Deliverable 3.1

Enhancing the autonomous smart objects and the overall system security of IoT based Smart Cities

Editor:	Dario Ruiz, ATOS
Deliverable nature:	Report(R)
Dissemination level: (Confidentiality)	Public (PU)
Contractual delivery date:	28-02-2015
Actual delivery date:	31-3-2015
Suggested readers:	Researchers, IERC, application developers, system administrators, security experts
Version:	1.3
Total number of pages:	160
Keywords:	End-to-end security, encryption, secure credential bootstrapping, digital signatures, ECDSA, JSON, OAP, ECC, dynamic monitoring, dynamic auto configuration, self-healing systems

Abstract

This document presents a detailed description of the RERUM Security architecture and its respective components that were described in RERUM Deliverable D2.3. The Security architecture can be split in three main parts, namely the Security, the Privacy and the Trust components. Privacy will be discussed in Deliverable D3.2 (due end of August 2015) and Trust will be discussed in D3.3 (due end of February 2016). Thus, the main objective of this document is to present a holistic platform to ensure the end-to-end and cross layer security within the RERUM system. For this respect, many security mechanisms are analysed, starting from the security of the devices and going all the way up to the applications. In this respect, the main techniques that are discussed in this document are: (i) symmetric and asymmetric cryptography, (ii) key management, (iii) transport layer security, (iv) on device signatures for integrity checking, (v) encryption key generation using RSSI measurements, (vi) authorization, (vii) secure network bootstrapping, (viii) secure auto configuration and (ix) self-monitoring of devices. The interactions between these components and the rest of the RERUM architecture are also presented here as extracted from D2.3 and revised as necessary. D3.1 goes beyond simple provisioning of a design of the security components providing a holistic view of the organization of the security mechanisms in RERUM and the technologies it is based on. For this purpose, D3.1 offers first an introduction with the whole picture of RERUM security and after this, it explains the technologies that are involved in the process, from the basement (cryptography background), over secure communication mechanisms, and to the upper level functions (authorization).

Disclaimer

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 609094

Impressum

Full project title	Reliable, resilient and secure IoT for smart city applications
Short project title	RERUM
Number and title of work-package	WP3 System & Information Security and trust
Number and title of task	T3.1 Secure object configuration and management T3.2 Overall system security
Document title	Enhancing the autonomous smart objects and the overall system security of IoT based Smart Cities
Editor: Name, company	Darío Ruiz, ATOS
Work-package leader: Name, company	Henrich C. Pöhls, UNI PASSAU

Executive Summary

Security is a major concern in any IoT deployment in smart cities, because the IoT applications are dealing with citizens' data, which have to be exchanged in a secure way to avoid malicious users intercepting them. Furthermore, no malicious users should be able to take control of the IoT system for their benefit. Insecure IoT deployments can decrease the trustworthiness of IoT infrastructures, hindering the adoption of this technology from both the citizens' and the service providers' point of view. Until recently the focus of IoT community was only on enabling the virtualization of the devices and the provision of services, not paying attention to the security of the IoT platforms. Due to this fact, various reports in the last few years discuss the need for excessive research in the security of IoT for addressing many device-related issues, i.e. lack of transport encryption, insufficient authentication, no possibility for remote updates of the software, etc. [HP14]. RERUM acknowledges this need and provides a holistic view of a security architecture that is tailored to the requirements of IoT deployments. This architecture is described in high level in RERUM Deliverable D2.3 and this document gives the design of the separate security components, with the exception of the privacy and trust components, whose design is described in a separate Deliverables (D3.2 and D3.3 respectively).

D3.1 is supposed to be both a report and a software prototype. The report (this document) provides the design of all the security components, plus a necessary theory background, while the software prototype provides an implementation of some of those security components that are considered to be essential for a proof of concept of the system.

D3.1 goes beyond simply providing the design of the security components, but it provides a holistic view of the organization of security in RERUM and the technologies it is based upon. For this purpose, D3.1 offers first an introduction with the whole picture of security in RERUM and after this, it explains the technologies that are involved in the process, from the lower (cryptography background) to the upper level functions (authorization).

The main outcomes of the document are:

- Algorithms for lightweight cryptography, generating encryption keys for Compressive Sensing using the Received Signal Strength Indicator (RSSI);
- Digital Signatures on Devices for data integrity and data origin authentication (which can support non-reputation): Protocol and first prototypical implementation of NIST curve P160 ECDSA signature of JSON encoded data on the device. This allows protecting the integrity of data flowing from, to or between RERUM devices. It furthermore allows identifying the origin of data by means of public keys. This works on any application level data (h-data), allowing broad use. The protocol and concept of on-device-signatures designed by RERUM and described in this deliverable is flexible and will allow RERUM to use different cryptographic signature mechanisms;
- Support of dynamic access control through a policy authorization engine: after a comparison of the main technologies that are applicable for access control, RERUM has adopted the approach of intercepting all requests incoming to the system and authorizing them individually following a design similar to the one proposed by the OASIS[1] [OASIS] standardization body, with a slight refinement when accessing user attributes. As a consequence, access control decisions in RERUM are carried out by evaluating XACML[2] [XACML13] policies;

[1] Standardization body that drives the development, convergence and adoption of open standards for the global information society

[2] XACML stands for eXtended Access Control Markup Language and is a standard language for defining access control criteria from the OASIS standardization body.

- Design of a fast and secure bootstrapping system to initialize the security credentials of all new nodes that are connected to the RERUM system;
- Integration of resource and network monitoring tools with a SIEM[3] to provide self-monitoring support: Both resource and network data are sent (via CoAP on UDP for a matter of efficiency) to a SIEM listener, which normalize them to a common message format understood by a SIEM tool. The SIEM tool executes some logic rules previously defined in its configuration. These logic rules take in consideration the normalized information and decide whether to trigger a security alert or not. In case a security alert if triggered, a reactor to this alert will take the corresponding actions based on the configuration of the SYSTEM for those specific actions and
- Automatic reconfiguration of security through the integration with a PRRS[4]: A PRRS allows defining a set of logical rules to decide whether it is necessary to upgrade the system, how and in which circumstances. By correlating system events, including those received from the SIEM and configuration rules, the PRRS is able to look for a suitable component in a trusted repository, build it if necessary and deploy the resulting object in the nodes of the system, communicating with the rest of RERUM components to upgrade the registry of the system with the changes performed.
- Datagram Transport Layer Security (DTLS) was brought onto the Zolertia Re-Mote: We developed a research prototype for Contiki and the Re-Mote platform. RERUM Devices are enabled to establish integrity and confidentiality (either end-to-end or hop-by-hop) at the transport layer level, including origin authentication. Additionally, we present the IEEE 802.15.4 security mechanisms that RERUM can use when securing low-level device communication.

[3] Security Information and Event Management

[4] Platform for Run-time Reconfiguration of Security

List of Authors

Company	Author	Contribution
ATOS	Darío Ruiz, Cristo Reyes	Final Editor, Introduction, Authorization, Dynamic Auto Configuration and Self Monitoring.
FORTH	Alexandros Fragkiadakis, Elias Tragos, Vasilis Siris, Apostolos Traganitis, Pavlos Charalampides	Secure Communication section, especially Profile lightweight and secure encryption using channel measurements; Network monitoring mechanisms. Editing and revising the document.
UNI PASSAU	Henrich C. Pöhls, Benedikt Petschkuhn, Johannes Bauer	Overview of Task interaction graphic. Description and graphics of Security Association; Cryptographic Background section; Background on malleable signatures, on integrity, on origin authentication, on Identity Based Signatures, on Implicit Certificates; Key Management; Secure Communication section: Introduction, Asymmetric Crypto and Profile On-Device-Signatures; Conclusion; Reviewing and Editorial Changes.
SAG	Kai Fischer, Jürgen Gessner, Santiago Suppan, Jorge Cuellar	Fast and secure network bootstrapping, background on group signatures, list of key material.
UNIVBRIS	Marcin Wójcik, George Oikonomou	Profile 802.15.4 Security, Profile DTLS, Comparison of Symmetric vs. Asymmetric Methods, Elliptic Curve Cryptography.

Version Control

Version	Company	Author	Change
1.0	RERUM Consortium	WP3 contributors	Draft release for internal contribution
1.1	ATOS	Darío Ruiz	Consolidation of contributor changes
			Harmonized introduction for Chapter 4
			Harmonized captions of figures
			Removed redundant numbers from references
			Moved references to OASIS and XACML to the Executive Summary
			Included caption for table 1
			Fixed wrong reference to Smart Object term in Section 5.2
		Cristo Reyes	Consolidation and contribution for harmonizing introduction for Chapter 5
			Included links to architecture in Section 5.3 and 5.4
	SAG	Kai Fischer , Juergen Gessner	Contribution to harmonize introduction for Chapter 5
	UNI PASSAU	Henrich C. Pöhls	Harmonized introduction for Chapter 2
			Harmonized introduction for Section 3.2
			Corrected and fixed wrong symbols in Chapter 2
	UNIBRIS	Marcin Wójcik	Harmonized introduction for Section 3.1 and 3.3
	FORTH	Alexandros Fragkiadakis	Harmonized introduction for Section 3.4
1.2	ATOS	Darío Ruiz	First consolidation of addressed reviewer's comments
			Added architecture figure in Section 4.1
			Added PDP figure in Section 4.4.2
		Cristo Reyes	Final check of consistency on structure and punctuation

	UNI PASSAU	Henrich C. Pöhls	Results and explanation of the research prototype added for Section 3.2
			Leading and editing the conclusion text with contributions from all contributors
			General review and editorial work, e.g. sorting the references
1.3	FORTH	Elias Tragos	Major editing in the document to correct formatting and language issues.
	UNI PASSAU	Henrich C. Pöhls	Major editing of the document. Writing conclusions.
	UNIVBRIS	Marcin Wójcik	Revision in section 2 and section 4
	LiU	Vangelis Angelakis	Revision in section 4

Table of Contents

Executive Summary 4
List of Authors 6
Version Control 7
Table of Contents 9
List of Figures 12
List of tables 14
Abbreviations 15
1 Introduction 18
1.1 Objective of this Document 18
1.2 Security by Design in RERUM 19
1.3 Structure of the Document 19
1.4 Overview of Security Associations (SA) in RERUM 21
1.4.1 Device Communication Layer 22
1.4.2 Service Layer 25
1.5 Overview of RERUM Security Mechanisms 25
2 Use of Cryptography in RERUM 27
2.1 Introduction, Motivation and Link to User Requirements 27
2.2 Background on Symmetric Cryptography with Focus on Constrained Devices 29
2.2.1 Integrity Using Symmetric Key Cryptography 30
2.2.2 Origin Authentication Using Symmetric Key Cryptography 30
2.3 Background on Asymmetric Cryptography with a Focus on Constrained Devices 30
2.3.1 Comparison of Symmetric vs. Asymmetric Methods 30
2.3.2 Use of Elliptic Curve Crypto (ECC) 31
2.3.3 Data Integrity Using Asymmetric Cryptography for 3rd-party verification (non-repudiation) without a fully trusted verifier 34
2.3.4 Origin Authentication Using Asymmetric Cryptography for 3rd-party verification (non-repudiation) without a fully trusted verifier 35
2.3.5 Notation and Background on Classic Public Key Based Digital Signatures 36
2.3.6 Notation and Background on Group Signatures 38
2.3.7 Notation and Background on Identity Based Signatures 40
2.3.8 Notation and Background on Implicit Certificates 41
2.3.9 New Harmonized Notation and Background on Malleable Signature Schemes 42
2.4 List of Key-Material needed in RERUM 56
2.4.1 Notation and Description of Key Material Needed and Generated during Credential-Boot-Strapping 57

2.4.2 Notation and Description of Key Material Needed and Used during Network Security Protocols 57
2.5 Key Management in RERUM 60
2.5.1 Certificate Chain Verification Overhead Reduced by Flat Hierarchy 60
2.5.2 Trusted Credential Store 64
3 Secure Communication 65
3.1 Profile DTLS 65
3.1.1 Introduction, Motivation and Link to User Requirements 65
3.1.2 DTLS Protocol 66
3.1.3 DTLS v1.2 Handshake 67
3.1.4 Analysis of the current state of software implementation 70
3.1.5 Summary of Profile 71
3.2 Profile On-Device-Signatures 71
3.2.1 Introduction, Motivation and Link to User Requirements 71
3.2.2 Analysis of the Current State and Selection of IETF Draft on JSON Web Signatures (JWS) 74
3.2.3 Description of Application in order to fulfil the security requirement 77
3.2.4 Research Prototype 77
3.2.5 Summary of Profile 79
3.3 Profile 802.15.4 Security 79
3.3.1 Introduction, Motivation and Link to User Requirements 79
3.3.2 Overview of Layer 2 Security in IEEE 802.15.4 80
3.3.3 Security Suites 80
3.3.4 IEEE 802.15.4 Security with the Contiki OS 81
3.3.5 Summary of Profile 81
3.4 Profile Lightweight and Secure Encryption Using Channel Measurements 82
3.4.1 Introduction, Motivation and Link to User Requirements 82
3.4.2 Compressed Sensing Background 84
3.4.3 Key Generation 85
3.4.4 RSSI Sampling 85
3.4.5 Performance evaluation 88
3.4.6 Summary of profile 90
4 Authorization in RERUM 91
4.1 Introduction, Motivation and Link to User Requirements 91
4.2 Service level authentication in RERUM 92
4.3 Analysis of authorization options 99

4.3.1 Option 1: Ad-hoc authorization provided by the application when registering in the system 99
4.3.2 Option 2: OAuth 100
4.3.3 Option 3: Policy based access proxy 102
4.3.4 Conclusions of the Analysis 103
4.4 Design of Authorization Components 103
4.4.1 Policy Enforcement Point (PEP) 104
4.4.2 Policy Decision Point (PDP) 108
4.4.3 Policy Retrieval Point (PRP) 109
4.4.4 Introduction Level 111
4.4.5 Authorization Policies Manager 113
4.4.6 Interaction with Other Modules 114
5 Secure RERUM Device Configuration 117
5.1 Introduction, Motivation and Link to User Requirements 117
5.2 Fast and Secure Network Bootstrapping 118
5.2.1 Related Technology 120
5.2.2 Initialization of the Security Center 121
5.2.3 Initialization and Bootstrapping of the RERUM Gateway 122
5.2.4 Distribution of the Join Key 123
5.2.5 Initialization and Bootstrapping of RERUM Devices 126
5.2.6 Credential Storage on RERUM Devices 129
5.2.7 Relation to RERUM Architecture 129
5.3 Secure and Context Aware Dynamic Auto Configuration 130
5.3.1 RERUM Device Initial Configuration 130
5.3.2 Integration with a PRRS 131
5.3.3 Over the Air Programming 134
5.4 Self Management and Self Monitoring Mechanisms 134
5.4.1 Network Monitoring Mechanisms 135
5.4.2 On-Device Resource Monitoring 138
5.4.3 Integrating a System Information and Event Management 139
6 Summary / Conclusion 148
Referencess 152

List of Figures

Figure 1: Overview of tasks and deliverables in WP3 and the most important links of D3.1 18
Figure 2: Overview of two different layers and different Security Associations (SA) 23
Figure 3: Overview of two different layers and different Security Associations (SA) v2 24
Figure 4: Architectural Layers of RERUM from Deliverable D2.3 29
Figure 5: Doubling a point (left) and adding two points (right). 32
Figure 6: Group Signature Scheme with linking functionality (adapted from [MF12]) 39
Figure 7: Certificate chain 41
Figure 8: Unforgeability Experiment for RSS 48
Figure 9: Unforgeability Experiment for SSS 48
Figure 10: Weak Blockwise Non-Interactive Public Accountability Experiment for SSS 48
Figure 11: Standard Privacy Experiment for SSS 49
Figure 12: Strong Privacy Experiment for SSS 49
Figure 13: Weak Privacy Experiment for RSS 50
Figure 14: Strong Privacy Experiment for RSS 50
Figure 15: Immutability Experiment for SSS 51
Figure 16: Experiment for SSS weak immutability 51
Figure 17: Use of Implicit Certificates in a flow of messages to allow obtaining key material for a later Device to Device authentication 63
Figure 18: Security components in the communication layer from D2.3 65
Figure 19: Overview of the handshake protocol 67
Figure 20: Prototype of DTLS on Re-Mote platforms 70
Figure 21: RERUM architecture (Fig. 29 from deliverable D2.3) with On-Device Signatures 73
Figure 22: On-Device-Signatures prototype implemented using Zolertia Z1's by UNI PASSAU 78
Figure 23: Key extraction from the wireless channel with the presence of an eavesdropper 85
Figure 24: Bit mismatch rate between the keys of Alice-Bob and Alice-Eve 87
Figure 25: Reconstruction error at Bob or Alice for different quantization levels 89
Figure 26: Bit mismatch rate for different quantization levels 89
Figure 27: Reconstruction error at Eve for different quantization levels 90
Figure 28: Security components in the communication layer from D2.3 91
Figure 29: Registering users 94
Figure 30: Retrieving a security token for later use 96
Figure 31: Authenticating users on each startup of the application 97
Figure 32: Reusing a security token already retrieved 98
Figure 33: Managing security policies 104
Figure 34: PEP components and their relationship with PDP 106

Figure 35: Interaction of PEP components with the rest of the system 106
Figure 36: Nomenclature for request contents to be included in the XACML context..... 108
Figure 37: PDP related classes..... 108
Figure 38: Interaction PDP-PRP 111
Figure 39: Obtaining security token through the IdA and using it 113
Figure 40: Managing security policies 114
Figure 41: PEP interaction with services 115
Figure 42: Network of RERUM Devices 119
Figure 43: Initialization of RERUM gateway 122
Figure 44: Pre-installed join key 124
Figure 45: No pre-installed join key 125
Figure 46: Bootstrapping 802.15.4 network key and the keys to communicate to GW and SC 127
Figure 47: Setup of IP communication and bootstrapping further credentials 129
Figure 48: Credential Bootstrapping Client / Authority and related security components 130
Figure 49: Configuration manager components from D2.3 130
Figure 50: PRRS components and workflow 131
Figure 51: PRRS rule web editor snapshot 133
Figure 52: Monitoring manager components from D2.3 134
Figure 53: Hierarchical network management and monitoring..... 137
Figure 54: SIEM components and workflow 140
Figure 55: Events sequence diagram..... 140
Figure 56: SIEM Action designer 146
Figure 57: OSSIM Policy designer 147

List of tables

Table 1: Location of the functionalities described in the DOW within this document 26
Table 2 Network Monitoring Information 138
Table 3 On-Device Resources, their Paths and Resource Types 139
Table 4: Requirements addressed in this document 150

Abbreviations

Acronym	Meaning
ABAC	Attribute Based Access Control
AES	Advanced Encryption Standard
API	Application Programming Interface
APP	Application
CA	Certificate Authority
CBC-MAC	Cipher Block Chaining Message Authentication Code
CDF	Cumulative Density Function
CES	Content Extraction Signature
CCM	Counter with CBC-MAC
CoAP	Constrained Application Protocol
CRC	Cyclic Redundancy Check
DICE	DTLS In Consternated Environments
DoS	Denial of Service
DOW	Description Of Work
DLP	Discrete Logarithm Problem
DSS	Digital Signature Scheme
DTLS	Datagram Transport Layer Security
D(n).(n)	RERUM Deliverable n.n
ECC	Elliptic Curve Cryptography
ECDH	Elliptic-curve-Diffie-Hellman
ECDLP	Elliptic Curve Discrete Logarithm Problem
ECDSA	Elliptic Curve Digital Signature Algorithm
ECQV	Elliptic Curve Qu-Vanstone
EO	Electricity Operator
HMAC	Hashed message authentication code

GW	Gateway
HSM	Hardware Security Module
HTTPS	HyperText Transfer Protocol Secure
ID	Identification
IdA	Identity Agent
IETF	Internet Engineering Task Force
IoT	Internet of Things
IPSO	IP for Smart Objects
ISO	International Organization for Standardization
JSS	JSON Sensor Signature
JWS	JSON Web Signature
MAC	Message Authentication Code
MSS	Malleable Signature Scheme
NIST	National Institute of Standards and Technology
LQI	Link-Quality-Indicator
OSSIM	Open Source Security Information Management
PEP	Policy Enforcement Point
PDP	Policy Decision Point
PFS	Perfect-Forward Secrecy
PRRS	Platform for Run-time Reconfiguration of Security
PRS	Policy Retrieval Point
RD	RERUM Device
RERUM	REliable, Resilient and secUre IoT for sMart city applications
REST	REpresentational State Transfer
RGW	RERUM GateWay
RSA	Rivest-Shamir-Adleman algorithm

RSS	Redactable Signature Schemes
RSSI	Received-Signal-Strength-Indicator
SA	Security Association
SbD	Security by design
SO	Smart Object
SOAP	Simple Object Access Protocol
SNMP	Simple Network Management Protocol
SSH	Secure Shell
SSS	Sanitizable Signature Scheme
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TOCTOU	Time-of-check Time-of-use
TTP	Trusted Third Party
UDP	User Datagram Protocol
VRD	Virtual RERUM Device
WSN	Wireless Sensor Networks
XACML	eXtended Access Control Markup Language

1 Introduction

Deliverable D3.1 consolidates the output of two tasks in the work package on System & Information Security and Trust. The output of these tasks consists of conceptual work on security components and initial prototypes of some of those components. D3.1 presents the design of the security components defined previously in the D2.3, excluding the ones focussing on privacy, which will be presented in D3.2. See Figure 1 for some interconnections.

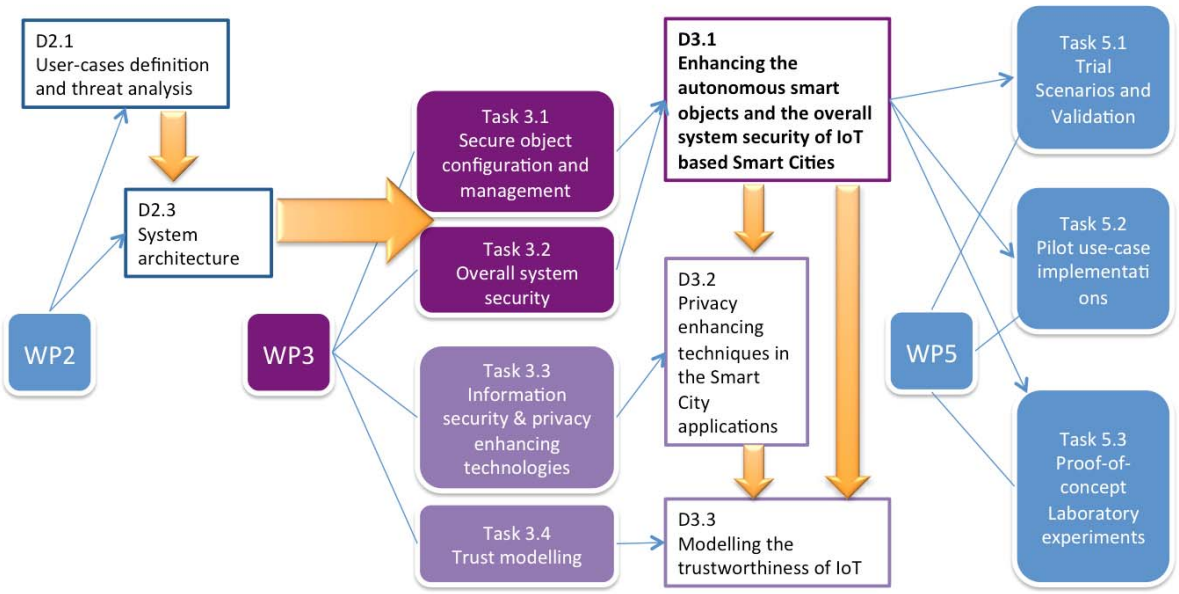


Figure 1: Overview of tasks and deliverables in WP3 and the most important links of D3.1

The prototype itself is an implementation of a subset of these designs provided in this document. However, the prototype does not implement all the designs, but only those that were considered essential to provide a proof of concept of RERUM. The conclusion of this deliverable offers an overview of which components have been implemented, and what security requirements have been addressed to ease checking requirement fulfilment.

Because the document covers (among other things) the work done on some novel procedures on cryptography and compression, it is necessary to have a deep technical understanding of those very mathematical parts. However, the documents tries to provide tutorial summaries and reduce the required expertise of readers as much as possible while also providing all necessary technical background needed to understand it to the consortium partners familiar with that level of technicality.

1.1 Objective of this Document

The main objective of this document is to explain and document the overall design of the security elements facilitated in RERUM. These are complex topics and this deliverable presents them such that it provides a holistic view of RERUM's security. Explaining the technical details of the work carried out in RERUM in the security. The result is RERUM's solutions for mechanisms and protocols for:

- Authentication
- Authorization
- Data integrity

- Confidentiality
- Non Repudiation
- Self configuration
- Self management
- Bootstrapping of the network
- Integration with a PRRS[5]

The document is not going to present these features individually, but providing the holistic and easy to understand view of the security components in RERUM. Hence, it is not an objective of the document to necessarily explain these functionalities in this order, but in whatever way that better contributes to understand the whole picture of security in RERUM.

Finally, D3.1 is meant to focus on security components, and privacy components are left outside of it. The reason is not that privacy is not a security part, but It is a so complex topic that it is dealt with in a separate deliverable (D3.2). As a result, some of the components described in the future D3.2 will extend ones already presented in this document, whose design does not cover the details of privacy preserving features, yet.

1.2 Security by Design in RERUM

Security by design (SbD) is not a formal specification that can be objectively checked but a general philosophy instead. In general terms, SbD can be described as the property of a system to have taken into account any possible security issues from its conception and has been designed to make a reasonable effort to deal with them.

In RERUM, security has been considered from the very conception of the project, which was reflected in the inclusion on a whole work package (WP3) in the DOW document devoted to security, privacy and trust.

In practice, security in RERUM has been taken into account by:

- Providing a threat analysis in the D2.1 document to study the assets of the system and the possible attacks to them;
- Including a set of security components in the architecture document D2.3 to deal with the threats identified in D2.1;
- Designing and implementing a prototype for the main security components defined in D2.3 through the tasks of Work package 3 and
- Evaluating the security components in Work package 5.

In fact, this document describes the prototype of the security components of the system, covering the authentication and authorization part. Privacy techniques will be described in D3.2 and a model for trustworthiness in IoT will be presented in D3.3.

1.3 Structure of the Document

As seen in the introduction, this document covers 10 different topics. But these topics are related with each other and often rely one on another. And each of them can have very complex issues, too many of these features are built on top of several of the others and it is necessary to explain the technologies of the basement before explaining the ones in the top.

[5] Platform for Run-time Reconfiguration of Security

To provide a consistent structure that is also easy to read this document follows the order of the technologies that are used in it. In short (this will be extended later), mechanisms and tools that the security components use are built this way:

- Cryptography building blocks
- Securing communications at network and data-transport level, built on top of cryptography technologies
- Authorizing requests, built on top of secure communications
- Configuration components for the previous components

After providing this overall picture, the document proceeds to explain the technologies mentioned.

In short, this is how the document is organized:

- Chapter 1: Introduction explains what is necessary to fully understand the document, including an overview of security associations and an overview of the RERUM security mechanisms. The overview of security associations explains the different levels of security in the system and their associations with the rest of the document. The overview of the RERUM security mechanisms provides a short resume of the mechanisms finally adopted in RERUM;
- Chapter 2: Use of cryptography in RERUM provides the background on cryptography technologies that are used by the rest of security components;
- Chapter 3: Secure Communication explains how the cryptography is used to secure the communications at network level;
- Chapter 4: Authorization in RERUM explain how the requests are authorized, relying in the security of the communications already presented in Section 3;
- Chapter 5: Secure RERUM device configuration explain how the components that govern the configuration of the previous ones are built in the system;
- Chapter 6: Summary / Conclusions offer a resume of which security requirements are covered and in which section of this document they are described.

Finally, readiness is not the only main objective of the document. The document presents all the work carried out in task 3.1 and task 3.2 in RERUM. This includes novel algorithms and protocols. As these are heavily technical, they need to be explained with a technical language too and also refer to the basis of cryptography to understand them. For this reason Chapter 2 Use of Cryptography in RERUM provides all necessary background for a technician to understand these cryptographic mechanisms. The mechanisms described in Chapter 3 Secure Communication are built using primitives described and defined in Chapter 4 Authorization in RERUM is based on Chapter 3 Authorization in RERUM; Chapter 5 Secure RERUM Device Configuration is built on all of them. Summarizing, the document tries to be as readable as possible but explaining the novel algorithms and protocols require a technical background. D3.1 provides exactly that technical background aiming for a technician to be able to understand RERUM's developments, but it has not been possible to ease readability beyond this point.

1.4 Overview of Security Associations (SA) in RERUM

Following RFC 2409[6] a security association (SA) is a set of policy and cryptographic key(s) used to protect information. In Figure 2 we have depicted the entities for which D3.1 will describe mechanisms to secure information exchanges.

They are as follows:

- Human user with an Application
- Application Server, when the Application has an additional backend that does processing (storage, computing) of the information
- RERUM Gateway
- RERUM Device(s)

Please refer to previous deliverables (e.g. D2.3 for details) or definitions for more information on each entity.

A SA can be usable for the establishment of secure communication in order to reach one or several specific security goals (integrity, origin authentication, confidentiality). This means that a security association enables, once established with suitable key material, to cryptographically protect communication between those entities, such that security goals are reached.

SA's can be hop-to-hop or end-to-end. In the case of hop-to-hop associations, the involved entities are those that can directly communicate with each other on the lower network layers, allowing using lower level security mechanisms.

Further, SA can be mutual or unilateral. A mutual SA allows offering the protection for both entities in a mutual fashion, e.g. a mutual SA for origin-authentication and integrity would allow both communicating entities of the SA to establish that a received message has not been changed in an unauthorised way and has been generated by the other entity. If the SA is unilateral, then only one entity in the communication can use the unilateral mechanism to protect the information based on that SA.

Let us give an example: For an unilateral SA between a SERVER and a RERUM GW, allowing the RERUM GW to identify messages as authentic if they are sent from the SERVER, we could build a SA based on the SERVER's public key certificate being trusted and residing on the RERUM GW, while the SERVER would be able to sign messages using the SERVER's private key. Thus, this set of keys would allow authenticating the origin of messages to be the SERVER. The same type of keys at the same entities could also be used to build a unilateral confidentiality SA, allowing establishing confidentiality protected communication in which the RERUM GW could use the SERVER's public key to encrypt messages towards the SERVER. Note, it is cryptographically not advisable to use the same key pair for encryption and signing.

The example for a hop-to-hop SA is "SA:RD1-to-RD3" from Figure 2 which is equal to the direct communication link (h a hop) on network between the two RERUM Devices #1 and #3. So if this SA is used to prove confidentiality, then the SA on RD#1 would for example be comprised of two keys, and the policy, such that RD#1 has one encryption key that RD#1 uses to encrypt communication before sending it to RD#3, and a decryption key that RD#1 is using to decrypt communication received from RD#3. The same SA on RD#3 also needs corresponding keys.

[6] RFC 2409 defines Security Associations in the context of internet key exchange protocols, e.g. ISAKMP, Oakley, SKME.

To provide another example, let us assume we want “SA:RD3-to-RGW” from Figure 2 to provide confidentiality, integrity and authentication of origin for a communication between the RERUM Gateway and the RERUM Device #3. You clearly see that a possible security channel between those entities spans across several hops, i.e. direct communication links. Communication paths are either Hop#1+Hop#2 or Hop#1+Hop#3+Hop#4. For this reason the SA:RD3-to-RGW for confidentiality and integrity and origin authentication must be end-to-end and not hop-to-hop. Again this SA will need cryptographic key(s) on which the security mechanisms will be able to build secure communication channels.

More details on the different security profiles and the underlying mechanism to build all these SA are found in this deliverable of RERUM, in Chapter 3 Secure Communication. More information on the cryptographic mechanisms and the keys can be found in Chapter 2 Use of Cryptography in RERUM.

After having distinguished SA into end-to-end and hop-to-hop, we will briefly describe how RERUM further differentiates between two layers on which SA are being formed to secure communication within RERUM.

1.4.1 Device Communication Layer

Device communication is logically concerned with addressing and connecting to devices, or between devices. Here RERUM foresees to extend, integrate and facilitate device centric security mechanisms, e.g. end-to-end encryption and signature mechanisms, but also transport layer security such as 802.15.4 security [IEEE802.15.4].

Figure 2 shows a typical scenario with a user running a standalone application that is trying to access a RERUM device. As the figure shows, here we have security associations from inside RERUM and outside RERUM. The security associations from outside RERUM are carried out from the Internet, and hence use standard Internet mechanisms, such as TLS[7].

[7] Transport Layer Security (TLS): Is a cryptographic protocol designed to provide communications security over a computer network widely used in the internet and replacer of previous cryptographic protocol SSL

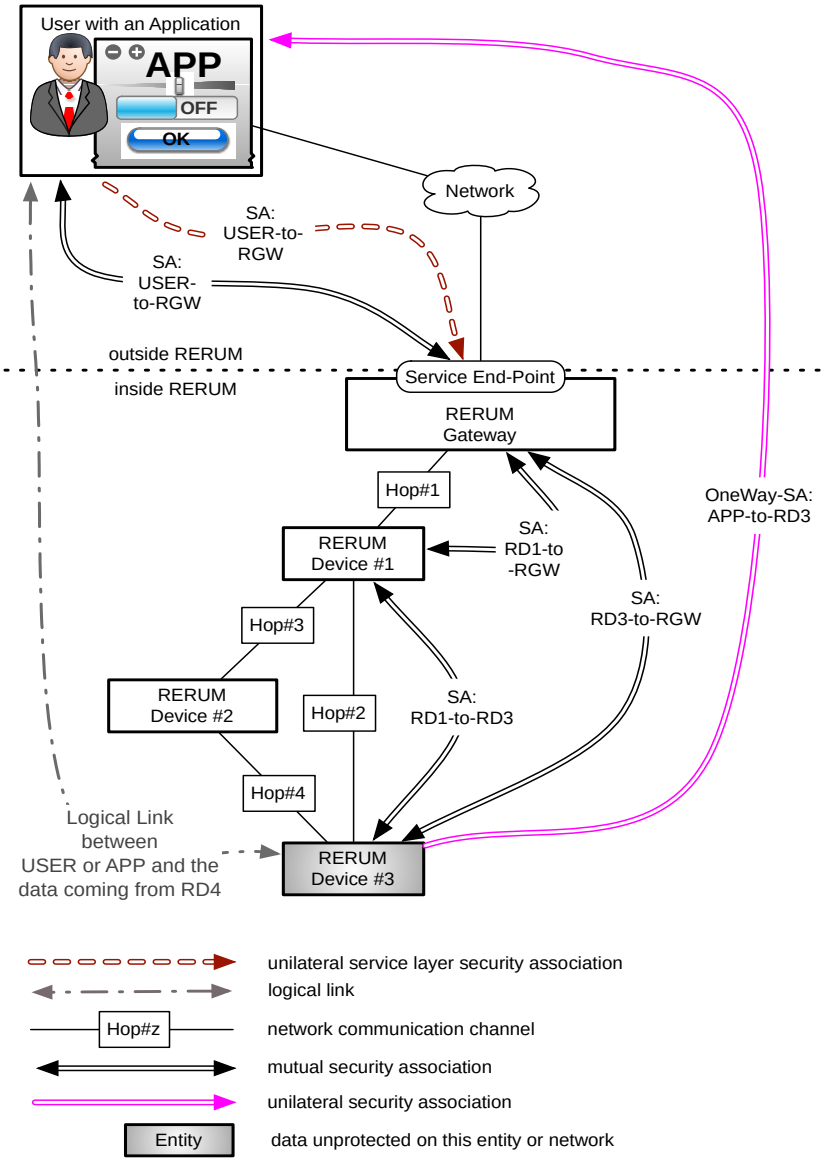


Figure 2: Overview of two different layers and different Security Associations (SA)

Security associations inside RERUM are carried out at device layer using, the mechanisms defined in this document.

But applications are not necessarily standalone applications. They can also be divided in two parts, a front end, which often runs in a browser and a backend running on an application server that is being run on a different system. Though this case is equivalent for RERUM because the difference only occurs outside RERUM. Figure 3 which is a slightly different from the former, illustrates that case.

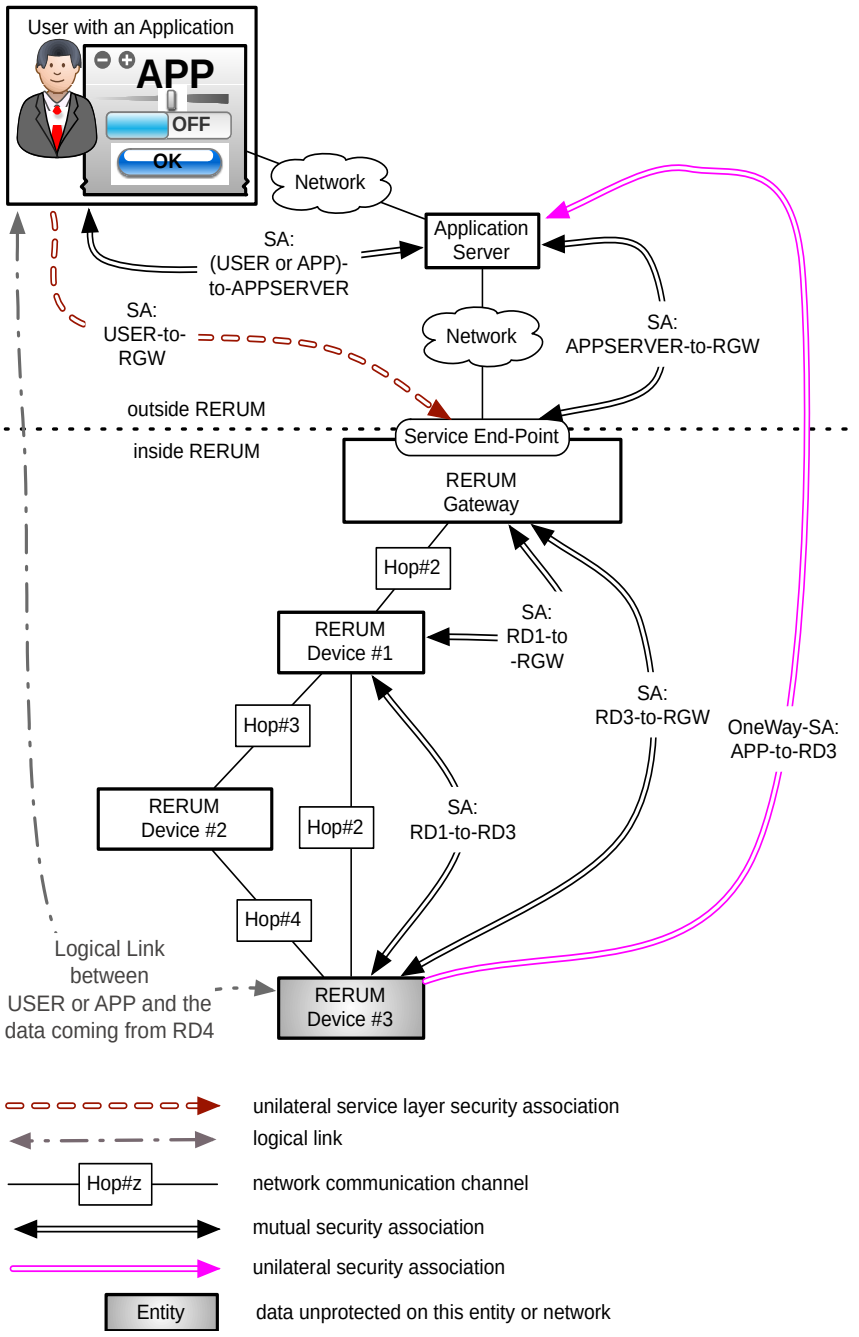


Figure 3: Overview of two different layers and different Security Associations (SA) v2

1.4.2 Service Layer

The Service Layer is logically concerned with the addressing, connection and communication between services, which are usually described using Web Services[8], exchanging data with SOAP[9] messages [SOAP07] or JSON[10] objects [JSON14], and have protocols like REST[11] [ELKS08].

Here RERUM plans to adopt this concept and integrate additional security mechanisms into existing communication infrastructures and protocols. This way RERUM is also able to leverage a lot of existing "Internet Security" mechanisms, e.g. TLS and HTTPS[12].

Summarising, plus the already mentioned mechanisms, RERUM provides security at service layer in the Security association between the application and the Gateway. That is, all incoming requests from the Internet are secured not only at device level, but also at service layer. Specifically, RERUM defines user registering and authentication mechanisms in Section 4.2 Service level authentication in RERUM, and authorization mechanisms that depend on the service accessed, which are covered in Section 4.4 Design of Authorization Components. These security associations are already shown in the previous figures in the red dotted arrows.

1.5 Overview of RERUM Security Mechanisms

At device communication layer, this deliverable contains the technical description of the following security mechanisms:

- Communication Security Profile: DTLS
- Communication Security Profile: on-device-signatures
- Communication Security Profile: 802.15.4 Security
- Communication Security Profile: Lightweight and secure encryption using channel measurements
- Design of bootstrapping components
- Design of automatic monitoring components at network and device level

At service layer, this deliverable contains the technical description of the following security mechanisms:

- Definition of User authentication procedures
- Design of authorization components
- Design of automatic monitoring components at service level
- Design of a Platform for Run-time Reconfiguration of Security

In short the functionalities mentioned in the DOW and their allocation in the document is shown in the following table:

[8] Software function provided over the internet.

[9] Simple Object Access Protocol: Protocol for exchanging structured information in web services.

[10] JavaScript Object Notation, format that uses human-readable text to transmit data consisting of attribute–value pairs

[11] Representational State Transfer: Set of guidelines for creating scalable web services

[12] HiperText Transfer Protocol Secure: Protocol for secure communication over a network

Table 1: Location of the functionalities described in the DOW within this document

Functionality	Chapter / Section
Authentication	Chapter 3 at network level
	Section 4.2 at service (user) level
Authorization	Chapter 4
Data Integrity and Non Repudiation	Chapter 3
Confidentiality	Chapter 3
Bootstrapping	Section 5.1
Self configuration	Section 5.2
Self management	Section 5.3

Note, it is not possible to provide a similar table for the innovations of D2.1 here because it would be necessarily incomplete, as the innovations of RERUM are not necessarily covered all in this document. Instead, each section that cover any innovation state so in their corresponding introduction section.

2 Use of Cryptography in RERUM

2.1 Introduction, Motivation and Link to User Requirements

This chapter gives the results of the analysis and provides background on existing security methods and related work in the area of cryptographic schemes.

This is done in order to provide RERUM with a joint terminology and provides the basis to understand how RERUM will use cryptographic methods and key material. In this chapter RERUM states the chosen symmetric and asymmetric methods and gives a list of what key material will be used or needed for these different methods.

RERUM's goal is to provide security on the basis of strong cryptographic primitives. These become the building blocks to allow secure communication and secure authorization. The communication security in RERUM is logically grouped around security associations (SAs) between communicating entities. They can be used to establish secure communication, which protect integrity, origin authentication and confidentiality. This will be described in Chapter 3 Secure Communication. However, the SAs are also used for authorization that is explained in Chapter 4 Authorization in RERUM.

This deliverable (D3.1) is focused on security rather than privacy; hence, the three goals of confidentiality, integrity and authorization are the security goals for which cryptographic mechanisms will be described here. However, security is required to achieve privacy. For example to make data only available to services that the user consented to, we need to authenticate the services, and we need to encrypt the data on the transport to those authorized services. Thus, without these basic security functionalities established in the IoT, solutions that can offer privacy are not possible at all. In short security mechanisms are a pre-requisite to privacy. However, RERUM's focus on privacy shows already in this deliverable, as general cryptographic mechanisms that will offer an increase in privacy are already mentioned here, e.g. Group Signatures or Malleable Signatures. More details on the application and adjustment of those cryptographic primitives will be provided in deliverable D3.2. Stating them already in D3.1 is done for several reasons: (a) to disseminate RERUM's first findings and enhancements of the current state of the art, (b) to document an agreed terminology and provide a project wide notation, and (c) to ensure and demonstrate that RERUM's mechanisms are flexible and can indeed be implemented using different cryptographic primitives that achieve the minimal desired properties. The latter ensures that RERUM's mechanisms are designed to be adaptable towards future cryptographic developments.

Following the requirements from deliverable D2.2, it is important that RERUM can provide mechanisms to support secure communication in the complete IoT device chain, from the devices all the way to the applications. RERUM's goal was to use cryptographically strong protection methods and this chapter provides the necessary cryptographic notation, background and insights. To address this need for suitable and configurable security, to allow protecting access to the system is universal in IoT, the mechanisms of this chapter are applicable to all use cases in the project and can be validated in all trials.

Thus, this chapter provides the background that is needed to cryptographically address many of the requirements listed in D2.2 Section 2.6 on Security and Privacy Requirements:

- Req. 2.4-10 Low energy consumption
- Req. 2.4-11 Lightweight dynamic data compression
- Req. 2.4-12 Over-the-Air Programming
- Req. 2.6-1 Energy-efficient cryptographic primitives

- Req. 2.6-2 Integrity protection of SL-I data in transit
- Req. 2.6-3 Integrity protection of SL-I data at rest
- Req. 2.6-6 Confidentiality protection of SL-C data at rest
- Req. 2.6-7 Confidentiality protection of personal data in transit
- Req. 2.6-8 Device authentication
- Req. 2.6-9 User authentication
- Req. 2.6-10 Attribute-based access control
- Req. 2.6-19 Secure bootstrapping of operational cryptographic credentials
- Req. 2.6-20 Availability of initial credentials
- Req. 2.6-21 Support of different operational credentials types
- Req. 2.6-22 Avoidance of manual interactions during credential bootstrapping
- Req. 2.6-23 Update of operational credentials
- Req. 2.6-24 Find deployable software to RERUM devices
- Req. 2.6-25 Object configuration isolated per application
- Req. 2.6-26 Secure design and implementation of RERUM components

In more detail, you will find these links again in the sections of this deliverable that describe the secure communication profiles or the authorization.

This chapter also shows RERUM's clear focus to select cryptographic mechanisms that are suitable to be executed on a RERUM device itself, which includes constrained devices. For this reason selected mechanisms will be subjected to lab experiments to execute in RDs, and if successful potentially be implemented into the trials. This also means that the mechanisms described in this chapter will be found at many places inside the RERUM architecture, as seen in Figure 4. Security is in effect in every layer, allowing to be enabled in the complete framework, fulfilling the "by design" requirement. Thus, RERUM supports use cases that need security by design, and if not required for a use case, it could be configured to suit the use case's needs.

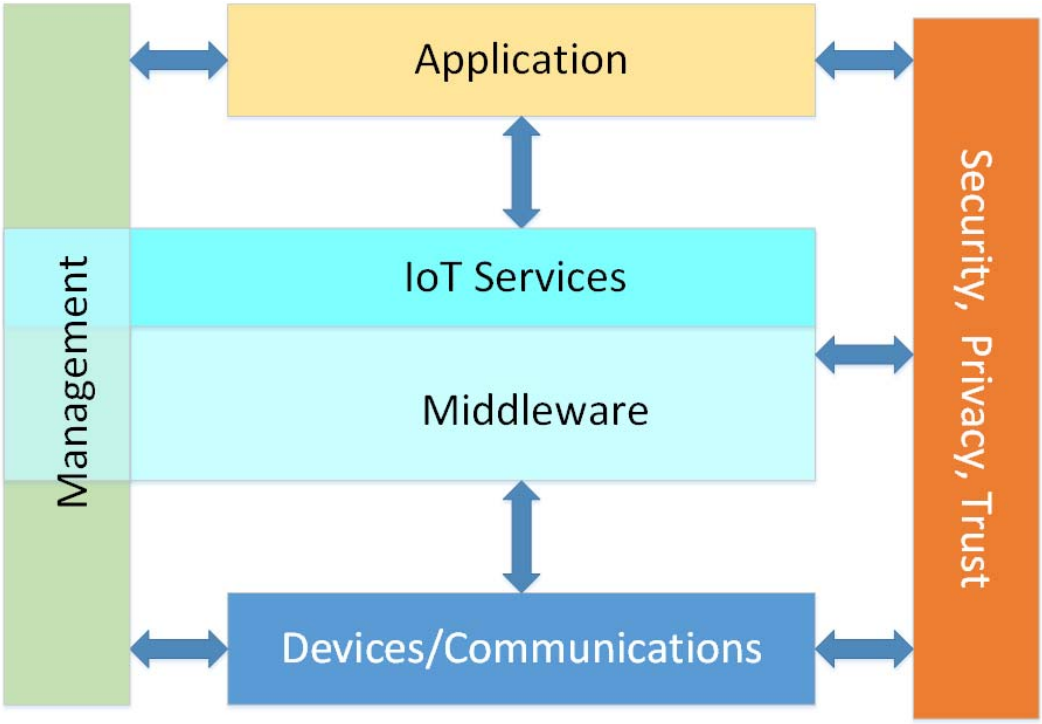


Figure 4: Architectural Layers of RERUM from Deliverable D2.3

This Chapter is organized in the following parts:

- Section 2.2 gives background on Symmetric and Asymmetric Cryptographic Mechanisms
  - The focus is on those mechanisms that RERUM selected as suitable for constrained devices, as RERUM seeks the possibility to provide end-to-end security, where one end is a constrained device.
  - The selected candidate cryptographic schemes and protocols are presented by stating the notation and description of the algorithms and if available software interfaces.
  - User Authentication at device level in RERUM is also explained. Here, RERUM will rely on existing user authentication approaches of the Internet space. Further research is not in the scope of RERUM.
- Section 2.3 lists the Key-Material. This follows the analysis of the required security associations and lists what kind of cryptographic material will be required by RERUM. This list is not meant to be exhaustive, but shows the barely needed keys.
- Section 2.4 explains briefly how the keys will be managed and stored for different security goals to be reached in RERUM.

2.2 Background on Symmetric Cryptography with Focus on Constrained Devices

RERUM warns that symmetric key, while it might be more efficient, always bears the risk of key-compromises having a severe impact. Shared keys are shared between several entities and could be extracted from devices if they get captured, so called node-capture. In order to reduce this attack, shared keys shall be only used for a short time, e.g. initial setup or as shorter-lived session keys.

2.2.1 Integrity Using Symmetric Key Cryptography

When protecting against unauthorised changes, e.g. tampering of messages in transit, mechanisms based on symmetric key cryptography require the integrity protection application and the verification to use the same key. Hence, the integrity protection cannot only be generated by the party that has applied the integrity protection mechanism to the data, but it can be generated (using the shared key) by all the parties that can access the shared key in plain. Access to that shared secret key is also needed by the integrity verification algorithm, which is executed by the recipient of the data when wanting to verify the data's integrity. The only way to prohibit the verifier from getting access to the shared secret is to embed the verification algorithm and the shared verification key inside a HSM[13]. Without this protection from a HSM a valid integrity check result corroborates that the integrity protected data has not been modified (or destroyed) in an unauthorised manner since the integrity protection mechanisms has been applied to the data by some `outside` party that has no access to the shared secret. These mechanisms are called Message Authentication Codes (MAC).

The state of the data's integrity cannot be proven by a verifier to a third party without such a HSM. Hence, RERUM calls this internally verifiable integrity protection. Internally as it does not build evidence that can be used to cryptographically convince a third party unless a HSM is used.

RERUM considers the use of MAC whenever there is a need for efficient exclusion of outsiders. RERUM might consider the use of HSM, e.g. as in [AF12] at later development stages.

2.2.2 Origin Authentication Using Symmetric Key Cryptography

A symmetric key based cryptography origin authentication could internally establish that the message originated from an entity that knows the shared secret. Internally as it does not build evidence of the data's origin that can be used to cryptographically convince a third party unless a HSM is used. Just like for integrity a MAC, described in 2.2.1, this means that the verifier can also generate the origin authentication on arbitrary messages, if he has access to the key and it is not inside a HSM which restricts use to the verify algorithm and does not leak it. RERUM considers the use of these origin authentications, which are usually very efficient, whenever there is a need for efficient differentiation between data / messages from insiders (know the shared secret) and malicious data / messages from outsiders (not knowing the key).

2.3 Background on Asymmetric Cryptography with a Focus on Constrained Devices

2.3.1 Comparison of Symmetric vs. Asymmetric Methods

At a very general level, one can divide cryptographic methods onto two groups, namely symmetric-key (private-key) and asymmetric-key (public-key) schemes. The first group, i.e. the symmetric-key group, typically includes schemes that, in order to encrypt and decrypt messages, use identical (or a trivially related) keys [MOV01]. Considering scenario, where two parties would like to communicate with each other and where data confidentiality is required, these keys (or a key) have to be exchanged (shared) beforehand via some trusted communication channel. In this category, two underlying classes of algorithms might be distinguished, i.e., block and stream ciphers. The former ones take as an input the fixed-size blocks of a plaintext and produce the fixed-size blocks of a ciphertext. The latter ones calculate (using a key) so-called key stream. This key stream is further combined (using a logic XOR

[13] Hardware Security Module: A hardware component specialized in performing security operations, supposedly much faster than a program executed by a non-specialized computer

operation) with a plaintext producing a ciphertext. One of a very well-known and commonly used in practices representative of the symmetric-key methods is AES block cipher [FIPS01], standardised by NIST in 2001.

Irrefutable advantage of symmetric-key cryptography (in comparison to public-key methods) is an ability to maintain a high data throughput using considerable low computational overheads and relatively short keys. On the other hand, as mentioned before, a shared key needs to be exchanged via trusted channel before communication and usually have to be unique for each communication channel. These features imply in some cases much more complicated key management techniques than those used in public-key counterparts.

The second group, i.e., asymmetric-key schemes, was introduced by Diffie and Hellman in [DH76]. In general, those schemes are based on the idea of using so-called trapdoor one-way functions, in which one can relatively easy compute the output, whereas is hard to compute the inverse of the function without knowing the trapdoor. Based on that fact, it is possible to design a scheme that uses two related keys: a private and public key pair. The private key is kept secret, whereas the public key can be distributed via untrusted and open communication channel. The public key is generated from the private key by applying above-mentioned trapdoor function. Such a construction allows one to encrypt data using other party public key and allows only a party that possesses the private key to preform decryption procedure successfully. The first practical and well-known algorithm that belongs to this category is called RSA [RSA78] and was introduced shortly after the theoretical work of Diffie and Hellman.

The main advantages of asymmetric-key methods (over symmetric-key schemes) are the fact that both parties can perform communication without a need of sharing a symmetric-key, which has also a positive impact on a complexity of key management procedures. Disadvantages however lay in relatively low data throughput and larger key sizes. Asymmetric-key schemes are also usually more computationally costly when compared to symmetric-key counterparts.

It is worth mentioning that in practice, especially in secure protocols, symmetric- and asymmetric schemes might be used in combination, mitigating their disadvantages, i.e., a public-key scheme might be deployed to exchange a symmetric-key, which is further used to protect communication data. Such a synergy has been successfully deployed in well-known protocols such as TLS/DTLS or SSH.

2.3.2 Use of Elliptic Curve Crypto (ECC)

Elliptic Curve Cryptography (ECC) is an alternative (to well-known and commonly used in practice RSA algorithm) technique that supports public-key schemes. Applications of elliptic curves in cryptography were presented independently by Miller [M86] and Koblitz [K87] for the first time in 1986. Since then, ECC became a subject of intensive studies from both theoretical and practical points of view and until now, it is considered to be secure and efficient approach to public-key cryptography.

An elliptic curve E over a field K is defined as a set of solutions (x,y) of the equation given by the formula E: $y^2+a_1xy+a_3y = x^3+a_2x^2+a_4x+a_6$, where a_1, a_2, a_3, a_4, $a_6 \in K$ and where K is an arbitrary field. The equation above is given as an affine version of a Weierstrass equation, whereas for cryptographic applications, K is usually set to be a finite field F_q, where $q = p^n$, p is a prime number and $n \in N$. Let denote E(K) as a set of K-rational points (the solutions of the equation above) and let include (in case of affine representation) the point at infinity O. This set, together with defined group law operations called chord-tangent operations forms an abelian group, where the point at infinity acts as an identity point of said group.

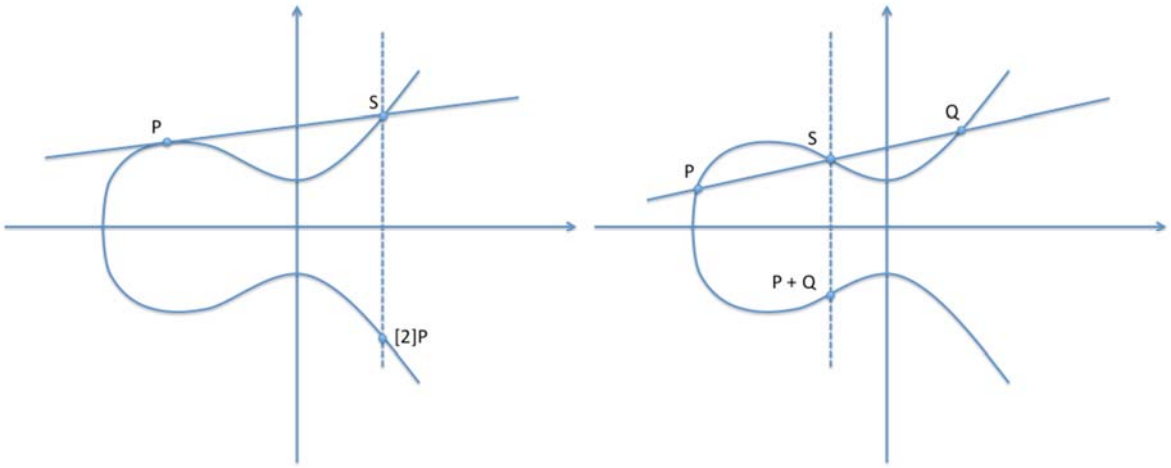


Figure 5: Doubling a point (left) and adding two points (right).

Let assumes that P and Q are elliptic curve points. In order to add points or multiply point by a scalar, the elementary group law operations, namely tangent and chord rules, have to be applied. The first, tangent operation is depicted in Figure 5, (here elliptic curve E is over real numbers). In such a case, one is able to double the point P, i.e., find a P+P (or [2]P in commonly used notation) point. Firstly, the tangent to the point P intersects the curve in point S. Secondly, having the point S, the desired point [2]P is the x-axes reflection of the point S. Similarly (see Figure 5) for chord rule, i.e., to find P+Q for a given P and Q, a line between P and Q intersects curve in a point S and again, an x-axes reflection of the point S is a desired point P+Q.

Using the chord rule iteratively, one is able to compute the basic elliptic curve cryptography operation, namely a point scalar multiplication. This operation is defined as [n]P, which is defined as an addition of point P repeated n-1 times, i.e., Q = [n]P = P+P+P+...+P. The defined scalar point multiplication plays the principle role in modern cryptography. It is very easy to compute [n]P for a given n and P, but it is believed to be hard to find the invert of such an operation, i.e., for a given point P and a given point [n]P it is hard to recover n. The aforementioned problem is known as Elliptic Curve Discrete Logarithm Problem (ECDLP) problem, which is an equivalent to Discrete Logarithm Problem (DLP) on elliptic curves. From the practical point of view, it is essential to note, that ECDLP appears to be much harder than DLP, which implies that schemes based on ECDLP provides greater security per bit than other equivalent DLP-based schemes. In practices, that means that elliptic curve based schemes utilises much shorter keys than non-elliptic curve based counterparts, i.e., for 3072-bit RSA key, 256-bit key in ECC is considered as equivalent in terms of the security level [ENIS13].

Considering implementation point of view, every EC-based cryptographic scheme could be decomposed on three levels of operations. The first (the lowest level) consists of finite field arithmetic operations such as addition, subtraction, multiplication and inversions. The middle level performs an elliptic curve point adding and doubling, whereas the highest level is a scalar point multiplication operation. For efficient and lightweight implementations, it is essential to apply optimisation efforts on each of these levels, i.e., for the highest level one can process several bits of the scalar at the same time, whereas for the lowest level, one can tweak finite field arithmetic. Applying these efforts different computational metrics such as speed, power consumption, small code footprint etc. might be easily achieved.

There exist many elliptic curve proposals and standards, in the section below, we introduce the most relevant ones taking into consideration security and the possible lightweight implementation on the prospect RERUM platform.

2.3.2.1 NIST Curves

First NIST standard that includes Elliptic Curve Digital Signature Algorithm (ECDSA) was published in 2000 (fourth revision can be find here [FIPS11]). Among other recommendations, NIST introduced five prime fields and selected randomly one elliptic curve for each prime field. Each of curves over prime field are defined by short Weierstrass equation $y^2 = x^3+ax+b$, where $a, b \in F_p$. This makes potential implementation simpler, i.e., one can use standard and simplified Application Programming Interface (API), which is independent from underlying selected curve. The NIST standard is well-known and widely used in practical implementations and protocols, thus there exist a deep practical knowledge of many possible optimisation strategies. For example DTLS protocol with a default cipher suite that utilise public-key cryptography uses a curve named P-256 from the mentioned standard. Selecting the curve P-256 gives also opportunity to use this curve as a part of both ECDH and ECDSA schemes, which can also bring some implementation advantages, i.e. smaller code footprint.

The recent controversy over the selection of parameters aims toward suggesting that NIST standard might include a secret backdoor [BC++14], in similar fashion to a backdoor discovered in Dual Elliptic Curve Deterministic Random Bit Generator (Dual_EC_DRBG), which was also standardised by NIST. This situation leads to a limited trust in NIST standard, and influence researchers to look for new elliptic curve designs. It is worth mentioning that standard NIST curves are also prone to side-channel attacks, i.e., they lack the feature of a constant execution time, and if needed, specific countermeasures have to be applied in implementations. Considering that standard is available for a long time there exists many specific libraries, including lightweight implementation for wireless sensor networks such as TinyECC [LN08] or NanoECC [SO++08].

2.3.2.2 Curve25519 and Ed25519

Curve25519 is a state-of-the-art elliptic-curve-Diffie-Hellman (ECDH) function introduced by Bernstein [B06]. The function is suitable to perform Diffie-Hellman key exchange operations. In contrast to NIST curves over primes, Curve25519 function is based on Montgomery curve defined as $y^2 = x^3 + 486662x^2 +x$, which allows efficient x-coordinate only point operations. Similarly Ed25519 [BD++12] is a signature algorithm, defied as the twisted Edwards curve $-x^2+y^2 = 1 - (121665/121666)x^2y^2$.

Advantage of these pair of presented algorithms (apart from their security levels) lays mostly in practical applications. Montgomery and Edward curves are well-known for their efficient elliptic curve operations, which in consequence lead to high-speed implementations (in comparison with Weierstrass counterparts. Both Curve25519 and Ed25519 features shot size of secret and public keys, i.e., 32-bit for both keys, and 64-bit for signature size (in case of Ed25519). Their design prevents input-dependent branches, thus algorithms also features side-channel timing attacks by design.

Apart from [B06, BD++12], where a high-speed implementation has been confirmed on 32-bit platforms, there exists as well investigations of Curve25519 and Ed25519 applicability towards embedded designs, i.e., using 8-bits AVR microcontrollers [HS13] or using NEON coprocessor [BS12]. In the former the authors presented a special version of software library named NaCl[14] (which contains both Curve25519 and Ed25519 implementations) running on 8-bit microcontroller showing possibility for a very compact code size and presented overall performance on said platform. In the latter publication, a NEON coprocessor, which is available in many ARM processors, has been utilised to increase a performance of both Curve25519 and Ed25519 computations compared to ``software'' only implementation running on the same ARM platform.

[14] Pronounced “salt”.

2.3.2.3 Microsoft's NUMS Curves

Recently, motivated by a possible backdoor in NIST standard, a group of researchers from Microsoft have designed a new set of elliptic curves [BCLN14] and proposed them to IEFT standardisation body [NUMS14]. Although this proposition is relatively new and security claims together with performance results have not been widely studied yet, there are several clues that the proposed set of curves could act as a good replacement for the NIST standard (especially taking into consideration curves defined over primes). Contrary to Curve25519 and Ed25519, and similarly to the NIST curves, one curve from NUMS set, (to be more specific those expressed as short Weierstrass equation and over primes) could be used both in ECDH and ECDSA algorithms. In addition, some of the proposed curves are expressed as short Weierstrass form, thus is likely to hold the same API. This might be important not only from software perspective but might be much more important for hardware coprocessors, where deployments on chip modifications are very costly. In case of the same API, they are very likely be able to handle those curves without any hardware modifications. The first attempt (made by authors) for an efficient implementation shows that performance of many presented curves is better than their NIST counterparts. Similarity to Curve25519 and Ed25519, the NUMS curves feature countermeasures against side-channel timing attacks by design. It is also worth noticing that NUMS curves proposition comes with clear selection procedure of curve parameters, which as mentioned before is currently the main consideration of the standard NIST curve set.

2.3.3 Data Integrity Using Asymmetric Cryptography for 3rd-party verification (non-repudiation) without a fully trusted verifier

The following definition has been used by RERUM since deliverable D2.2 for data integrity

Data Integrity	Source: D2.2 and ISO 10181-6
The integrity protection, that RERUM requires, must allow the “detection of integrity compromises” [96], as opposed to mechanisms concerned with “prevention” [ISO_10181-6] of integrity breaches. The focus will be on protection against the following three violations: “a) unauthorized data modification; c) unauthorized data deletion; d) unauthorized data insertion;” [ISO_10181-6]	

When Data Integrity is cryptographically ensured, this means that the data’s integrity is in a verifiable state. However, a third party wanting to verify the integrity needs a cryptographic verification key. When using asymmetric key crypto, the verifier only needs a verification key, and the verifier does not get a key able to generate new integrity verification codes, e.g., new signatures need the secret key. Hence, mechanisms based on asymmetric cryptography allow **verifiable integrity by a third party**, which – in contrast to internally verifiable integrity – must not be entrusted to guard a secret. By just knowing the public verification key we can corroborate that the integrity-protected data has not been modified (or destroyed) in an unauthorised manner since the integrity protection mechanisms has been applied to the data.

Notes:

- The above notion does explicitly not exclude authorised modifications;
- What constitutes an unauthorised or authorised change must be precisely codified in the relevant integrity security policies that must be enforced to the full extend by the integrity protection mechanism. Hence, by choice of the protection mechanism the applying entity defines what constitutes an unauthorised change;
- The above notion sees data integrity in accordance with ISO 10181-6[15] as "a specific invariant on data" [ISO10181-6]. Following this, the integrity protection mechanism "detects the violation of internal consistency" [ISO10181-6]. "A datum is internally consistent if and only if all modifications of this item satisfy the relevant integrity security policies." [ISO10181-6];
- The above notion is concerned with allowing the detection of integrity violations by third parties, e.g. not limited to the party that applied the integrity protection;
- The above notion is concerned that the current state of integrity can be proven to hold also to a third party, e.g. not the sender and the intended receiver of data. This does not limit its use internally, i.e. by the party that applied the protection. Hence, third party verifiable integrity includes the notion of internally verifiable integrity.

2.3.4 Origin Authentication Using Asymmetric Cryptography for 3rd-party verification (non-repudiation) without a fully trusted verifier

RERUM sees as an important requirement to offer cryptographically strong authentication of message origin, which requires entity authentication.

Entity Authentication, Verifier, Claimant	Source: [MOV01]
Entity authentication is the process whereby one party is assured (through acquisition of corroborative evidence) of the identity of a second party involved in a protocol, and that the second has actually participated (i.e., is active at, or immediately prior to, the time the evidence is acquired). By Verifier we denote the party that is, after a successful run of the protocol, assured of the identity of a second party involved in an entity authentication protocol. By Claimant we denote the party that is generating a proof of its identity in an entity authentication protocol.	

Following the definition of [MOV01], this is the process whereby one party is assured of the identity of a second party involved in a protocol. RERUM will not require the generation of evidence used for non-repudiation as included in the above textbook definition. While RERUM does not restrict using authentication mechanisms that are strong enough to be used to generate evidence of the authentication that can be proven to hold also to a third party, it does not mandate this.

[15] ISO 10181-6 is about integrity frameworks

It is also in line with the RFC 4949[16] that actually quotes the ISO 7498-2[17]:

"the corroboration that a peer entity in an association is the one claimed." [ISO7498-2, RFC 4949].

The goal of entity authentication is crucial because only if this can be established, the recipient of a message can identify from which entity in a communication the message originated. Or in other words the entity that is able to authenticate the other entity can detect if the sending entity is not forged.

"This service is used at the establishment of, or at times during, an association to confirm the identity of one entity to another, thus protecting against a masquerade by the first entity." [RFC 4949]

The two parties involved in an entity authentication dialog can be distinguished. RERUM calls them verifier and claimant.

Note, for a short term or one-time authentication token, it might be required to involve in the issuance of such a token, some long term token. Both tokens must be safe guarded against theft by both parties.

Note further, that authentication protocols usually only allow to corroborate the other parties claimed identity at the time the protocol is run. RFC 4949 states this limitation by saying that "... the corroboration provided by the service is valid only at the current time that the service is provided." [RFC4949] Hence to extend the time of check to the time of use, care must be taken to not become vulnerable to so called Time-of-check Time-of-use (TOCTOU) Race Condition attacks[18].

2.3.5 Notation and Background on Classic Public Key Based Digital Signatures

2.3.5.1 General Algorithmic Description of Digital Signature Schemes (DSS)

For a message we assume $m \in \{0,1\}^*$. With the symbol $\perp \notin \{0, 1\}^*$ we denote an error or an exception. The security parameter is denoted as λ.

A Digital Signature Scheme for RERUM consists of at least three efficient (with polynomial runtime) algorithms DSS := (KGensig, Sign, Verify):

Key Generation. There is a key generation algorithm that generates a key pair for the signer. The pair of keys contains the private signature generation key sk_{sig} and the public signature verification key pk_{sig}.

$(pk_{sig}, sk_{sig}) \leftarrow KGensig(1^{\lambda})$

Signing. The Sign algorithm takes $m \in \{0,1\}^*$, the signer's secret signature generation key sk_{sig}. It outputs the message m and a signature σ (or $\perp$, indicating an error):
$(m, \sigma) \leftarrow Sign(1^{\lambda}, m, sk_{sig})$

[16] RFC4949 provides definitions, abbreviations, and explanations of terminology for information system security. It offers recommendations to improve the comprehensibility of written material that is generated in the Internet Standards Process (RFC 2026).

[17] ISO 7498-2 provides a general description of security services and related mechanisms, which can be ensured by the Reference Model, and of the positions within the Reference Model where the services and mechanisms may be provided.

[18] http://cwe.mitre.org/data/definitions/367.html

Verification. The Verify algorithm outputs a decision d ∈ {true, false} verifying the validity of a signature σ for a message m ∈ {0, 1}∗ with respect to the public signature verification key pk_{sig}:

d ← Verify(1^{λ}, m, σ, pk_{sig})

We require the usual correctness properties to hold. In particular, all genuinely signed messages are accepted by verify. For a formal definition of correctness, refer to [BF++09] and [BPS12].

2.3.5.2 Choice Between Approved Algorithms: RSA-PSS, ECDSA, SHA256, SHA512

The research projects NESSIE[19] and CRYPTREC[20] gave recommendations for digital signature algorithms to use. From their list the overlapping choice was to use either:

- ECDSA by Certicom Corp
- RSA-PSS by RSA Laboratories

The former uses elliptic curves, while the latter is based on RSA. As current research shows elliptic curve cryptography (ECC) is far better suited for constrained devices [Certicom], RERUM will base its asymmetric key cryptography when possible on ECC. For a mathematical introduction to elliptic curves see [M06].

Both algorithms work in two steps when generating the signature of a message m, they first hash the message and then sign the digest, that why RERUM calls these Hash and Sign.

In RERUM we will use H to denote a cryptographically secure hashing function. So with H being a secure hashing function and m being the message the two steps in the signature generation are denoted as follows: (m, σ) ← Sign(1^{λ}, H(m), sk_{sig}).

Among the widespread implementations of hash functions that are found to be secure according to NESSIE and CRYPTREC, are:

- SHA-256 defined in US FIPS 180-2;
- SHA-512 defined in US FIPS 180-2 and
- WHIRPOOL defined in ISO/IEC 10118-3.

Those three, all have existing implementations, and those will be re-used preferably in RERUM for on device implementations.

2.3.5.3 Details on the ECC based Signature Algorithm: ECDSA

RERUM plans to use ECDSA algorithms to generate signatures. UNI PASSAU has built a research prototype on Zolertia Z1 devices where two Z1 are exchanging UDP messages that are signed. Details of this can be found in this deliverable in Section 3.2 on the On-Device-Signatures. To be able to use signatures we need a minimum of two interfaces: one for singing and one for verification. RERUM foresees the headers of that interface to look as follows in C language.

[19] https://www.cosic.esat.kuleuven.be/nessie/

[20] http://www.cryptrec.go.jp/english/images/cryptrec_01en.pdf

Signing:

```
/*
 * \param shasum        Hash of the message to sign using SHA256 or others
 * \SHA_DIGEST_LENGTH   Length of the digest output that shasum has,
                        depends on algorithm used
 * \param r             Parameter of the curve used
 * \param s             This will store the Signature generated
                        over the shasum
 * \param pr_key        private key used to sign shasum
 */

void ecdsa_sign(uint8_t shasum[SHA_DIGEST_LENGTH], NN_DIGIT *r, NN_DIGIT
*s, NN_DIGIT *pr_key);
```

Verification:

```
/*
 * \param shasum             Hash of the message to verify using
                             SHA256 or others
 * \SHA_DIGEST_LENGTH        Length of the digest output that shasum has,
                             depends on hash algorithm used
 * \param r                  Parameter of the curve used
 * \param s                  Signature of the message
 * \param pb_key             public key used to verify the Signature
                             over shasum
 * \return                   1 if the signature is verified under pb_key
 */

uint8_t ecdsa_verify(uint8_t shasum[SHA_DIGEST_LENGTH], NN_DIGIT *r,
NN_DIGIT *s, point_t *pb_key);
```

2.3.6 Notation and Background on Group Signatures

Group signatures have been recognized as one of the major privacy enhancing technologies to reach data minimization and privacy by design [MF12]. Group signatures ensure unforgeability for messages, authenticate signers and provide k-anonymity[21] for the members of a signing group. Since the original proposal of group signatures by Chaum and van Heyst [CV91] (the scheme is described in the following), there have been many extensions and adaptions to group signatures. We refer to the simplified description of [MF12] to explain how group signatures work.

[21] A group member is said to be k-anonymous if his signatures cannot be distinguished from at least k-1 other group members, which could have generated the same signatures as well.

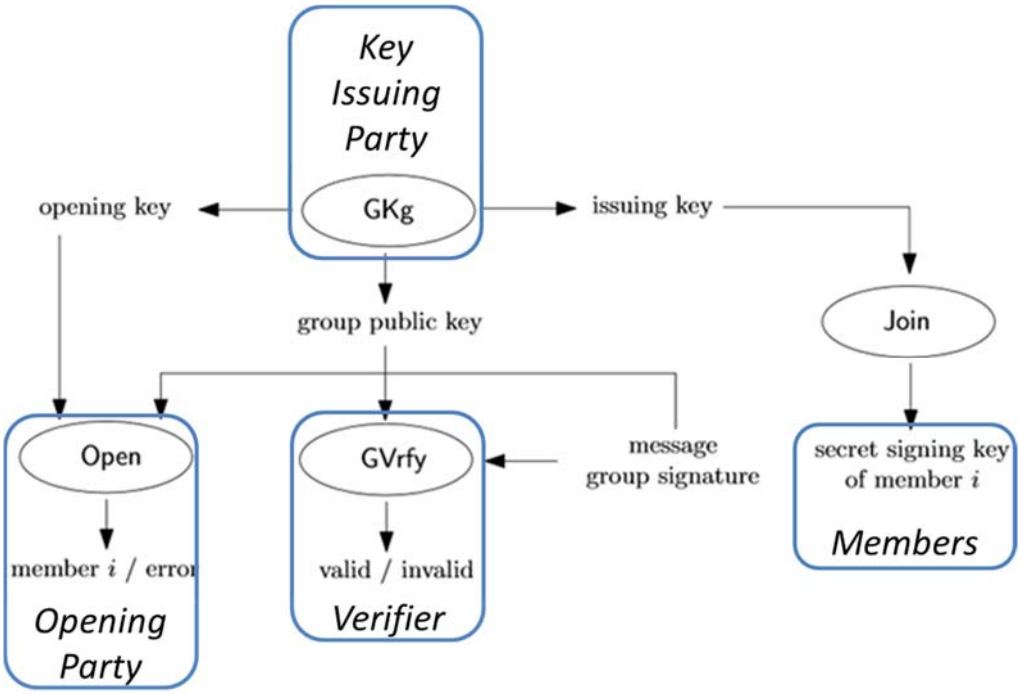


Figure 6: Group Signature Scheme with linking functionality (adapted from [MF12])

Figure 6 displays the overall scheme of group signatures. Several parties can be identified in a group signature scheme:

The key issuing party does the key management for all participants. The issuing party generates, distributes and revokes keys. It is not involved in the verification or linking/opening procedures of the scheme.

The issuing party is able to generate all keys with a function GKg( ), which takes as input the initial size of a group:

`GKg(number_of_initial_members_i)`

The function generates as an output three keys, the group's public key, the member keys and the opening key:

`GPubKey, Memberkey`$_i$`, GOpeningKey`

The issuing party then distributes the keys between the different parties:

`Issuing Party -> Member 1 to Member i: Memberkey`$_{1\ldots i}$

`Issuing Party -> Verifier(s) or public: GPubKey`

`Issuing Party -> Opening Party: GOpeningKey`

Note: The issuing party has to remember which member received which key to assist the opening party in relinking a signature to a member. This is done in a member-secret distribution table.

Additionally, the issuing party can add several members to the group with the function AddMember():

`AddMember(GPubKey, number_of_new_members_m)` creates additional member keys `Memberkey(i+1)…(i+m)`

The issuing party can also revoke keys with the function:

`RevokeKey(GPubKey, Memberkey`$_i$`)`

The results of revocation are different depending on the used scheme. Several schemes allow the revocation of the group public key to successfully revoke a member key. Others need to update the group public key and all member keys.

The group members join a group by obtaining secret signing keys. They are able to create digital group signatures in the name of the whole group. Neither a verifier nor the members themselves are able to recognize the source of a signed message.

GSign(Memberkey$_i$, message) generates a signature Gsignature$_{message}$

The signature does not resemble the original message key and is verifiable with the group public key.

The verifier(s) has access to the group's public key. He is able to verify that the group is the authentic source of a message, but he is not able to pinpoint the group member that originally signed the message.

The function GVrfy(GPubKey, Gsignature$_{message}$, message) gives the output True/False in case of a valid or invalid verification of the message.

The opening party has a unique key, which allows it to relink a signature to one member key of the group. Relinking means recreating the link between a given signature, a message and the original signer. This process, in contrast to the revoking process of a key, does neither change member-keys nor the group public key. That means, there are no further consequences for neither the verifiers nor the group members. This property is needed, in case a malicious member is culpable of fraud or has to provide compensation of damages for his misbehaviour. The opening party is supposed to have a high trust level, and only relink signatures in justifiable circumstances.

The opening party uses the function Open() with following arguments:

Open(GOpeningKey, Gsignature$_{message}$, message, key_distribution_table)

The function then outputs a specific member key Memberkey$_i$

Note: The member key does not identify a user by itself. The opening party needs to know from the issuing party, which user was given the key that was generated as output from the Open()-function.

2.3.7 Notation and Background on Identity Based Signatures

When using Identity based cryptosystem the public key needed to encrypt data for a recipient or to verify the signature consists only of public identification information, e.g. a unique name like the hardware address or the IP in the current network, and the public information of the trusted authority. The public parameters are denoted as IBE_params and the trusted authority is denoted T.

T holds a IBE_masterkey that corresponds to the IBE_params, such that they could be seen as the asymmetric keypair, and indeed IB-schemes are classified as asymmetric [MOV01]. For participating entities T generates a private key that computed from two main inputs: entity's identity information and IBE_masterkey.

Note, only T can create valid private keys corresponding to given identification information under the IBE_params.

T acts as a key generator and is a trusted third party, when it comes to generating keys, hence, T could impersonate (by generating itself keys) any entity and it needs to safeguard IBE_masterkey. As the IBE_masterkey that is held by the trusted third party T can be used to generate the secret keys for any arbitrary ID string.

Very common instantiation is the Feige-Fiat-Shamir identification scheme [FFS88].

As the authors of [MOV01] have already mentioned, the main benefit for this schemes lies in interactions where "another party's public key is needed at the outset" [MOV01]. This means that these schemes are theoretically quite efficient when used in key agreement and public-key encryption but not so much when used for signature and entity authentication. Recent state of the art shows that it is important to choose exactly the right scheme and that schemes with certain flexibilities, e.g. multi-proxy signatures, are not secure in strong attacker models, e.g. secure against proxy key exposure [ASS14].

The state of the art is rapidly evolving in the area of applying IB-identification to Wireless sensor networks. Especially approaches like [MA12], [YRW10], or [QZ++14] published recently, sound interesting. RERUM continues to monitor activity in this field.

2.3.8 Notation and Background on Implicit Certificates

Implicit certificates are another special form of an asymmetric cryptographic mechanism. One published instantiation is based on the Elliptic Curve Qu-Vanstone (ECQV). When comparing the verification steps for a signature based on an implicit certificate with the steps when using a conventional certificate the most notable difference is that in implicit certificates there is no separate verification of the certificate chain. If you look at a classical certificate chain as depicted below, than you would always also have to check if the next trusted hierarchy, e.g. finally the root-of-trust, has issued it. Using implicit certificates the signature verification process gets the following input: the trusted public parameters of the CA, the signature, and the signed data. It outputs valid if and only if the data is as it was supposed to be (integrity preserved) and if the secret signature generation key was indeed endorsed by the CA to which the public parameters correspond. In order to do this the implicit certificate also has the subject's ID public key but they are super imposed into a string the size of the public key including the implicit endorsement of the CA. This results in savings of some additional operations and a decrease in the size of the certificate, the stated example says that on a curve with 160 bit sized points, an implicit certificates has the size of 160 bits [Certicom], [PV00], [BGV01].

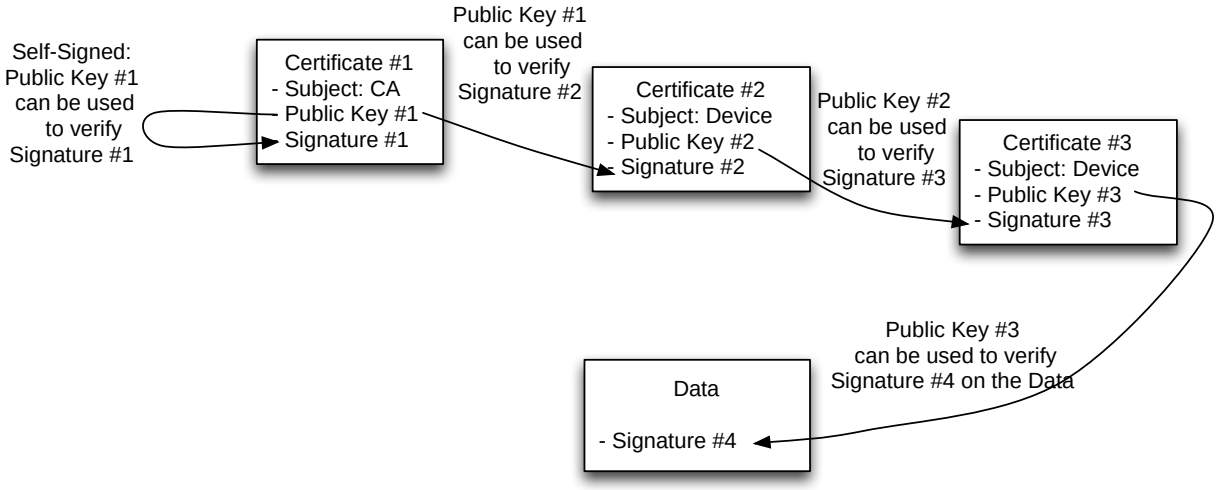


Figure 7: Certificate chain

Figure 7 shows an example of a Certificate Chain, with Certificate#1 being trusted it is possible to derive trust into the Public Key #3 and by that in Signature#4 and the data that this signature protected.

The following notation is adapted from the original authors as follows:

The entity, called Bob has an implicit certificate for himself, denoted `IMPL_CERT`, this shall contain unique information, denoted I in the original paper, identifying Bob, e.g. ID or address and some

certificate ID. This IMPL_CERT contains the secret IMPL_SK but also allows the reconstruction of the public key IMPL_PK (originally B) when combined with the trusted public parameters of the CA, denoted (originally C) IMPL_PARAMS.

Setup: Initially the system parameters / keys of the CA are generated; there are public IMPL_PARAMS and secret IMPL_CA_SEC. The secret key can then be used to issue IMPL_CERT for strings of information I.

Sign: When signing, Bob will use his IMPL_SK to create a signature σ for a message $m \in \{0, 1\}^*$

$(m, \sigma) \leftarrow \text{Sign}(1^\lambda, m, \text{IMPL\_SK})$

Verify: For verification the verifier needs to have IMPL_PK, but only needs to trust IMPL_PARAMS. The Verify algorithm outputs a decision $d \in \{\text{true, false}\}$ verifying the validity of a signature σ for a message $m \in \{0, 1\}^*$ with respect to the public key of the signer IMPL_PK and the trusted public key of the Certificate Issuing TTP IMPL_PARAMS:

$d \leftarrow \text{Verify}(1^\lambda, m, \sigma, \text{IMPL\_PK}, \text{IMPL\_PARAMS})$

2.3.9 New Harmonized Notation[22] and Background on Malleable Signature Schemes

Apart from standard digital signature schemes based on the hash-then-sign paradigm and using cryptographically secure hashes like SHA256 or RIPE-MD, there are so called malleable signatures.

This section of the deliverable will briefly discuss the background behind the notion of malleable signatures, it will state the mathematical notation used in RERUM, and also provide an executive summary of the results published as a result of RERUM's analysis at ESSoS'14 [MPPS14].

At the time of preparing this section of the deliverable there was no work that considered identifying the mathematical relations between the notions of two important strands of malleable signatures: redactable signature schemes (RSS) and sanitizable signature schemes (SSS). For a deeper and thorough understanding, and in order to identify which schemes to implement on RERUM Devices first, so as part of the work in T3.2 (aiding also T3.3) RERUM carried out this analysis and an executive summary is provided, all mathematical proofs can be found in [MPPS14].

Malleable signature schemes (MSS) allow generating a signature over data that allows a specified third-party to modify signed data and re-compute a potentially different signature, which is again valid for the modified data; the re-computation of the signature can be done without the signer's signature generation key.

The signature on the modified data is valid under the signer's public verification key if and only if the signer-specified rules for subsequent modifications are adhered to.

As such, malleable signatures schemes (MSS) offer:

1. integrity protection for the message, protecting against subsequent malicious or random, but unauthorised modifications and
2. authentication of origin of the message, as the party that applied the protection on a message, by signing it, can be identified by the corresponding verification key.

RERUM thoroughly analysed the current state of MSS to understand what algorithms to choose for RERUM's idea to apply malleable signatures already on devices, identified two different currently studied cryptographic constructions:

[22] These research results from RERUM have been published in [MPPS14].

1. redactable signature schemes (RSS) and
2. sanitizable signature schemes (SSS).

These research results from RERUM have been published in [MPPS14].

2.3.9.1 Concept of Redactable Signature Schemes (RSS)

Redactable Signatures (RSS) have been introduced in 2002 in two independent works under two different names:

(1) Steinfeld, Bull and Zheng named it “content extraction signature” [SBZ02] and
(2) Johnson, Molnar, Song and Wagner named it “homomorphic signature” [JMSW02].

The original description from Steinfeld et al. is easy to understand, so we cite it here:

> “A CES [Content Extraction Signature] allows the owner, Bob, of a document signed by Alice, to produce an ’extracted signature’ on selected extracted portions of the original document, which can be verified to originate from Alice by any third party Cathy, while hiding the unextracted (removed) document portions.” [SBZ02]

RERUM follows recent work calling them redactable signature schemes (RSS), e.g. [WH++12], [BB++10], or [SR10] as the schemes allow removing parts from signed data and allowing to re-compute the signature, such that this removal will not be invalidating the new signature.

2.3.9.2 Concept of Sanitizable Signature Schemes (SSS)

Sanitizable signature schemes (SSS) have been introduced by Ateniese et al. [ACMT05] at ESORICS in 2005. They differ from RSS mainly in the fact that the signer can control who – that entity is denoted as sanitizer – is able to modify the document by the use of cryptographic keys. The party authorised to do the change has the correct cryptographic keys and is then able to change the parts for which it holds authorisation arbitrarily. The initial description of Ateniese et al. from [ACMT05] captures this and so it is re-stated here:

> “We define a sanitizable signature scheme as a secure digital signature scheme that allows a semi-trusted censor to modify certain designated portions of the message and produce a valid signature of the resulting (legitimately modified) message with no interaction with the original signer.” [ACMT05]

This definition also states a very important point of MSS: both RSS and SSS do not require the party modifying parts, that it has been authorised to, to engage in any interaction with the signer, neither before, nor during, nor after the redaction. The same holds true for the relation and interaction between the party verifying the signature of modified content and the party that modified it, there is no interaction or key exchange needed between those two parties.

This independence of the execution of a once authorised modification makes MSS such a suitable tool for integrity in environments where entities benefit from being loosely coupled on the networking layer, like in the IoT.

2.3.9.3 State of the Art in Redactable and Sanitizable Signatures Schemes

SSS have been introduced by Ateniese et al. [ACMT05] at ESORICS'05. Brzuska et al. formalized the most essential security properties [BF++09][BF++09b]. These have later been extended for the properties of unlinkability [BFAS10] [BPS14] and (block/groupwise) non-interactive public accountability [BPS12] [MPPS13]. Moreover, several extensions and modifications like limiting-to-values [CJ10] [KL06] [PSP11], trapdoor SSS [CLM08] and multi-sanitizer environments [CJL12] have been considered.

RSS have since its introduction [SBZ02 [JMSW02] in 2002 been subject to much research and got extended to tree-structured data [BB++10] [KB08] and to arbitrary graphs [KB10]. Samelin et al. introduced the concept of redactable structure in [SP++12b]. The standard security properties of RSS have been formalized in [BB++10] [CLX09] [SP++12] Ahn et al. introduced the notion of context-hiding RSS [AB++11]. Even stronger privacy notions have recently been introduced in [ALP12] [ALP13]. However, the scheme by Ahn et al. only achieves the less common notion of *selective* unforgeability [AB++11]. Moreover, [AB++11] [ALP12] [ALP13] all not offer the flexibility needed for RERUM's IoT data, as they are limited to quoting, i.e., redactions are only possible at the beginning, or end resp., of a list. There exists many additional works on RSSs. However, note that RERUM requires for reasons of privacy as described in detail in D3.2 that usable schemes fulfil a notion that RERUM would call *strongly private* RSS defined formally by Brzuska et al. in [BB++10]. If a verifier can make statements about the original message, that contradicts the intention of an RSS, and hence the privacy definition presented in [BB++10] forbids this. Hence, for RERUM schemes like [HH++08] [IIKO11] [IKPS09] [LLP12] [MI++05] are not suitable for RERUM as they are not considered strongly private under Brzuska et al.'s privacy notion. Most of these schemes achieve a weaker notion that RERUM will call weak privacy.

Combinations of both approaches, i.e. RSS and SSS, appeared in [HH++08], [IIKO11], [IKPS09]. However, their schemes do not preserve privacy as stated in [SP++12].

2.3.9.4 Executive Summary of Analysis Results on the Relation between RSS and SSS (published in ESSoS'14 [MPPS14])

At the time of preparing this section of the deliverable, and to the best of the author's knowledge, there was no work that considered identifying the mathematical relations between the notions of redactable signature schemes (RSS) and sanitizable signature schemes (SSS). For a deeper and thorough understanding, and in order to identify which schemes to implement on RERUM Devices first, so as part of the work in T3.2 (aiding also T3.3) RERUM carried out this analysis.

The following results were obtained:

- an RSS is not *trivially* a "special case" of SSS as mentioned in [BF++09], [YSLM08] and [YPL10]
- an unforgeable SSS can emulate a standard signature by disallowing any modifications by any sanitizer [YPL10]
- a RSS for a message of *n* blocks, fulfilling strong privacy according to [BB++10], can trivially, but runtime inefficiently, be constructed by deploying $O(n^2)$ standard digital signatures [SP++12] [BB++10]. As SSS can emulate standard signatures, by disallowing modifications, $O(n^2)$ SSS are cryptographically sufficient to construct one RSS.

Thus, from a theoretical point of view, SSSs directly imply the existence of RSSs. However, from a practical point of view, such constructions are rather inefficient. Especially as RSSs can be constructed in $O(n)$ for computation and storage [SP++12].

The analysis results in the statement that RSSs are less expressive than SSSs. In other words, no unforgeable RSS can be transformed into an SSS. For the opposite direction a black-box transformation of a single SSS, with tightened security, into an RSS is possible.

The detailed results containing the mathematical proofs of this analysis have been published by Springer International Publishing in the In Proc. of the 6th International Symposium on Engineering Secure Software and Systems (ESSoS 2014), see [MPPS14].

2.3.9.5 Notation of an SSS (published in ESSoS'14 [MPPS14])

The used notation is adapted from in [BF++09].

For a message m = (m[1],...,m[l]), we call m[i] ∈ {0,1}∗ a block, where "," denotes a uniquely reversible concatenation of blocks or strings. The symbol ⊥ ∉ {0, 1}∗ denotes an error or an exception. For a visible redaction, we use the symbol □∉ {0, 1}∗, □ is not equal to ⊥.

A SSS consists of at least seven efficient (with polynomial runtime) algorithms SSS := ($KGen_{sig}$,$KGen_{san}$,Sign,Sanit,Verify, Proof, Judge):

Key Generation. There are two key generation algorithms, one for the signer and one for the sanitizer. Both create a pair of keys, a private key and the public key, using the security parameter λ:

(pk_{sig}, sk_{sig}) ← $KGen_{sig}(1^{\lambda})$, (pk_{san}, sk_{san}) ← $KGen_{san}(1^{\lambda})$

Signing. The Sign algorithm takes m = (m[1],...,m[l]), m[i] ∈ {0,1}∗, the signer's secret key sk_{sig}, the sanitizer's public key pk_{san}, as well as a description adm of the admissibly modifiable blocks, where adm contains the number l of blocks in m, as well the indices of the modifiable blocks. It outputs the message m and a signature σ (or ⊥, indicating an error): (m, σ) ← Sign(1^{λ}, m, sk_{sig}, pk_{san}, adm)

Sanitizing. Algorithm Sanit takes a message m = (m[1], ...,m[l]), m[i] ∈ {0,1}∗, a signature σ, the public key pk_{sig} of the signer and the secret key sk_{san} of the sanitizer. It modifies the message m according to the modification instruction mod, which contains pairs (i, m[i]') for those blocks that shall be modified. Sanit calculates a new signature σ' for the modified message m' ← mod(m). Then Sanit outputs m' and σ' (or ⊥, indicating an error):

(m', σ') ← Sanit(1^{λ}, m, mod, σ, pk_{sig}, sk_{san})

Verification. The Verify algorithm outputs a decision d ∈ {true, false} verifying the validity of a signature σ for a message m = (m[1], ...,m[l]), m[i] ∈ {0, 1}∗ with respect to the public keys:

d ← Verify(1^{λ}, m, σ, pk_{sig}, pk_{san})

Proof. The Proof algorithm takes as input the security parameter, the secret signing key sk_{sig}, a message m = (m[1], ..., m[l]), m[i] ∈ {0, 1}∗ and a signature σ as well a set of (polynomially many) additional message-signature pairs {(m_i, σ_i) | i ∈ N} and the public key pk_{san}. It outputs a string π ∈{0, 1}∗ (or ⊥, indicating an error):

π ← Proof(1^{λ}, sk_{sig}, m, σ, {(m_i, σ_i) | i ∈ N}, pk_{san})

Judge. Algorithm Judge takes as input the security parameter, a message m = (m[1],...,m[l]), m[i] ∈ {0,1}∗ and a valid signature σ, the public keys of the parties and a proof π. It outputs a decision d ∈ {Sig, San, ⊥} indicating whether the message-signature pair has been created by the signer or the sanitizer (or ⊥, indicating an error): d ← Judge(1^{λ}, m, σ, pk_{sig}, pk_{san}, π)

To have an algorithm actually able to derive the accountable party for a specific block m[i], Brzuska et al. introduced the additional algorithm Detect [BFA10]. The algorithm Detect is not part of the original SSS description by Ateniese et al., since it is not required for the purpose of a SSS [ACMT05] [BF++09]. (See Def. 6 of the original paper [MPPS14]).

Detect. On input of the security parameter λ, a message- signature pair (m, σ), the corresponding public keys pk_{sig} and pk_{san}, and a block index $1 \le i \le l$, Detect outputs the accountable party (San or Sig) for block i (or ⊥, indicating an error):

$d \leftarrow Detect(1^{\lambda}, m, \sigma, pk_{sig}, pk_{san}, i)$, $d \in \{San, Sig, \perp\}$

We require the usual correctness properties to hold. In particular, all genuinely signed or sanitized messages are accepted, while every genuinely created proof π by the signer leads the judge to decide in favour of the signer. For a formal definition of correctness, refer to [BF++09] [BFA10]. It is also required by every SSS that adm is always correctly recoverable from any valid message-signature pair (m, σ). This accounts for the work done in [GQZ11].

An SSS is secure if it is unforgeable, immutable and strongly private.

RERUM will define software interfaces for SSS in T3.3 delivered in D3.2. such that SSS can be used in the integrity verification component.

2.3.9.6 Notation of an RSS (published in ESSoS'14 [MPPS14])

The following notation is derived from [SP++12b].

For a message m = (m[1], ...,m[l]), we call $m[i] \in \{0,1\}^*$ a block, where "," denotes a uniquely reversible concatenation of blocks or strings. The symbol $\perp \notin \{0, 1\}^*$ denotes an error or an exception. For a visible redaction, we use the symbol $\square \notin \{0, 1\}^*$. Note, □ is not equal to ⊥.

An RSS consists of four efficient (with polynomial runtime) algorithms RSS := (KeyGen, Sign, Verify, Redact):

KeyGen. The algorithm KeyGen outputs the public key pk and private key sk of the signer, where λ denotes the security parameter:

$(pk, sk) \leftarrow KeyGen(1^{\lambda})$

Sign. The algorithm Sign gets as input the secret key sk and the message m = (m[1], ..., m[l]), $m[i] \in \{0, 1\}^*$: $(m, \sigma) \leftarrow Sign(1^{\lambda}, sk, m)$

Verify. The algorithm Verify outputs a decision $d \in \{true,false\}$, indicating the validity of the signature σ, w.r.t. pk, protecting m = (m[1],...,m[l]), $m[i] \in \{0, 1\}^*$: $d \leftarrow Verify(1^{\lambda}, pk, m, \sigma)$

Redact. The algorithm Redact takes as input the message m = (m[1], ..., m[l]), $m[i] \in \{0,1\}*$, the public key pk of the signer, a valid signature σ and a list of indices mod of blocks to be redacted. It returns a modified message $m' \leftarrow mod(m)$ (or ⊥, indicating an error):

$(m', \sigma') \leftarrow Redact(1^{\lambda}, pk, m, \sigma, mod)$

We denote the transitive closure of m as $span_{|=}(m)$. This set contains all messages derivable from m w.r.t. Redact.

As for SSSs, the correctness properties for RSSs are required to hold as well. Thus, every genuinely signed or redacted message must verify. Refer to [BB++10] for a formal definition of correctness.

RERUM will define software interfaces for RSSs in T3.3 delivered in the future deliverable D3.2.

2.3.9.7 Executive Summary of the Security Required to be Achieved by RSS and SSS for Use in RERUM

RSS and SSS are called secure for the use in RERUM if they achieve the following four fundamental security properties:

- **Unforgeability.** No one should be able to compute a valid signature on a message not previously issued without having access to any private keys [BB++10]. This is analogous to the unforgeability requirement for standard signature schemes [GMR88], except that it excludes valid redactions from the set of forgeries for RSSs, while for SSSs no alterations are allowed.
- **Immutability**. The idea behind immutability is that an adversary generating the sanitizer key must only be able to sanitize admissible blocks. Hence, immutability is the unforgeability requirement for the sanitizer.
- **Privacy**. No one should be able to gain any knowledge about sanitized parts without having access to them [BF++09]. This is similar to the standard indistinguishability notion for encryption schemes. The basic idea is that the attacker provides two, potentially different, messages and instructions on how to change them to another party, whose operations it can not observe, called "oracle". The Oracle either signs and sanitizes the first message (m_0) or the signs and sanitizes the second (m_1), while the oracle checks that the resulting sanitized message must be the same for each message and change instructions. One sanitized message with a valid signature is then given from the oracle to the adversary. The adversary must not be able to decide which input message was used when given the signed but sanitized output.
- **Weak Blockwise Non-Interactive Public Accountability**. The basic idea behind breaking accountability is that an adversary, i.e., in the role of a sanitizer, has to be able to make the Detect algorithm accuse the signer, even if it was not the accused party that signed the specific block. Moreover, in this weak accountability notion foreseen as suitable for use of MSS in RERUM, the signer is not considered adversarial, contrary to Brzuska et al. [BPS12]. An example for a weakly blockwise non-interactive publicly accountable SSS is the scheme introduced by Brzuska et al. [BPS12].

Next follows the detailed cryptographic security definitions containing the mathematical games and the proofs from the published paper [MPPS14].

2.3.9.8 Cryptographic Security Models and Properties of RSS and SSS

This section contains the required security properties and models. They are derived from [BF++09] [GQZ11] [SP++12b], but have been significantly altered. The requirement that adm[23] is always correctly reconstructible is captured within the unforgeability and immutability definitions. Note, following [BF++09] [BPS12] [BPS14], an SSS must at least be unforgeable, immutable, accountable and private to be meaningful. Hence, we assume that all used SSS fulfill these four fundamental security requirements; if these requirements are not met, the construction is not considered an SSS and the results of this research are not directly applicable. On the other hand, an RSS must be unforgeable and (weakly) private to be meaningful [BB++10]. In the following game based security definitions are used, so for every property there is a so-called "experiment" which is a game that the attacker must win with non-negligible probability.

[23] adm holds the information about admissibly modifiable blocks, in detail think that adm contains the number l of blocks in m, as well the indices of the modifiable blocks.

Definition of **RSS Unforgeability:**

An RSS is said to be unforgeable, if for any efficient (PPT) adversary *A* the probability that the game depicted in Figure 8 below returns 1, is negligible (as a function of λ).

$$
\begin{array}{l}
\textbf{Experiment } \mathsf{Unforgeability}_{\mathcal{RSS},\mathcal{A}}(\lambda) \\
\quad (pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda) \\
\quad (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, sk, \cdot)}(pk) \\
\qquad \text{let } i = 1, \ldots, q \text{ denote the queries to Sign} \\
\quad \text{return 1, if} \\
\qquad \mathsf{Verify}(1^\lambda, pk, m^*, \sigma^*) = 1 \text{ and} \\
\qquad \text{for all } i = 1, \ldots, q : m^* \notin \mathrm{span}_{\models}(m_i)
\end{array}
$$

Figure 8: Unforgeability Experiment for RSS

Definition of **SSS Unforgeability:**

An SSS is said to be unforgeable, if for any efficient (PPT) adversary *A* the probability that the game depicted in the Figure 9 below returns 11 is negligible (as a function of λ).

$$
\begin{array}{l}
\textbf{Experiment } \mathsf{Unforgeability}_{\mathcal{SSS},\mathcal{A}}(\lambda) \\
\quad (pk_{\mathrm{sig}}, sk_{\mathrm{sig}}) \leftarrow \mathsf{KGen}_{\mathrm{sig}}(1^\lambda) \\
\quad (pk_{\mathrm{san}}, sk_{\mathrm{san}}) \leftarrow \mathsf{KGen}_{\mathrm{san}}(1^\lambda) \\
\quad (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, \cdot, sk_{\mathrm{sig}}, \cdots), \mathsf{Proof}(1^\lambda, \cdot, sk_{\mathrm{sig}}, \cdots), \mathsf{Sanit}(1^\lambda, \cdots, sk_{\mathrm{san}})}(pk_{\mathrm{sig}}, pk_{\mathrm{san}}) \\
\qquad \text{let } (m_i, \mathrm{ADM}_i, pk_{\mathrm{san},i}) \text{ and } \sigma_i \text{ for } i = 1, 2, \ldots q \\
\qquad \text{denote the queries/answers to/by the oracle Sign,} \\
\qquad \text{let } (m_j, \mathrm{MOD}_j, \sigma_j, pk_{\mathrm{sig},j}) \text{ and } (m'_j, \sigma'_j) \text{ for } j = q+1, \ldots, r \\
\qquad \text{denote the queries/answers to/by the oracle Sanit.} \\
\quad \text{return 1, if} \\
\qquad \mathsf{Verify}(1^\lambda, m^*, \sigma^*, pk_{\mathrm{sig}}, pk_{\mathrm{san}}) = \mathtt{true} \text{ and} \\
\qquad \text{for all } q = 1, \ldots q : (pk_{\mathrm{san}}, m^*, \mathrm{ADM}^*) \neq (pk_{\mathrm{san},i}, m_i, \mathrm{ADM}_i) \text{ and} \\
\qquad \text{for all } j = q+1, \ldots, r : (pk_{\mathrm{sig}}, m^*, \mathrm{ADM}^*) \neq (pk_{\mathrm{sig},j}, m_i, \mathrm{ADM}_i)
\end{array}
$$

Figure 9: Unforgeability Experiment for SSS

Next is the definition of Weak Blockwise Non-Interactive Public Accountability. The reasons for this adversary model are given after the introduction of all required security properties. Note, pk_{san} is the sanitizer's public key and it is fixed for the oracles. For SSS, we also have sanitization and proof oracles [BF++09].

Definition of **SSS Weak Blockwise Non-Interactive Public Accountability**:

A sanitizable signature scheme SSS is *weakly non-interactive publicly accountable*,

if Proof=⊥, and if for any efficient algorithm *A* the probability that the experiment given in the Figure 10 below returns 1 is negligible (as a function of λλ).

$$
\begin{array}{l}
\textbf{Experiment } \mathsf{WBlockPubAcc}_{\mathcal{SSS},\mathcal{A}}(\lambda) \\
\quad (pk_{\mathrm{sig}}, sk_{\mathrm{sig}}) \leftarrow \mathsf{KGen}_{\mathrm{sig}}(1^\lambda) \\
\quad (pk_{\mathrm{san}}, sk_{\mathrm{san}}) \leftarrow \mathsf{KGen}_{\mathrm{san}}(1^\lambda) \\
\quad (m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, \cdot, sk_{\mathrm{sig}}, pk_{\mathrm{san}}, \cdot), \mathsf{Proof}(1^\lambda, \cdot, sk_{\mathrm{sig}}, \cdots, pk_{\mathrm{san}})}(pk_{\mathrm{san}}, sk_{\mathrm{san}}, pk_{\mathrm{sig}}) \\
\qquad \text{let } (m_i, \mathrm{ADM}_i) \text{ and } (m_i, \sigma_i) \text{ for } i = 1, \ldots, k \text{ be queries/answers to/by Sign} \\
\qquad \text{return 1, if} \\
\qquad\quad \mathsf{Verify}(1^\lambda, m^*, \sigma^*, pk_{\mathrm{sig}}, pk_{\mathrm{san}}) = \mathtt{true} \text{ and} \\
\qquad\quad \exists q, \text{ s.t. } \mathsf{Detect}(1^\lambda, m^*, \sigma^*, pk_{\mathrm{sig}}, pk_{\mathrm{san}}, q) = \mathtt{Sig} \text{ and} \\
\qquad\quad \text{for all } i = 1, \ldots, k : (m^*[q], \sigma^*) \neq (m_i[q], \sigma_i). \\
\quad \text{return } 0
\end{array}
$$

Figure 10: Weak Blockwise Non-Interactive Public Accountability Experiment for SSS

Definition of **SSS Standard Privacy**:

> An SSS is said to be (standard) private, if for any efficient (PPT) adversary *A* the probability that the game depicted in the Figure 11 below returns **1**, is negligibly close to **½** (as a function of λ).

$$
\begin{aligned}
&\textbf{Experiment } \mathsf{Privacy}_{SSS,\mathcal{A}}(\lambda)\\
&\quad (pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \mathsf{KGen}_{\text{sig}}(1^\lambda)\\
&\quad (pk_{\text{san}}, sk_{\text{san}}) \leftarrow \mathsf{KGen}_{\text{san}}(1^\lambda)\\
&\quad b \leftarrow \{0,1\}\\
&\quad a \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, sk_{\text{sig}}, \cdots), \mathsf{Proof}(1^\lambda, sk_{\text{sig}}, \cdots), \mathsf{LoRSanit}(\cdots, sk_{\text{sig}}, sk_{\text{san}}, b), \mathsf{Sanit}(1^\lambda, \cdots, sk_{\text{san}})}(pk_{\text{sig}}, pk_{\text{san}})\\
&\qquad \text{where oracle } \mathsf{LoRSanit} \text{ on input of:}\\
&\qquad m_0, \text{MOD}_0, m_1, \text{MOD}_1, \text{ADM}\\
&\qquad \text{if } \text{MOD}_0(m_0) \neq \text{MOD}_1(m_1), \text{ return } \perp\\
&\qquad \text{if } \text{MOD}_0 \not\subseteq \text{ADM} \vee \text{MOD}_1 \not\subseteq \text{ADM}, \text{ return } \perp\\
&\qquad \text{let } (m, \sigma) \leftarrow \mathsf{Sign}(1^\lambda, m_b, sk_{\text{sig}}, pk_{\text{san}}, \text{ADM})\\
&\qquad \text{return } (m', \sigma') \leftarrow \mathsf{Sanit}(1^\lambda, m, \text{MOD}_b, \sigma, pk_{\text{sig}}, sk_{\text{san}})\\
&\quad \text{return 1, if } a = b
\end{aligned}
$$

Figure 11: Standard Privacy Experiment for SSS

The aforementioned privacy definition from [BF++09] only considers outsiders as adversarial. However, RERUM would also require that even insiders, i.e., sanitizers, are not able to win the game. Note, the signature generation key $\mathsf{sk}_{\mathsf{san}}$ is *not* generated by the adversary, but by the sanitizer, only known to it. The need for this alteration is hopefully clearer after the next definitions. For the definition of strong privacy, the basic idea remains the same: no one should be able to gain any knowledge about sanitized parts without having access to them, with one exception: the adversary is given the secret sanitizing key $\mathsf{sk}_{\mathsf{san}}$ of the sanitizer. This notion extends the definition of standard privacy to also account for parties knowing the secret sanitizer key. In a sense, this definition captures some form of "forward-security". Examples for strongly private SSS are the schemes introduced by *Brzuska* et al. [BF++09b] [BP12] [BPS14], as their schemes are perfectly private. As the adversary now knows $\mathsf{sk}_{\mathsf{san}}$, it can trivially simulate the sanitization oracle itself.

Definition of **SSS Strong Privacy**:

> An SSS is said to be strongly private, if for any efficient (PPT) adversary *A* the probability that the game depicted in Figure 12 below returns 1, is negligibly close to ½ (as a function of λ).

$$
\begin{aligned}
&\textbf{Experiment } \mathsf{SPrivacy}_{SSS,\mathcal{A}}(\lambda)\\
&\quad (pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \mathsf{KGen}_{\text{sig}}(1^\lambda)\\
&\quad (pk_{\text{san}}, sk_{\text{san}}) \leftarrow \mathsf{KGen}_{\text{san}}(1^\lambda)\\
&\quad b \leftarrow \{0,1\}\\
&\quad a \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, \cdot, sk_{\text{sig}}, pk_{\text{san}}, \cdot), \mathsf{Proof}(1^\lambda, sk_{\text{sig}}, \cdots, pk_{\text{san}}), \mathsf{LoRSanit}(\cdots, sk_{\text{sig}}, sk_{\text{san}}, b)}(pk_{\text{sig}}, pk_{\text{san}}, sk_{\text{san}})\\
&\qquad \text{where oracle } \mathsf{LoRSanit} \text{ on input of:}\\
&\qquad m_0, \text{MOD}_0, m_1, \text{MOD}_1, \text{ADM}\\
&\qquad \text{if } \text{MOD}_0(m_0) \neq \text{MOD}_1(m_1), \text{ return } \perp\\
&\qquad \text{if } \text{MOD}_0 \not\subseteq \text{ADM} \vee \text{MOD}_1 \not\subseteq \text{ADM}, \text{ return } \perp\\
&\qquad \text{let } (m, \sigma) \leftarrow \mathsf{Sign}(1^\lambda, m_b, sk_{\text{sig}}, pk_{\text{san}}, \text{ADM})\\
&\qquad \text{return } (m', \sigma') \leftarrow \mathsf{Sanit}(1^\lambda, m, \text{MOD}_b, \sigma, pk_{\text{sig}}, sk_{\text{san}})\\
&\quad \text{return 1, if } a = b
\end{aligned}
$$

Figure 12: Strong Privacy Experiment for SSS

In a weakly private RSS, a third party can derive which parts of a message have been redacted without gathering more information, as redacted blocks are replaced with □, which is visible. The basic idea is that the oracle either signs or sanitizes the first message (m_0) or the second (m_1). As before, the resulting redacted message m' must be the same for both inputs, with one additional exception: the length of both inputs must be the same, while □ is considered part of the message. For strong privacy, this constraint is not required. Note, that *Lim* et al. define weak privacy in a different manner: they prohibit access to the signing oracle [LLP12]. Our definition allows for such adaptive queries. Summarized, weak privacy only makes statements about blocks, not the complete message. See [KB08] for possible attacks. Weakly private schemes, following above definition, are, e.g., [HH++08] [KB08]. In their schemes, the adversary is able to pinpoint the indices of the redacted blocks, as □ is visible.

Definition of **RSS Weak Privacy:**

An RSS is said to be *weakly* private, if for any efficient (PPT) adversary *A* the probability that the game depicted in below Figure 13 returns 1, is negligibly close to ½ (as a function of λ).

Experiment $\mathsf{WPrivacy}_{\mathcal{RSS},\mathcal{A}}(\lambda)$
$(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$b \leftarrow \{0,1\}$
$d \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, sk, \cdot), \mathsf{LoRRedact}(\cdots, sk, b)}(pk)$
where oracle LoRRedact
for input $m_0, m_1, \mathrm{MOD}_0, \mathrm{MOD}_1$:
if $\mathrm{MOD}_0(m_0) \neq \mathrm{MOD}_1(m_1)$, return $\perp$
Note: redacted blocks are denoted □, which are considered part of m
$(m, \sigma) \leftarrow \mathsf{Sign}(1^\lambda, sk, m_b)$
return $(m', \sigma') \leftarrow \mathsf{Redact}(1^\lambda, pk, m, \sigma, \mathrm{MOD}_b)$.
return 1, if $b = d$

Figure 13: Weak Privacy Experiment for RSS

The next definition is similar to weak privacy. However, redacted parts are *not* considered part of the message.

Definition of **RSS Strong Privacy:**

An RSS is said to be strongly private, if for any efficient (PPT) adversary *A* the probability that the game depicted in the Figure 14 below returns 1, is negligibly close to ½ (as a function of λ). This is the standard definition of privacy found in [BB++10].

Experiment $\mathsf{SPrivacy}_{\mathcal{RSS},\mathcal{A}}(\lambda)$
$(pk, sk) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$b \leftarrow \{0,1\}$
$d \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda, sk, \cdot), \mathsf{LoRRedact}(\cdots, sk, b)}(pk)$
where oracle LoRRedact
for input $m_0, m_1, \mathrm{MOD}_0, \mathrm{MOD}_1$:
if $\mathrm{MOD}_0(m_0) \neq \mathrm{MOD}_1(m_1)$, return $\perp$
Note: redacted blocks are *not* considered part of the message
$(m, \sigma) \leftarrow \mathsf{Sign}(1^\lambda, sk, m_b)$
return $(m', \sigma') \leftarrow \mathsf{Redact}(1^\lambda, pk, m, \sigma, \mathrm{MOD}_b)$.
return 1, if $b = d$

Figure 14: Strong Privacy Experiment for RSS

Definition of **SSS Immutability**:

> A sanitizable signature scheme SSS is *immutable*, if for any efficient algorithm *A* the probability that the experiment from the Figure 15 below returns 1 is negligible (as a function of λ) [BF++09].

Experiment $\mathsf{Immutability}_{SSS,\mathcal{A}}(\lambda)$
$(pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m^*, \sigma^*, pk^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda,\cdot,sk_{\text{sig}},\cdot,\cdot),\mathsf{Proof}(1^\lambda,sk_{\text{sig}},\cdots)}(pk_{\text{sig}})$
let $(m_i, \text{ADM}_i, pk_{\text{san},i})$ and σ_i for $i = 1, \ldots, q$ be queries/answers to/by Sign
return 1, if:
$\mathsf{Verify}(1^\lambda, m^*, \sigma^*, pk_{\text{sig}}, pk^*) = \texttt{true}$ and
for all $i = 1, 2, \ldots, q : (pk^*, m^*[j_i], \text{ADM}^*) \neq (pk_{\text{san},i}, m_i[j_i], \text{ADM}_i)$ and
if $(m^*[j_i], \text{ADM}_i, pk_{\text{san},i}) \neq (m_i[j_i], \text{ADM}_i, pk_{\text{san},i})$, also $j_i \notin \text{ADM}_i$
where shorter messages are padded with $\perp$

Figure 15: Immutability Experiment for SSS

For weak immutability, an adversary knowing, but not generating, the sanitizer key must only be able to sanitize admissible blocks. Hence, once more pk_{san} is fixed and cannot be changed by the adversary.

Definition of SSS Weak Immutability:

> A sanitizable signature scheme SSS is *weakly immutable*, if for any efficient algorithm *A* the probability that the experiment given in the Figure 16 below returns 1 is negligible (as a function of λ).

Interestingly, weak immutability is enough for the construction to become unforgeable, while for an RSS used in the normal way, this definition is obviously not suitable at all due to accountability reasons. We omit the security parameter λ for the rest of the section to offer an increased readability.

Experiment $\mathsf{WImmutability}_{SSS,\mathcal{A}}(\lambda)$
$(pk_{\text{sig}}, sk_{\text{sig}}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(pk_{\text{san}}, sk_{\text{san}}) \leftarrow \mathsf{KeyGen}(1^\lambda)$
$(m^*, \sigma^*) \leftarrow \mathcal{A}^{\mathsf{Sign}(1^\lambda,\cdot,sk_{\text{sig}},pk_{\text{san}},\cdot),\mathsf{Proof}(1^\lambda,sk_{\text{sig}},\cdots,pk_{\text{san}},\cdot)}(pk_{\text{sig}}, pk_{\text{san}}, sk_{\text{san}})$
let (m_i, ADM_i) and σ_i for $i = 1, 2, \ldots q$ be queries/answers to/by Sign
return 1, if:
$\mathsf{Verify}(1^\lambda, m^*, \sigma^*, pk_{\text{sig}}, pk_{\text{san}}) = \texttt{true}$ and
$\forall i, i = 1, 2, \ldots, q : (m^*[j_i], \text{ADM}^*) \neq (m_i[j_i], \text{ADM}_i)$ and
if $(m^*[j_i], \text{ADM}_i) \neq (m_i[j_i], \text{ADM}_i)$, also $j_i \notin \text{ADM}_i$
where shorter messages are padded with $\perp$

Figure 16: Experiment for SSS weak immutability

2.3.9.8.1 Implications and Separations of the Cryptographic Properties.

Theorem 1: There exists an RSS, which is only weakly private.

Proof: For a proof see for example in [HH++08] [IIKO11] [IK++09] [LLP12].

Theorem 2: Every SSS which is immutable, is also weakly immutable.

Proof: Trivially implied: *A* generates the sanitizer key pair honestly.

Theorem 3: There exists an SSS, which is private, but not strongly private.

Proof: The proof is done by modifying an arbitrary existing strongly private SSS. Let SSS=($KGen_{sig}$,$KGen_{san}$,Sign,Sanit,Verify,Proof,Judge) be an arbitrary private SSS. Next the private SSS is altered into SSS' =($KGen'_{sig}$,$KGen'_{san}$,Sign',Sanit',Verify',Proof',Judge') as follows:

- $KGen'_{sig} := KGen_{sig}$
- $KGen'_{san} := KGen_{san,}$ while an additional key pair for a IND-CCA2-secure encryption scheme ENC is generated.
- Sign' is the same as Sign, but it appends the encryption *e* of a digest of original message to the final signature, i.e., σ`=(σ,*e*), where $e \leftarrow ENC(pk_{san}, H(m))$ and H() some cryptographic hash-function.
- Sanit' is the same as Sanit, while it first removes the encrypted digest from the signature, appending it to the resulting signature.
- Verifiy' , Proof' and Judge' work the same as their original counterparts, but removing the trailing *e* from the signature before proceeding as the originals.

Clearly, a sanitizer holding the corresponding secret key for ENC, can distinguish between *messages* generated by the signer and the sanitizer using the encrypted information contained in *e* within the *signature* σ'=(σ,*e*). Without knowledge of the encryption key, which is part of sk_{san}, this information remains hidden due to the IND-CCA2 encryption.

2.3.9.8.2 Definition of a Secure RSS and a Secure SSS.

Especially note that accountability, as defined for SSS in [BF++09], will be researched as part of RERUM's work plan in the near future and can be found in deliverable D3.2. This is because Redact is a public algorithm. Hence, no secret sanitizer key(s) are required for redactions. To circumvent this inconsistency, this research suggests utilize a standard SSS and let the signer generate the sanitizer key sk_{san}, attaching it to the public key of the signer. This also explains why pk_{san} is fixed in the security model. If any alteration without sk_{san} is possible, the underlying SSS would obviously be forgeable. As previously it was defined that this would mean that this SSS is non-secure this case is omitted. Hence, the secret sk_{san} becomes public knowledge and can be used by every party. This is the reason why the adversary only knows sk_{sig} but cannot generate it. These at first sight very unnatural, restrictions are required to stay consistent with the standard security model of SSS that have been formalized in [BF++09] by Brzuska et al. This security model is very common and well established in the research community and the current body of work. Hence, the urge to stay consistent and compatible with this important security model for SSS. Moreover, the signer is generally *not* considered an adversarial entity in RSS [BB++10]. If other notions or adversary models are used, the results may obviously differ.

In the following sections, we show that any SSS achieving only standard privacy, is not enough to construct a weakly private RSS and additional impossibility results.

2.3.9.8.3 Generic Transformation

This section presents the generic transform. In particular, a generic algorithm is provided that transforms any weakly immutable, strongly private, and weakly blockwise non-interactive publicly accountable SSS into an unforgeable and weakly private RSS.

The basic idea of the transform is that every party, including the signer, is allowed to alter *all* given blocks. The verification procedure accepts sanitized blocks, if the altered blocks are □. □ is treated

as a redacted block. Hence, redaction is altering a given block to a special symbol. As it was defined that an SSS only allows for strings m[i] $\in \{0,1\}^*$, we need to define $\square := \varnothing$ and m[i]$\leftarrow$0, if m[i]= $\varnothing$ and m[i]$\leftarrow$m[i]+1 else to codify the additional symbol $\square$. Here, $\varnothing$ expresses the empty string. Hence, we remain in the model defined. Moreover, this is where weak blockwise non-public interactive public accountability comes in: the changes to *each* block need to be detectable to allow for a meaningful result, as an SSS allows for arbitrary alterations. As $\square$ is still visible, the resulting scheme is only weakly private, as statements about m can be made. This contradicts our definition of strong privacy for RSS. Moreover, as an RSS allows every party to redact blocks, it is obvious that sk_{san} must be known to every party, including the signer. Therefore, we need a strongly private SSS to achieve our definition of weak privacy for the RSS, the proof follows after the construction.

Let SSS=($KGen_{sig}$,$KGen_{san}$,Sign,Sanit,Verify,Proof,Judge,Detect) be a secure SSS. Define RSS=(KGen,Sign,Redact,Verify) as follows:

Key Generation:

> Algorithm KeyGen generates on input of the security parameter λ, a key pair (pk_{sig}, sk_{sig}) ← $SSS.KeyGen_{sig}(1^{\lambda})$ of the SSS and also a sanitizer key pair (pk_{san}, sk_{san}) ← $SSS.KeyGen_{san}(1^{\lambda})$. It returns (pk,sk) = (($pk_{sig,}$ pk_{san}, sk_{san}), sk_{sig}). Where pk=($pk_{sig,}$ pk_{san}, sk_{san}) is considered the public key.

Signing:

> Algorithm RSS.Sign on input m $\in \{0,1\}^*$, sk, pk sets ADM = (1,…l) and computes σ ← $SSS.Sign(1^{\lambda}$, m, sk_{sig}, pk_{san}, ADM). It outputs: (m, σ).
>
> Redacting:
>
> Algorithm RSS.Redact on input message m, modification instructions mod, a signature σ, keys pk=($pk_{sig,}$ pk_{san}, sk_{san}), first checks if σ is a valid signature for m under the given public keys using RSS.Verify. If not, it stops outputting ⊥⊥. Afterwards, it sets mod'= {(i, $\square$) | i $\in$ mod}. In particular, it generates a modification description for the SSS which sets block with index i $\in$ mod to $\square$. Finally, it outputs (m',σ') ← $SSS.Sanit(1^{\lambda}$, m, MOD', σ, pk_{sig}, sk_{san}).

Verification:

> Algorithm RSS.Verify on input a message m $\in \{0,1\}^*$, a signature σ and pk first checks that ADM = (1,…l) and that σ is a valid signature for m under the given public keys using SSS.Verify. If not, it returns false. Afterwards, for each i for which $SSS.Detect(1^{\lambda}$, m,MOD', σ_s, pk_{sig}, pk_{san} ,i) returns San, it checks that m[i] = $\square$. If not, it returns false. Else, it returns true. One may also check, if sk_{san} is correct and that all m[i] are sanitizable, if required.

Theorem 4: The above Generic Transformation Construction is Secure

If the utilized SSS is weakly blockwise non-interactive publicly accountable, weakly immutable and strongly private, the resulting RSS is weakly private, but not strongly, and unforgeable.

Proof: Proving this requires to show that the resulting RSS is unforgeable and weakly private, but not strongly private.

Each property is proven on its own.

- **Unforgeability.** Let A be an algorithm breaking the unforgeability of the resulting RSS. We can then construct an algorithm B which breaks the weak blockwise non-interactive public accountability of the utilized SSS. To do so, B simulates A's environment in the following way:

  1. B receives pk_{san}, sk_{san}, pk_{sig} and forwards them to A
  2. B forwards every query to its own signing oracle
  3. Eventually, A outputs a tuple (m*,σ*)
  4. If (m*,σ*) does not verify or is trivial, abort
  5. B outputs (m*,σ*)

  m cannot be derived from any queried message, with the exception of m[i] = ☐ for any index i. Hence, $\exists$ i : m[i] $\neq$ ☐, which has not been signed by the signer. The accepting verification requires that Sig=Detect(1^{λ},m*,σ*,pk_{sig},pk_{san}). Therefore, (m*,σ*) breaks the weak blockwise non-interactive publicly accountability. The success probability of B equals the one of A.

- **Weak Privacy.** To show that our scheme is weakly private, we only need to show that an adversary A cannot derive information about the prior content of a contained block m[i], as ☐ is considered part of the resulting message m' and all other modifications result in a forgeable RSS. Let A be winning the weak privacy game. We can then construct an adversary B which breaks the strong privacy game in the following way:

  6. BB receives pk_{san}, sk_{san}, pk_{sig} and forwards them to A
  7. B forwards every query to its own oracles
  8. Eventually, A outputs its guess b*

  B outputs b* as its own guess. The oracle requires that $mod_1(m_1) = mod_2(m_2)$, disregarding ☐ Note, the messages are the same. Hence, the success probability of B is the same as A's. This proves the theorem.

- **No Strong Privacy.** Due to the above, we already know that our scheme is weakly private. Hence, it remains to show that it is not strongly private. As a redaction leaves a *visible* special symbol, i.e., ☐, an adversary can win the *strong* privacy game in the following way: Generate two messages m_0 and m_1, where m_1=(m_0,1), so there is one additional part in m_1. Hence, they differ in length such that $l_0 < l_1$, while m_0 is a prefix of m_1. Afterwards, it requests that the only different part $m_1[l_1]$ is redacted, i.e., $mod_1 = (l_1)$ and $mod_0 = ()$. Hence, if the oracle chooses b=0, it will output m_2=m_0 and for b=1, m_2 = (m_1, ☐). Hence, the adversary wins the game, as (m_1, ☐) is easily detectable to be different from m_0.

As RSS allow for removing every block, we require that adm= (1,…,l). This rules out cases where a signer prohibits alterations of blocks. This constraint can easily be transformed into the useful notion of consecutive disclosure control [MHI06] [SP++12].

2.3.9.8.4 Requirements to Transform a SSS into a RSS

In this section, it is shown that standard private SSS are not enough to build weakly private RSS. Moreover, it is proven that weak blockwise non-interactive public accountability is required to build an unforgeable RSS. To formally express these intuitive goals, we need the following Theorems:

Theorem 5: Any non-strongly private SSS results in a non-weakly private RSS.

If the transformed SSS is not strongly private, the resulting RSS is not weakly private.

Proof: Let A be an adversary winning the strong privacy game as defined previously. Then one can construct an adversary B, which wins the weak privacy game as defined previously, using A as a black-box:

2. B receives the following keys from the challenger: pk_{san}, sk_{san}, pk_{sig} and forwards them to A
3. B simulates the signing oracle using the oracle provided
4. Eventually, A returns its guess b*
5. B outputs b* as its own guess

Following the definitions, the success probability of B equals the one of A. This proves the theorem.

Theorem 6: No Transform can Result in a Strongly Private RSS. There exists no algorithm which transforms a secure SSS into a strongly private RSS.

Proof: Once again, every meaningful SSS must be immutable, which implies weak immutability due to Theorem 2. Hence, no need to make any statements about schemes not weakly immutable. It remains to show that any transform T achieving this property uses a SSS' which is not weakly immutable. Let RSS' denote the resulting RSS. One can then derive an algorithm which uses RSS' to break the weak immutability requirement of the underlying SSS in the following way:

1. The challenger generates the two key pairs of the SSS. It passes all keys but sk_{sig} to A
2. A transforms the SSS into RSS' given the transform T
3. A calls the oracle SSS.Sign with a message m=(1,2)
4. A calls RSS'.Redact with mod= (1)
5. If the resulting signature σ does not verify, abort
6. AA outputs (m', σ_{SSS}) of the underlying SSS

As $l_m \neq l_{mod(m)}$, meaning that the length is different, the tuple with the modified m, denoted as (mod(m), σ_{SSS}) breaks the weak immutability requirement of the SSS. Moreover, as hiding redacted parts of a message is essential for strong privacy, no algorithm exists, which transforms a weakly immutable SSS into a strongly private RSS, as adm needs to be correctly recoverable. This proves the theorem. This concrete example is possible, as only required behaviour was used.

Theorem 7: Weak Blockwise Non-Interactive Public Accountability is required for any Transform T. For any transformation algorithm T, the utilized SSS must be weakly blockwise non-interactive publicly accountable to result in an unforgeable RSS.

Proof: Let RSS' be the resulting RSS from the given SSS. Perform the following steps to show that the used SSS is not weakly blockwise non-interactive publicly accountable. In particular, let A be an adversary winning the unforgeability game, which is used by B to break the weak blockwise non-interactive public accountability of the used SSS.

1. The challenger generates the two key pairs of the SSS. It passes all keys but sk_{sig} to B
2. B forwards all received keys to A
3. A transforms the SSS into RSS' given the transform T
4. Any calls to the signing oracle by A are answered genuinely by B using its own signing oracle
5. Eventually, A returns a tuple (m', σ_{RSS}) to B
6. If the resulting signature does not verify or does not win the unforgeability game, A and therefore also B abort
7. B outputs the underlying message-signature pair (m', $\sigma_{SSS'}$)

Following the definition (m', $\sigma_{SSS'}$) breaks the weak blockwise non-interactive public accountability requirement of the SSS, as there exists a block, which has not been signed by the signer, while the signer will be accused by Detect. Moreover, the success probabilities are equal. The contrary, i.e., if the SSS used is not weakly blockwise non-interactive publicly accountable, the proof is similar. To achieve the correctness requirements, our accountability definition must hold blockwise.

Theorem 8: No Unforgeable RSS can be transformed into an SSS. There exists no transform T, which converts an unforgeable RSS into an unforgeable SSS.

Proof: Let SSS' be the resulting SSS. Now perform the following steps to extract a valid forgery of the underlying RSS:

1. The challenger generates a key pair for an RSS. It passes pk to A.
2. A transforms RSS into SSS' given the transform T
3. A calls the oracle RSS.Sign with a message m=(1,2) and simulates SSS'.Sign with adm=(1)
4. A calls SSS'.Sanit with mod= (1,a), a $\in_R \{0,1\}^{\lambda}$
5. If the resulting signature does not verify, abort
6. Output the resulting signature σ_{RSS} of the underlying RSS

As (a, 2) $\notin$ $span_{|=}(m)$, ((a,2), σ_{RSS}) is a valid forgery of the underlying RSS. Note, this concrete counterexample is possible, as only required behaviour is used.

2.4 List of Key-Material needed in RERUM

Different cryptographic mechanisms require different keys. As RERUM foresees that very diverse cryptographic mechanism can be used in the course of the project, it therefore considers a diversity of cryptographic key-material, too. The following tables show potential key-material that shall be handled / generated / distributed within RERUM if the mechanism is used. This key-material list provides information about each key, including where possible the foreseen length in bits, which is important for the impact on communication when transport keying material and for the storage.

2.4.1 Notation and Description of Key Material Needed and Generated during Credential-Boot-Strapping

Notation for Key Material	Description of Key Material and its use	Potential Format and Size (indicative)
JK_RDx_SC	The join key is a symmetric key to establish a first security association between the new RERUM Device and the Security Center. The key allows a RERUM Device to join the RERUM network.	Symmetric (128 bit)
NK_PAN	The network key is a symmetric key used for security on layer 2 (802.15.4). Each IEEE 802.15.4 PAN cluster has its own network key which is distributed during the credentials bootstrapping process to RERUM Devices and RERUM Gateways.	Symmetric (128 bit)
K_RDx_SC	Each RERUM device gets during the credential bootstrapping procedures a symmetric key to protect further communication with the SC based on DTLS.	Symmetric (128 bit)
CERT_SC	The Security Center's public key must be installed and marked as trusted on RERUM Devices. This is a public key from an asymmetric crypto scheme that serves as the root of trust for the flat public key infrastructure (PKI).	X509 certificate

2.4.2 Notation and Description of Key Material Needed and Used during Network Security Protocols

Notation for Key Material	Description of Key Material and its use	Potential Format and Size (indicative)
K_RDx_SC	Each RERUM device gets during the credential bootstrapping procedures a symmetric key to protect further communication with the SC based on DTLS.	Symmetric (128 bit)
RAW_SC	Each RERUM Device gets during the credential bootstrapping procedures a raw public/private key pair in the form of limited X509 certificate or in the design specific format.	X509 certificate (limited version) or design specific format
CERT_SC	The Security Center's public key must be installed and marked as trusted on RERUM Devices. This is a public key from an asymmetric crypto scheme that serves as the root of trust for the flat public key infrastructure (PKI).	X509 certificate
CS-based key generation	The key is symmetric and used to encrypt/decrypt information at the application layer	Size can vary

2.4.2.1 Notation and Description of Key Material Needed and Used during Integrity and Origin Authentication Keys

RERUM further follows RFC 4949 that states an association to be established beforehand "... an association to exist between the two entities ..." [RFC4949]. In the case of origin authentication RERUM needs the means to check the presented identity. This, so called, security association can be build either upon the knowledge of a shared or corresponding token on both sides.

Notation for Key Material	**Description of Key Material and its use**	**Potential Format and Size (indicative)**
MACKEY	Using Message Authentication Codes (MAC) a verifier and claimant both have access to a shared secret; the authentication protocol will make use of symmetric encryption and decryption as cryptographic primitives involving this token as a key. Usable by protocols like HMAC [ISO/IEC 9797-1]	HMAC_SHA256 is 256 bit size HMAC_SHA512 is 512 bit size
IBE_params, IBE_masterkey, IBE_privkey	Using Identity based authentication the claimant has access to a secret key that corresponds to his ID and every verifier has the trusted public parameters of the TTP, denoted IBE_params. This is called identity-based cryptography (IBE) and is in the group of public-key cryptography. The ID is a string representing the claimant's publicly known identity, like the physical device address or the IP address. The authentication protocol is based on the fact that the verifier can derive the claimant's public key from the ID and the public parameter; this can be facilitated in an authentication protocol based on asymmetric IBE encryption and IBE decryption or to run a signature verification algorithm. The public parameters IBE_params, must be trusted and known to the verifier. There also needs to be a trusted third party that is able to issued the IBE_privkey to the entitiy that shall be able to carry out private operations, e.g. signing or decrypting, this TTP needs to safeguard IBE_masterkey. As the IBE_masterkey that is held by the TTP can be used to generate the secret keys for any arbitrary ID string.	depends on the concrete scheme selected
GPub-Key, Memberkey_i, GOpeningKey	Group signatures allow identity based authentication and, at the same time, provide k-anonymity for the signers. For verification, the group public key is used. The length of the group public key is not affected by the group size, see [CS97]. The members of the group use their individual member keys to generate group signatures of a size between 171 and 200 Bits. The group opening key, which is used to relink a group signature to a member key, is of the same length as the group public key. It is provided to the Security Center.	Same size as any private (Memberkey) and public key (GPubKey) for system entities. The opening key equals the size of the group public key, see [CS97].

sk_{san}, pk_{san}	Key pair (Secret key and Public key) belonging to and identifying the sanitizer in a malleable signature scheme.	Same size as any secret and public key for system entities.
CERT_SC, sk_{sig}, pk_{sig}	The Security Center's key must be pre-installed and marked as trusted on RERUM Devices to identify messages originating from the Security Center (SC). CERT_SC contains (among potentially other information) the public key from an asymmetric crypto scheme that is used to authenticate the Security Center or to send encrypted messages only to the Security Center. Additionally CERT_SC serves as the root of trust for the flat public key infrastructure (PKI). For authentication the claimant has access to the secret signature generation key sk_{sig} that corresponds to the verifier's signature verification key pk_{sig}; the authentication protocol is based on signature generation and signature verification algorithms using the public and secrets parts of an asymmetric key as a token. The claimant's public key must be trusted or it must be traceable to a trusted origin, like the CERT_SC. Hence, this is commonly known as a public key infrastructure (PKI). RERUM assumes this to be ECC based keys, with a security of an ECC keysize of at least 256bit. For the X509 encoding see RFC 3279, RFC 5480, and RFC 5758.	a 256-bit elliptic curve needs two 32-byte values, giving a total key size of 64 bytes X.509 probably has a size in the order of of around 688 Bytes[24]
IMPL_PK, IMPL_PARAMS, IMPL_SK, IMPL_CERT, IMPL_CA_SEC	Implicit certificates nicely reduce the amount of operations necessary to verify a signature and the CA-issued certificate of the public key holder. It does this in a combined operation that is less heavy than consecutive checking of the signer's signature on the data and the CA's signature on the signer's public key (e.g. the signer's certificate). This is shown possible in ECC, especially in Elliptic Curve Qu-Vanstone (ECQV).	Implicit certificates has size equal to points, e.g. in a 160 bit ECQV[25], the implicit certificate has 160 bits

Finally, let us note that the security association can be built upon a proof of knowledge of a token by the claimant, without the need to revealing it, using so called zero-knowledge proofs. RERUM has no plans to investigate the latter further in this Deliverable; however, there might be interesting applications in RERUM like scenarios, so these types of protocols are subject to be revisited in D3.2.

[24] `openssl ecparam -genkey -name prime256v1 -out key.pem`
`openssl req -new -key key.pem -out csr.pem` providing the standard values
`openssl req -x509 -days 365 -key key.pem -in csr.pem -out certificate.pem`
`ls –la certificate.pem` yields a size of 688 Bytes

[25] Elliptic Curve Qu-Vanstone (ECQV): one kind of implicit certificates.

2.5 Key Management in RERUM

Keys being used for the authentication of origin must be bound to the entities that they represent, and in RFC 4949 there exists a definition of the term system-entity.

“system-entity: A system entity that is the subject of a public-key certificate and that is using, or is permitted and able to use, the matching private key only for purposes other than signing a digital certificate; for example, an entity that is not a Certificate Authority. The subject of a digital certificate is created by using a Distinguished Name (DN) which may contain the entity’s identity information.” [RFC4949]

The above definition from [RFC4949] is in very close relation to entity authentication mechanisms that builds on certified public-key, so some form of PKI. While RERUM will not forbid such structures to be used, it does not mandate them, nor thinks that they are extremely suitable for resource constrained device environment, without adaptation. Hence, this section states the approach of RERUM.

RFC 4949 points to X509-public-key-certificates; a data structure for holding public keys defined by X.509. Currently in version 3 such an X.509 public-key certificate contains a sequence of fields, and finally a digital signature computed over that sequence of fields. Following [Coo+08] the minimum fields found are:

- Version
- Serial number
- Signature Algorithm
- Issuer
- Validity (time period)
- Subject
- Subject’s Public Key Information

RERUM will seek to provide a more suitable representation, or reduce fields in order to reduce the sheer size this information needs.

2.5.1 Certificate Chain Verification Overhead Reduced by Flat Hierarchy

Basing the entity authentication on a Public Key Infrastructure (PKI) implies that the verifier needs to verify and validate public-key-certificates, not only the certificate of the other entity, but also all relevant certificates if a chain is provided to establish trust to the entity that is authenticated. The steps are the following:

1. Detect / extract /obtain the public-key-certificate of the claimant
2. Detect / extract /obtain the certificate chain that connects the claimant’s public-key-certificate to a trusted entity that the verifier trust
3. Cryptographic Verification of the claimant’s public-key-certificate including its certificate chain
4. Validation of the claimant’s public-key-certificate involving the collection of signed status information of a public-key-certificate from the certificate authority.

If you take the example of verifying an X509-public-key-certificate then you must have a trusted public key of the certificates issuer in order to generate useful output from the execution of cryptographic algorithms.

If public-key-certificates shall be used to control who can still be participating in the RERUM application and who shall not, then the public-key-certificates can be marked as valid only for some time, or even before that pre-defined time marked as invalid. For example if a claimant's key pair has been compromised or if the device holding this key has been sold or stolen. This process of invalidating a certificate before its expiry date is called certificate revocation [AL11]. If such a revocation happens, step 4, the process to retrieve the current status information, which would contain the information about this revocation from a certificate authority, also called certificate validation, is important.

The use of certificate chains is done in order to deduce some trust from the hierarchical structure used in a Public Key Infrastructure. The idea is that a number of initially trusted entities (trust anchors, or root of trust) are implanted during setup by the security administrator, e.g. during the credential bootstrapping this is the key of the Security Center (SC) denoted by CERT_SC. Trust here is meant as the trust that is managed/transported by the means of a Public Key Infrastructure [Jø13] as meaning: Trust is a directional relationship between two parties that can be called the relying party and the trusted party.

To deduce this trust, several so-called Trust Models can be used. Different kinds of trust models have been developed and described with their corresponding trust relationships [Jø13] [MC96][SM95][JP04][Woe06]:

The overhead for this checking and decoding increases linearly with each intermediate layer for each certificate in the certificate chain, the trust model's security properties must be fulfilled. This results in the need to carry out step 3 the cryptographic verification, but also the validation. Thus, the verifier needs to verify a certificate as valid at verification time t. IETF in RFC 1421 – 1424 defined three common trust models, further discussion can be found in [Lin93][Ken93][Bal93][Kal93]:

- Shell Trust Model: At verification time all certificates in the chain need to be valid
- Modified Shell Trust Model: At signature generation time, which is before the verification, all certificates in the chain need to be valid
- Certificate Chain Trust Model: Following [Woe06] this is the weakest; each certificate in the path was valid at the moment of use

Generally, the overhead for each certificate is generated by execution of the cryptographic signature verify algorithm once to gain Cryptographic Verification (step 3). However, if required the certificate additionally needs to be checked for the validity (step 4). This might again need additional cryptographically secured communication. Might, because it may be enough to just checking validity periods against the current time. But if getting a notion of current time on a constrained device might require getting it from a trusted source, and as such requires secure communication. Another reason for additional communication overhead involving cryptographic security protection is when requesting the current status, i.e. revoked or not revoked, from a trusted third party.

To get the current time on the RD might be required by other functions of RERUM, in case of this re-usability the extra effort might be well spent [PV11], but it must be noted that to get current time securely is not an easy task, and comes with additional costs.

Under the supervision of Henrich C. Pöhls, Markus Doering listed in his M.Sc. thesis[26] all the steps that need to be carried out when validating and verifying each hierarchical level in a certificate chain. As this overhead is linear with the number of certificates in that chain, which corresponds to the level of hierarchies in the PKI, the overhead can be reduced when the hierarchy is flattened.

RERUM proposes to use the only existing separation of duties to build a flat PKI hierarchy: RERUM Gateway is the root of trust and Certificate issuer and RERUM Devices under this Gateway get their public-keys all certified by the Gateway.

RERUM wants to build on the cryptographic primitives best suited. We have identified that ECC based crypto is promising in terms of speed and can be implemented to run in a variety of constrained devices [LN08, SO++08, HS13]. To reduce the overhead in key management and certificate verifications we use the flat PKI, enabling us to more or less directly save the ECC point values of the trusted keys inside the RERUM Device, as there is foreseeable only a limited number of those keys needed, e.g. CERT_SC. Also, RERUM plans to facilitate the internal workings of ECC by using implicit certificates or curves where one only needs one point, instead of two to become even more efficient.

In Figure 17 below you can see how RERUM could use implicit certificates to distribute the keys for a later lean device to device authentication based on implicit certificates. After this has been done by all devices, each device could authenticate all other devices that are below the same RERUM Gateway. The RERUM Gateway (RG) acts as the Local CA (LCA). The Security Center (SC) acts as the global CA (GCA) that is in the hierarchy above the local CAs.

[26] Steps and Challenges in Certificate based Entity Authentication - Illustrated with Practical Implementations of TLS by Markus Doering in fullfilment of his M.Sc. degree at the UNI PASSAU

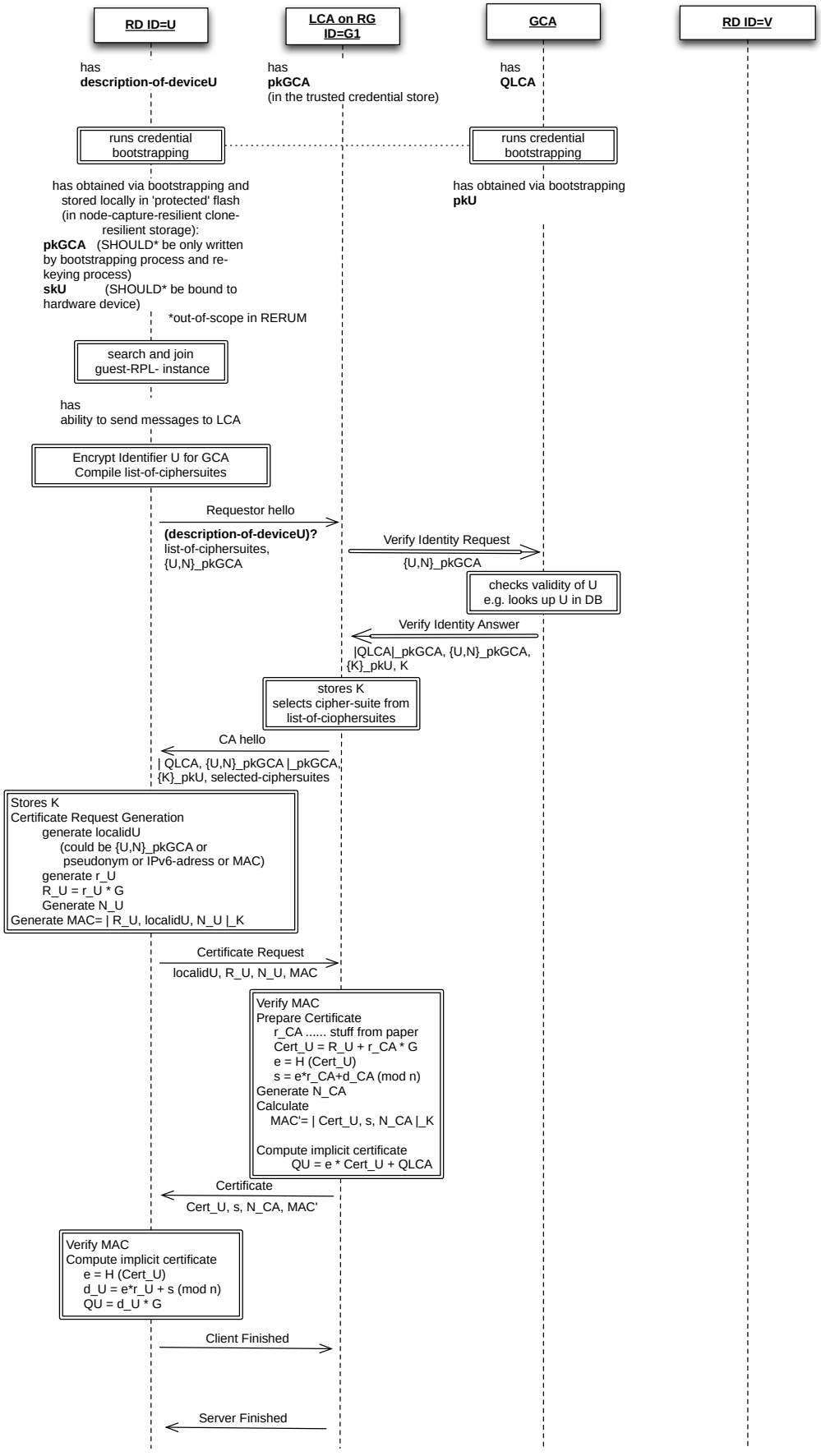


Figure 17: Use of Implicit Certificates in a flow of messages to allow obtaining key material for a later Device to Device authentication

2.5.2 Trusted Credential Store

According to D2.3, the trusted credential store 'is responsible for storing the credentials in a secure and trusted way. In this respect, the store restricts unauthorized read or write access to credentials stored in it. Regarding read and write access this means that RERUM assumes to store secret credentials, like cryptographic tokens and keys in this store and assumes that the access to those is restricted to only the component that is authorized to access the credential. With respect to only write access, the Trusted Credential Store allows providing a public, but read-only, list of public keys that are associated with another component or service and that this relation is marked as trusted. This would allow storing a public-key certificate of the application server, a certification authority, or another trusted entity here'.

RERUM differentiates between those RDs capable of running a Java Virtual Machine (JVM) and those that not. RDs that cannot run or do not want to run a JVM due to hardware constraints, such as the Z1, provide their own custom trusted credential store. Besides, they are custom implementations, their design is out of the scope of this document.

Regarding those RDs capable of running a JVM, which normally act as servers hosting middleware or security components, such as the gateway instance or the PRRS server, the selected trusted credential store is the java class java.security.CertStore for direct java invocation or, alternatively, its command line counterpart keytool. Both of them support PKCS#12 certificates, the ability to work with both symmetric and asymmetric keys and they also support protecting the keys themselves via a password. Complete documentation of the keytool tool is accessible at http://docs.oracle.com/javase/8/docs/technotes/tools/#security, the technical documentation for the java.security.CertStore class can be found at the jce documentation at oracle.com[27]

There is, however, an important note regarding the use of the new encrypting mechanisms of RERUM. No matter what mechanism is used for cyphering or signing any object, it is necessary a code for implementing them, and the new signing mechanisms for RERUM are not an exception. For this reason, in order to use the new keys in RERUM, it is necessary to develop the code for using them. In the concrete case of the java classes for cryptography, it is possible to use existing classes, including CertStore, for combining them with the new types of keys providing only the code to deal with the new key without having to rewrite the whole communication libraries. This, in practice, means that for any application trying to use CertStore to hold the new keys, it is necessary to write a security provider able to deal with that code. Complete documentation about how to write a security provider for java is available the Oracle guide to implement a security provider[28].

[27] Java KeyStore Class: http://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html

[28] Java Security Provider:
http://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/HowToImplAProvider.html

3 Secure Communication

In the following RERUM will explain four different, but freely combinable, mechanisms to secure the communication, which RERUM in this deliverable will call *profiles*. The communication security in RERUM is logically grouped around security associations (SAs) between communicating entities. At the end of each of the four profiles this chapter states between which entities the profile can be used to establish secure communication, and which protection goal (integrity, authentication and confidentiality) the profile accomplishes.

We recall below, the Fig. 39 from Deliverable D2.3 on RERUM's architecture (see Figure 18).

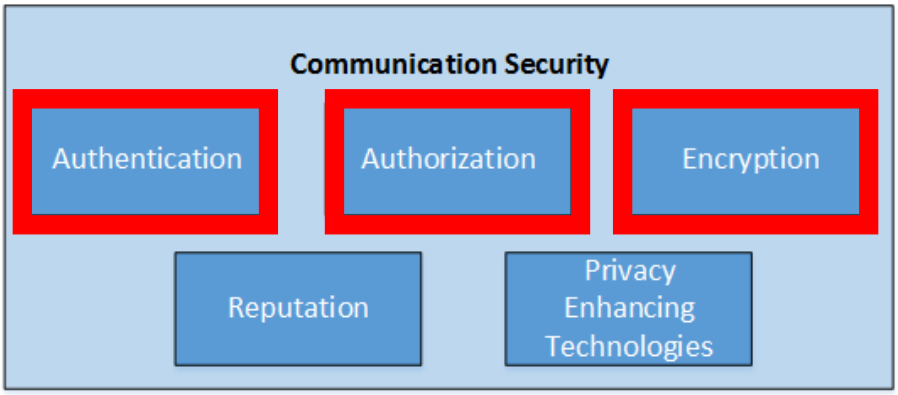


Figure 18: Security components in the communication layer from D2.3

The profiles described in this chapter will be able to provide the functionalities of: encryption, integrity checking, authentication and authorization. The profiles are designed to be flexible enough to encompass future RERUM developments in the direction of reputation (Deliverable D3.3) and privacy (Deliverable D3.2).

3.1 Profile DTLS

3.1.1 Introduction, Motivation and Link to User Requirements

Among others, the main factors of successful future Smart Cites lays in secure and reliable IoT networks. Considering the complexity of task, both these requirements are not easy to fulfil in practice by one general and elegant solution. Focussing only on security issue, there exist many different solutions to address main security requirements such as confidentiality, integrity and origin authentication. In this section we introduce the first of our security profiles, namely DTLS profile, in which all these three requirements are met, especially device-to-device authentication.

DTLS is a well-known security protocol that recently received significant attention in IoT community. One of the major factors of this attention lays in maturity of the said protocol, meaning that DTLS has been widely used by non-IoT world. This suggests that in case of successful adoption in IoT, e.g., in RERUM Devices, DTLS might be used not only as secure communication protocol between RERUM Devices, but in much more interoperable way, i.e., between RERUM Devices and a cloud server, which supports the DTLS protocol. Although DTLS seems to be theoretically applicable, the real issues come with a practical deployment in constrained environments. Security schemes, which are key components of the said protocol, demand (especially those public-key based) significant computational power in order to be efficient. This efficiency usually depends on selected security schemes and underlying platform. Such efficiency, i.e., memory requirements, speed and power consumption is a subject of future investigation in D5.1.

Described DTLS profile is applicable to any devices in RERUM architecture that requires secure communication (either end-to-end or hop-by-hop) at the transport layer level. The profile can be used transparently by any end applications, which themselves resident in the application layer of OSI model. This transparent feature implies that DTLS profile could be applied to all four presented in D2.1 Use-Case scenarios and also directly addresses the following User Requirements:

- UR-7: The user needs to protect his measurements from malicious users;
- UR-25: User requires open solutions for authentication between devices, ensuring the integrity of their data as well as the confidentiality.

That is, user requires his messages to not be forged by malicious users and user requires open solutions for authentication between devices, ensuring the integrity of their data as well as the confidentiality.

3.1.2 DTLS Protocol

Transport Layer Security (TLS) [DR08] is a cryptographic protocol, designed to allow a secure communication over the Internet between two parties (usually referenced as peers or as a client and a server). In peer-to-peer networks the client is a peer that initialises communication, whereas the server is the peer that responds on the client initialisation. TLS protocol is widely used by many popular applications such as web browsers, e-mails or voice-over-IPs. It belongs to a class of TCP/IP protocols and residents in the session and presentation layers in the standard OSI model. TLS itself consists of two layers, namely TLS Record Protocol (residents in OSI presentation layer) and TLS Handshake Protocol (residents in OSI session layer). The former is deployed to encapsulate higher-level protocols from OSI application layer, providing secure and reliable connection. The latter is used to authenticate client and server with each other (or optionally only one peer), negotiate cryptographic keys and algorithms used later in TLS Record Protocol. Both TLS Record and Handshake Protocols run over a reliable TCP channel and are transparent for higher-level protocols. The reliability feature of a TCP channel means that TLS protocol might assume that no single package is lost and all are delivered in the correct order, which is essential for some cryptographic algorithms mode of operations.

Since TLS requires reliable TCP channel to communicate, it is not suitable for networking protocols based on unreliable communication such as User Datagram Protocol (UDP) and to address this issue a new Datagram Transport Layer Security (DTLS) protocol has been introduced [MR04]. In general, DTLS is a modified version of TLS designed to handle issues associated with unreliable connection, i.e., DTLS provides a mechanism that allows packet retransmissions and reordering whenever it is required. DTLS reassembles also most of the TLS features and design choices and provides equivalent security guarantees. Due to lower packet header overheads, code footprint and RAM requirements UDP is considered as a better option for Wireless Sensor Networks (WSN) than TCP. This implies DTLS as a potentially better and more suitable choice for a cryptographic protocol in WSN. Taking into consideration those arguments, Internet Engineering Task Force[29] (IETF) standardisation community appointed the DTLS In Consternated Environments (DICE) group [DTLS15] to investigate applicability of the mentioned standard to constrained devices. The main tasks of the group are ``to define DTLS profile suitable for Internet of Things'', ''to define how DTLS record layer can be used to transmit multicast messages securely'' and ``investigate practical issues around the DTLS handshake in constrained environments'' [DTLS15]. As mentioned, DTLS is a transport layer protocol and it is independent from application layer, thus is able to prove security objectives such as authenticity, integrity and confidentiality to application layer protocols i.e., it might be easily integrate with

[29] IETF is an open standard organisation with a strong focus on developing Internet standards and protocols.

emerging application layer protocol designed especially for constrained environments, so-called Constrained Application Protocol (CoAP) [BFSH12].

3.1.3 DTLS v1.2 Handshake

As mentioned in the previous section, DTLS consist of handshake and record protocol. The main purpose of the handshake protocol is to authenticate communication parties, negotiate a cipher suite and exchange keys that are later used to protect traffic data. Focusing only on authentication, the handshake allows three options to be considered: no authentication, unilateral (a client or a server) authentication and mutual authentication, i.e. both client and server are authenticated. The handshake protocol in DTLS is very similar to the TLS handshake, with few significant changes, which are adopted to handle unreliable transport protocol. The first important addition is an introduction of a stateless cookie, which is exchanged over the first phase of the handshake. Its main purpose is to prevent Denial-of-Service (DoS) attacks[30]. The exact prevention technique has been adopted from [KS99]. The second difference is a modification of the handshake header in order to support message losses and message reorders, whereas as the third main addition, one can mark the introduction of retransmit timers in order to support aforementioned massage losses [MR04].

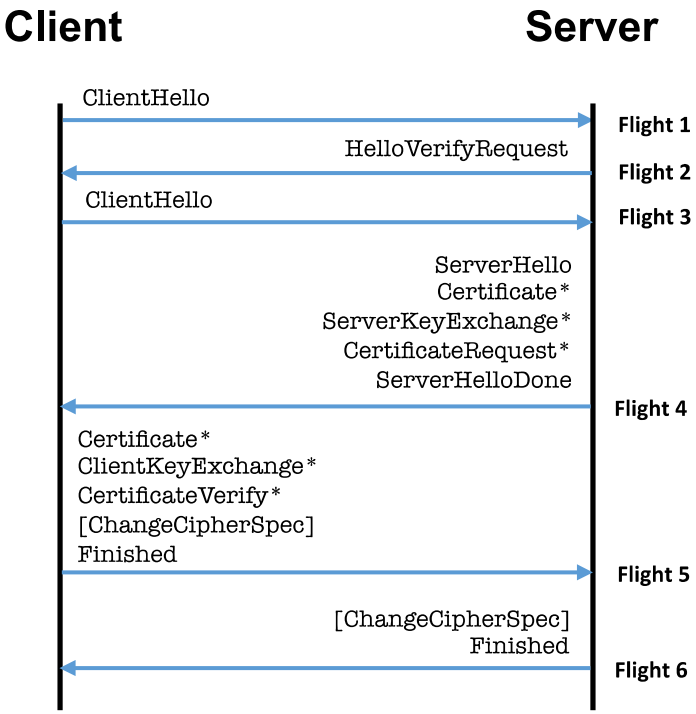


Figure 19: Overview of the handshake protocol

All handshake messages can be divided on six groups of messages (so-called flights) between the communication parties (cf. Figure 19); three flights are sent from a client to a server and three from a server to a client. The client initialises the first flight by sending the `ClientHello` message. The message contains a 32-bit random value, used later on to generate shared keys. The server answers with the `HelloVerifyRequest` message (mark on the Figure as the second flight), which contains a stateless cookie. After receiving the message from the server, the client (the third flight) retransmits

[30] A type of attack where adversary tries to make a target machine or a service unavailable for users.

the ClientHello message, which includes the cookie. The server validate client's cookie and has a possibility to terminate handshake whenever the validation fails or proceeds with the fourth flight. It is worth mentioning, that the DoS prevention mechanisms mitigates possible DoS attacks with spoofed IP[31] addresses (when the client is not able to retransmits a cookie) but does not prevent DoS from valid IP addresses. These three above-mentioned flights are present only in DTLS and are not present is such a form in original TLS protocol. Additionally, in cases where the raw public key mode is considered, the third flight, i.e., the ClientHello message contains extension of ClientCertificateType and ServerCertificateType, which indicates what types (raw or X.509 certificates) of public keys are going to be used, e.g., both sides can use raw public keys or only one side can use raw keys while other can use X.509 certificates.

The fourth flight starts with the ServerHello message sent by the server and consists of at least two messages. The exact number and types of messages is depended on mode of the handshake, i.e., for pre-shared keys this flight consist only ServerHello and ServerHelloDone messages, whereas for the raw keys and X.509 certificates additional messages such as Certificate, ServerKeyExchange and CertifacateRequest are included. The ServerHello message contains a 32-bit random value and selected cipher suite, which is based on the available cipher suite information received from the client. Similarly like 32-bit value received form the client in the ClientHello message, the random value sent by server is used in the process of generation of shared keys. If pre-shared keys are the considered option, the flight is closed with the ServerHelloDone message, whereas for the public-key option, the server sends its certificate (included in the Certificate message). In case of the full X.509 certificates option, the client (after receiving certificate) can start validating the certificate chain, i.e., by using information provided by the certificate. Whenever certain types of cipher suites are considered, i.e., with ECDH key exchange, the server might start generating ephemeral DH public and secret keys. The ephemeral DH public key is then encapsulated (together with the elliptic curve parameters) and signed using the server's public-key in the ServerKeyExchange message.

Then next batch of messages (the fifth flight) is the client replay to the server's fourth flight. The client sends his certificate in Certificate massage (if public-key mode is considered), ClientKeyExchange, which contains the client's ephemeral DH public-key and the CertifcateVerify message, which contains the signed hash of previous message flights. Upon sending the CertifcateVerify message, both sides exchanged all necessary information to calculate a secret session key. First, both sides calculate ECDH shared secret key, following by calculation using Pseudo-Random Function, where both sides uses ECDH secret keys and exchanged random values. The PRF output is used as a session key to encrypt traffic data. This flight might also consist the ChangeCipherSpec message, which indicates that data messages will be protected using the newly negotiated cipher suite and keys. The flight is ended with the Finished message, in which data are encrypted under negotiated keys and consist of the hash value of all flights. The server responses in a similar way with the sixth flight, i.e., by sending the ChangeCipherSpec and Finised messages. After the successful handshake, the server and the client are authenticated to each other (or only one side depends on the option used) and possesses the shared key (negotiated in case of use public keys), which is used to encrypt and authenticate transferred data.

As mentioned above, the DTLS protocol consists of both secret (a pre-shared key) and public-key (raw public keys and X.509 certificates) options. All three are described in more details below.

3.1.3.1 Pre-Shared Keys

A pre-shared key technique applies to the scenario where secret keys (symmetric keys) are shared between peers before communication is established [ET05]. Following the argumentation of [ET05],

[31] An attacker creates an IP packet inserting a trusted host IP as a source address.

one can distinguish at least two possible advantages for such scenario. The first one is a possibility to mitigate (depending on the cipher suite) a need of public-key operations, which in general, might be very costly (computation time, bandwidth etc.) especially for resource-constrained devices. The second argument concerns that in some deployments, e.g., where devices might be strictly controlled and connections are pre-defined, the manual configuration of the keys might be much more efficient than general key management techniques. On the other hand, the disadvantage of this approach comes with scalability, i.e., more deployed devices, the harder the key management problem becomes. In the worst-case scenario each peer has to handle a different key for each peer whom it wants to communicate with. This also increases a key storage cost issues, which are especially important in constrained environments. Mandatory cipher suite for pre-shared keys defined in the DTLS protocol standard is `TLS_PSK_WITH_AES_128_CCM_8`, which requires AES-128 in CCM[32] mode of operation with 8-octet authentication tag. The standard also assumes a default PRF, which includes HMAC[33] that utilise the SHA-256 hash function. It is worth noticing, that the pre-shared techniques without applying DH do not provide Perfect-Forward Secrecy (PFS), i.e., an attacker is able to decrypt old messages if keys are compromised, lack of which makes protocol deployment more vulnerable.

3.1.3.2 Raw Public Keys

A raw public key technique is the extension to the handshake protocol that has been introduced in [WT++14]. Similarly to X.509 certificates, it uses a public/secret key par but without a full support of X.509 certificates, which in general means that a key pair is generated by a device manufacturer (or deployment entity) and installed on a specific device.

In many DTLS/TLS deployments, a commonly used is an in-band procedure for validating client and server public keys. This is performed during a handshake protocol and this X.509-based procedure uses trust anchors to validate mentioned keys. In general its complexity might be high, especially for constrained environments. The less complicated variant uses so-called self-signed certificates, which are also commonly adopted especially in small deployments. In such a case, the certificate distribution is out-of-band, but generated certificates still uses full structure of X.509 certificates with all unnecessary information overheads. Apart from protocol-based methods for obtaining the client and the server public keys suggested in [WT++14], there is also possibility to include all necessary information in the firmware, i.e., a device is pre-configured with its public and private keys as well as a server address and together with its public key.

In order to reduce processing and storage overheads and avoid unnecessary overheads, DTLS may use the raw public keys option, which contains only a small subset of certificate structure, i.e., `SubjectPublicKeyInfo`. The certificate format still might be hold, i.e., exiting ANS.1 format is allowed but the certificate contains very limited information.

A raw public key option extends `ClientHello` and `ServerHello` messages in the handshake with `ClientCertificateType` and `ServerCertificateType` information leaving the possibility for a hybrid solutions, i.e., a client might indicate an use of raw public keys where server might indicate an use of the full X.509 certificate method (or vice versa). An authentication of a client and/or a server is done by using information carried out in the `SubjectPublicKeyInfo` field of the reduced certificate with a use of an out-of-band method. In case of raw public keys, a default cipher suite is set to `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8`, which uses NIST's secp256r1 curve, along with AES-128 block cipher in CCM mode and 8-octet authentication tag. To communicate effective with many peers many raw keys are allowed.

[32] (Counter with CBC-MAC), a mode of operation for cryptographic block ciphers

[33] Hash Message Authentication Code: Construction for calculating a message authentication code (MAC) involving a cryptographic hash function in combination with a secret key

3.1.3.3 X.509 Certificates

Apart from the raw public key option, DTLS might use a full X.509 certificates (details are in [C08]). This is the most common deployment scenario for DTLS/TLS, and assumes use of public key infrastructure (e.g., certificate authority) with X.509 certificate specifications. Each device houses a public/secret-key pair, as well as list of the trust anchors, which are used to validate certificates.

The certificate meta-data and data are expressed in Abstract Syntax Notation One (ASN.1) format, and handles three main sections, namely: certificate data, certificate signature algorithm, and certificate signature. The certificate signature algorithm holds a name of algorithm that is being used for signature, i.e., ECDSA, whereas the certificate signature field holds a signature of whole certificate data field. Further, the certificate data field holds additional information such as issuer, validity of certificate, as well as information about extensions (cf. [C08] for specific details). Similarly to raw public keys, this option must support `TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8` cipher suite, i.e., NIST's secp256r1 curve and ECDSA signature algorithm.

Although PKI with X.509 gives great flexibility, especially across different CA domains, it increases computation and bandwidth costs. The certificate validation chain procedure is one of the main issues. As another issue, it can be objected the information overheads that are needed to support this flexibility, which increases the storage costs and the computational costs for parsing the ASN.1 structures. The one of the solution of this problem is an aforementioned extension to DTLS to handle raw keys, which gives a possible trade-off between flexibility and associated overheads.

3.1.4 Analysis of the current state of software implementation

Currently, DTLS version 1.2 (open source implementation TinyDTLS[34]) has been integrated and run under Contiki on Re-Mote platform. The version could be configured to support both: pre-shared keys and certificate-based (no support for X.509, public/secret keys are provided as raw data in software code). Both versions run under emulator in Cooja, i.e., a handshake is successfully completed, as well as run on the real hardware platform (see Figure 20). Our test platform consists of two Re-Mote boards configured to communicate with each other. The optimization of the ECC-based version, running on the said test platform, is a subject of further development.

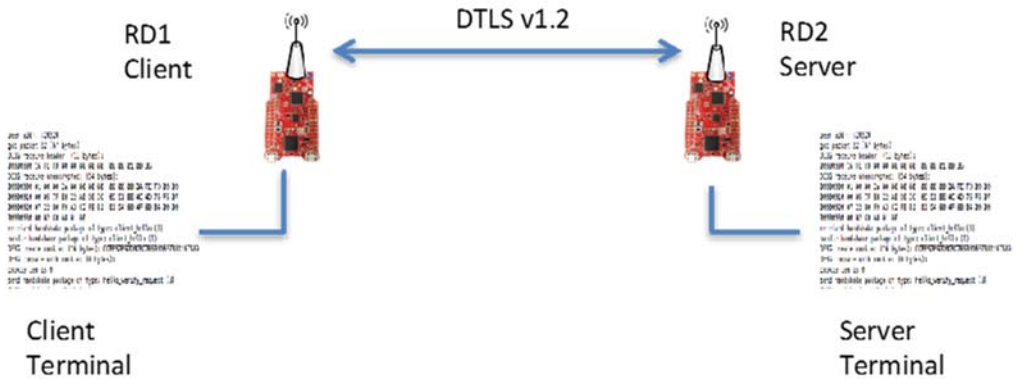


Figure 20: Prototype of DTLS on Re-Mote platforms

[34] http://tinydtls.sourceforge.net

3.1.5 Summary of Profile

Layer	Transport
Security Association provided between	• RERUM Device to RERUM Device (e.g. RD1-to--RD3 in Figure 2) • RERUM Device to RERUM Device – End-to-End (e.g. RD1-to-RD3 via RD2 in Figure 2) • RERUM Device to RERUM Gateway (e.g. RD3-to-RGW in Figure 2) • RERUM Device to RERUM Gateway – End-to-End (e.g. RD3-to-RGW via RD2 and RD1 in Figure 2) • USER to RERUM Gateway (e.g. USER-to-RGW in Figure 2) • USER to RERUM Device (e.g. USER-to-RD3 in Figure 2)
Provides Confidentiality	Yes
Provides Integrity	Yes
Provides Origin Authentication	Yes
End-2-End	Yes
Hop-2-Hop	Yes
Needed Key Material	`K_RDx_SC, RAW_SC, CERT_SC`

3.2 Profile On-Device-Signatures

3.2.1 Introduction, Motivation and Link to User Requirements

The result of D2.1 showed RERUM, that users would like to have the ability to identify if data was modified by unauthorized parties or in unauthorized ways. The goal of RERUM is to provide integrity on the basis of strong cryptographic primitives. This then becomes a building block to allow secure communication and secure authorization. The mechanism for Integrity is part of RERUM's functional component for secure communication (see deliverable D2.3). It must be flexible to be used with different signature schemes. It can be facilitated to protect the integrity and provide origin-authentication between any two entities for which a security association (SA) exists.

The on-device capability of signing and verification described here is one of the steps on RERUM's roadmap to fulfil Contribution 6: Implicit Certificate Based Device-to-Device Authentication as given in deliverable D2.1 (in the latest version).

In more detail, the on-device-signature profile is created to fulfil the following requirements in deliverable D2.2:

- Req. 2.6-1 Energy-efficient cryptographic primitives;

- Req. 2.6-2 Integrity protection of SL-I data in transit;
- Req. 2.6-3 Integrity protection of SL-I data at rest;
- Req. 2.6-4 Authorised modification of integrity protected data;
- Req. 2.6-5 Detection of authorised modification of integrity protected data
  if the underlying signature scheme allows it (i.e. malleable signatures);
- Req. 2.6-8 Device authentication;
- Req. 2.6-9 User authentication and
- Req. 2.6-18 Accountability
  if the underlying signature scheme and signature generation key can be used to identify the device and if the scheme gives accountability

As an example take the over-the-air programming capability of RERUM Devices. It is not secure enough to just send the update to the RDs and have them install it and then reboot: an attacker could send malicious software updates. To counter this attack, the origin of the software update must be authenticated by the receiving RD and the software received shall have not been modified in an unauthorized way. The capability to verify signatures on the RD allows RERUM to protect software updates by a signature. Only when the signature has been created by an entity with a signature generation key for which the verifying RD has a trusted verification key, the update will be installed.

This is where on-device signatures help to secure the communication across the complete IoT layers, if needed. On device signatures secure the integrity of the signed data end-to-end between devices, or between MW and devices or even from the device all the way to the applications outside RERUM.

The mechanisms described in this section enhance the capabilities on the RERUM Device. While this focus is important, please note this means that the signature created by a RD can be verified wherever it is still present in functional components of “upper” architectural layers. Also vice versa, a signature created by a component for which the RD has a trusted signature verification key can be verified as the originator of that message. Hence, the mechanisms described in this section will be found at many places inside the RERUM architecture, as seen in Figure 21. The profile can be used as indicated by the overlaid arrows allowing integrity protected communication.

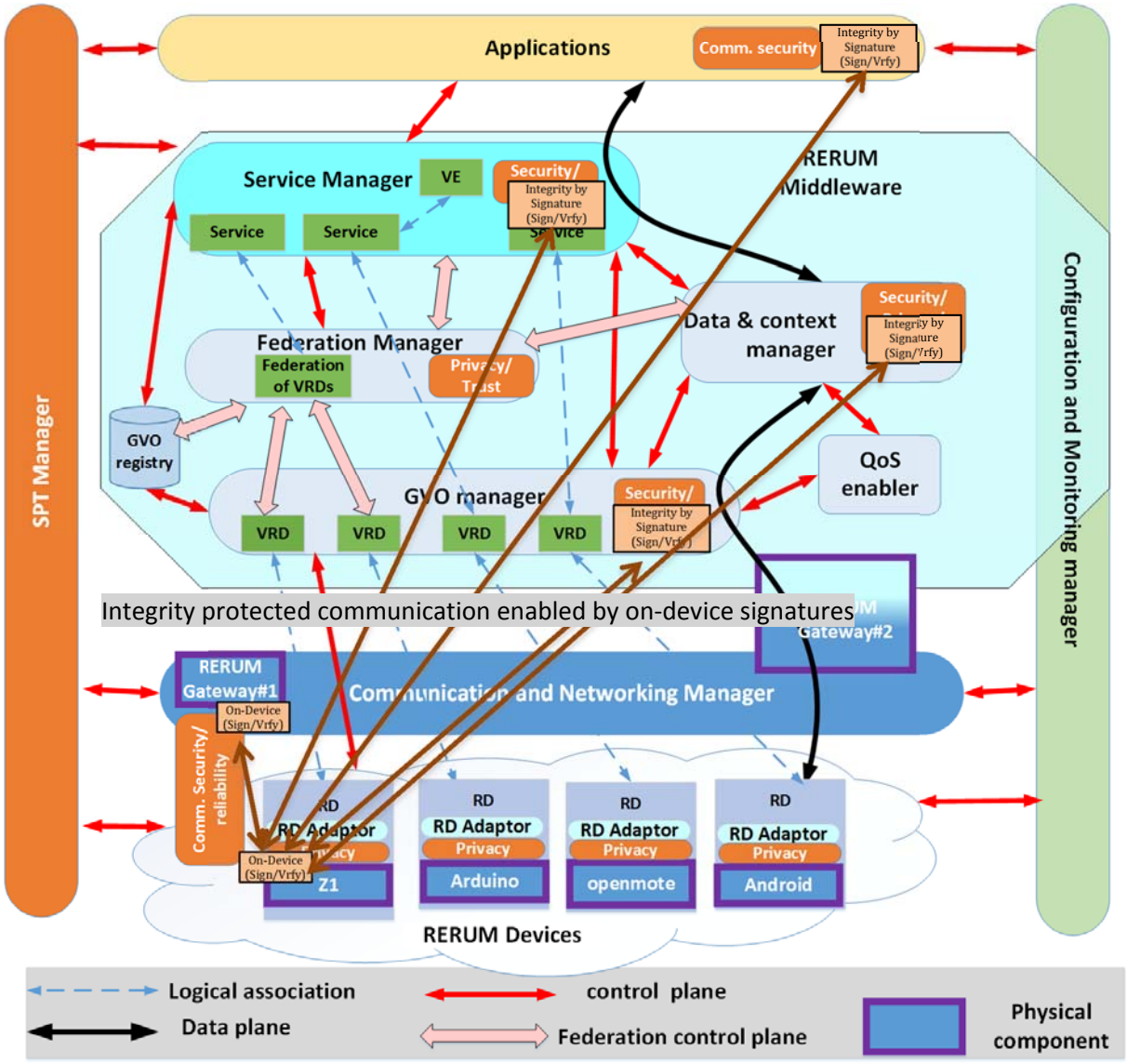


Figure 21: RERUM architecture (Fig. 29 from deliverable D2.3) with On-Device Signatures

Following the technical requirements for the capability to allow integrity protection in an end-to-end fashion all the way to the devices, RERUM uses as the integrity mechanisms ECC signatures on messages, such that h-data, e.g. the actual application level content such as sensed values or commands can be signed and verified on the RERUM Device, regardless of the underlying transport layer protections. That means that as this allows the signature to travel with the message, the message's integrity can be verified whenever the corresponding public verification key is known by the endpoint. This end-point can be on a RERUM Device or on the "Internet". This signature is applied on the h-data. H-data is application level data, e.g. the sensed value. Hence this signature mechanism's protection is compatible with all other Profiles (e.g. DTLS) and also when other transport layer mechanisms on the Internet are facilitated (e.g. TLS, IPsec). Note that the security of Internet Communication is not in RERUM's scope. However, the integrity protection and origin authentication is achieved standalone by this profile.

Further note, that the actual signature algorithm and the used key material steer the protection properties achieved by the signature over the h-data.

This also allows this profile to facilitate any of the previously described signature scheme:

- Classical signatures like ECDSA

- Group Signatures
- Identity based signatures
- Implicit Certificates
- Malleable Signatures

Note, the exact properties regarding integrity protection and origin authentication will depend on the selected algorithm. For this deliverable RERUM has researched the applicability of classical signatures on resource constrained devices (e.g. Zolertia's Z1). The next deliverable (D3.2) is planned to provide among others insight into the applicability of malleable signatures on resource constrained devices.

3.2.2 Analysis of the Current State and Selection of IETF Draft on JSON Web Signatures (JWS)

Data needs to be encoded for transport. XML is one standard way of doing it, but seems to be very heavy on the increase in message sizes and the parsing overhead is quite large. Looking at current trends on how data is represented, RERUM assumed that data could be emitted by the devices as JSON, and indeed the website for JSON list good arguments why JSON is superior over XML[35].

For RERUM the following reasons where of most importance and resulted in the selection of JSON as RERUM's data encoding format:

- JSON is a standard (ECMA-262)
- JSON format is text only, just like XML
- JSON is, compared to XML, a lightweight data-interchange format
- JSON is language independent
- JSON is "self-describing", easy to understand

In the following "**Example"** we show JSON formatted temperature value of 23.4 with some meta-data:

```
{
      "measurement_id": 3,
      "node_id": "foo_bar",
      "data": "23.4"
}
```

Following JSON further, this shall be transformed into a canonical representation of JSON. A canonical representation is needed to provide the same string representation repeatedly. Because signatures need to hash the JSON-encoded data during the signature generation. Canonical JSON shall remain parsable with any full JSON parser and also allow it to be arranged differently if the meaning is not changed. The signature generation and verification processes need to ensure that input is in canonical form before generating any hash of that input.
However, there is no standard for the transformation, which shall not be confused with what is called minification. There was an IETF draft, but that expired in 2013. The OLPC group[36] has documented the following procedure called canonical JSON that RERUM has slightly adopted:

- No whitespace.
- No escape sequences in strings other than \" and \\.
  All other characters must be represented literally, including control characters.

[35] http://www.json.org/xml.html

[36] http://wiki.laptop.org/go/Canonical_JSON

- No trailing commas.
- Object keys sorted by Unicode character values (code points).
  The sorting occurs before escape sequences are added.
- No decimal points in numbers (i.e. only integers allowed) or leading zeros. "-0" is not allowed.
- ~~OLPC: Encoded as UTF-8 in Unicode Normalization Form C~~.
  RERUM: Keep encoded as is

Note: The OLPC spec allows arbitrary byte sequences in strings, for easy storage of binary data. But this contradicts the JSON specification, which clearly states that "a string is a sequence of zero or more Unicode characters".

During canonicalization care must be taken not to remove whitespaces from values, but only from the JSON object. So for the example `"node_id": "foo bar"` must not become `"node_id":"foobar"`.

3.2.2.1 Steps to Generate the JSON-Sensor-Signature (JSS)

JWS is currently only in draft state at the IETF. RERUM in the following documents the steps of the current IETF draft. However, RERUM suggests to adapt them slightly to enable the signature to be less intrusive, than the JWS which results in an encoded instead of a plaintext h-data, e.g. RERUM allows non-signature aware processing steps to still access the h-data encoded in JSON in the same manner as if it would not be signed. For the purpose of differentiation RERUM will call these JSON-Sensor-Signature (JSS).

Note, the following steps are the thought example for the canonical JSON payload data from the above example in the beginning of Section 3.2.2. That is a single temperature of 23.4 with the metadata in one JSON object is going to be signed.

Note, using BASE64URL[37] [RFC4648] encoding allows to include actual UTF encoding of newlines or the JSON content being binary data etc. if they occur in the JSON data. For this simplified printed example all the encoded values have been truncated denoted by the `0`s.

As there was no BASE64URL encoding RERUM has developed a implementation for Contiki, which was missing, and plans to release this implementation as open source to Contiki within the scope of RERUM.

RERUM next describes how to generate the ECC signature using ECDSA build on a NIST p256 curve over a SHA256 to generate a JSON Sensor Signature (JSS).

0.) Key Setup for an ECC key based on curve secp256r1, that is the P-256 curve equivalently used in XML Signatures described as `http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256`

1.) Creation of the JSS header specifying the algorithm ECDSA/SHA256 is specified in the IETF draft (JSON Web Algorithms) as being referenced by ES256: `{"alg":"ES256"}`

2.) Encoding header as URL-safe Base 64: `eyJhbGciOiJFUzI1NiJ9Cg`

3.) Encoding of payload, the payload is the actual minified JSON object as seen in the above mentioned example. This will be URL-safe Base64 encoded: `eyAibWVh000000000000zLjR9Cg`

[37] Base 64 Encoding with URL and Filename Safe Alphabet

4.) Concatenation of the `header.payload`, so the base64url-encoded header followed by the character "." followed by base64url-encoded payload:
`eyJhbGciOiJFUzI1NiJ9Cg.eyAibWVh000000000000zLjR9Cg`

5.) Hash the concatenated string using the cryptographic hash function defined in the heard, here we have selected SHA-256

6.) Sign hash using the signature algorithm selected, here ECDSA, using the signer's secret key

7.) Base64url encode the generated cryptographic signature:
`MEYCwoBK000000000000emhDlLzc-27rM9KLVF8pA`

8.) Integrate the header and the signature alongside the payload JSON object

3.2.2.2 RERUM Proposal: Transporting JSS as Enveloped or Enveloping Signature

Following the current IETF draft the flattened JWS JSON Serialization Syntax would look like this:

```
{
 "payload":"<payload contents>",
 "protected":"<integrity-protected header contents>",
 "header":<non-integrity-protected header contents>,
 "signature":"<signature contents>"
}
```

However, this will, when following the IETF draft for JWS, always have the payload being base64url encoded. Because the IETF draft for JWS states that the "payload" member MUST be present and contain the value BASE64URL(JWS Payload)."[38].

RERUM does not want to require all non-signature interested parties to need BASE64 decoding. Henceforth we define two encodings for transporting the JSS along the JSON payload: enveloped and enveloping. This follows the ideas that XML signatures[39] have already long time.

Enveloped Signature:

```
{  <plain payload contents>,
   {"protected":"<plain integrity-protected header contents>",
   "signature":"<base64url encoded signature contents>"}
}
```

Enveloping JSON Sensor Signature (JSS) is around the payload:

```
{
   {<plain payload contents>},
   "protected":"<plain integrity-protected header contents>",
   "signature":"<base64url encoded signature contents>"}
}
```

A detached signature makes no sense, because (a) we want to keep the JSS with the JSON payload and (b) it would require to a have a message id that could be used to link the JSS object to the JSON payload

[38] https://tools.ietf.org/html/draft-ietf-jose-json-web-signature-41#appendix-A.7

[39] http://www.w3.org/TR/xmldsig-core/

object that was signed. Note that line breaks and indents are for readability and the size of the base64-encoded data is shortened.

Enveloped JSON Sensor Signature (JSS) is inside the payload:

```
{ "measurement_id":3,"node_id":"foo bar","data":"23.4",
 {"protected":{"alg":"ES256"},
  "signature":"MEYCwoBK000000000000emhDlLzc-27rM9KLVF8pA"
 }
}
```

Flattened and canonical JSON following OLPC (line breaks are only for this printed version):

```
{"data":"23.4","jws.protected":{"alg":"ES256"},"jws.signature":"MEYCwoBK000
emhDlLzc27rM9KLVF8pA","measurement_id":3,"node_id":"foo bar"}
```

3.2.3 Description of Application in order to fulfil the security requirement

This security mechanism of signing JSON data with a JWS protects against the Loss of U-DATA Integrity (Threat#01 from RERUM Deliverable D2.1).

An attacker can modify U-DATA while at rest or in transit.

- **At Rest**: If a user manages to modify U-DATA by executing malicious code on an SO or gateway, or by gaining remote access to one.
- **In Transit**: Through attacks on the network infrastructure. This can be the result of a man in the middle attack, by exploiting routing protocol, or neighbour discovery vulnerabilities.

If an attack is successful, the application will start providing incorrect S-DATA values. Of a more severe nature is the case of integrity loss of A-DATA, whereby a malicious user can trigger undesirable, potentially even privacy-breaching actuations, such as opening a window or turning on audio recording equipment.

This attack may be used as a facilitator to launch subsequent attacks against software confidentiality, integrity as well as availability, and it is thus considered of very high severity.

A more detailed scenario could be the following: Assume that certain environmental hazards when detected based on sensor reading would cause a costly reaction, e.g. dispatching special units of the fire brigade or evacuating a public building. The sensor's that monitor these conditions could be deployed with signature generation keys. The signature verification keys can be distributed among various parties as they are public. Then the sensors can be instructed to sign their readings before sending them to the gateway. The gateway could verify the signatures locally if required and if the public verification key(s) of the sensors are trusted and known. The gateway then could only process or forward correctly signed data. As this is end-to-end even an application behind the gateway or a functional component in the RERUM middleware could, if provided with access to the sensors' public verification key(s) and a way to evaluate that the public key belongs to that sensor be able to still verify the integrity and origin (by means of the identifying string that is linked to the key).

3.2.4 Research Prototype

To demonstrate and check the ability to implement and overhead induced, this mechanism in an early form was built as a research prototype. The setup is shown in the following Figure 22.

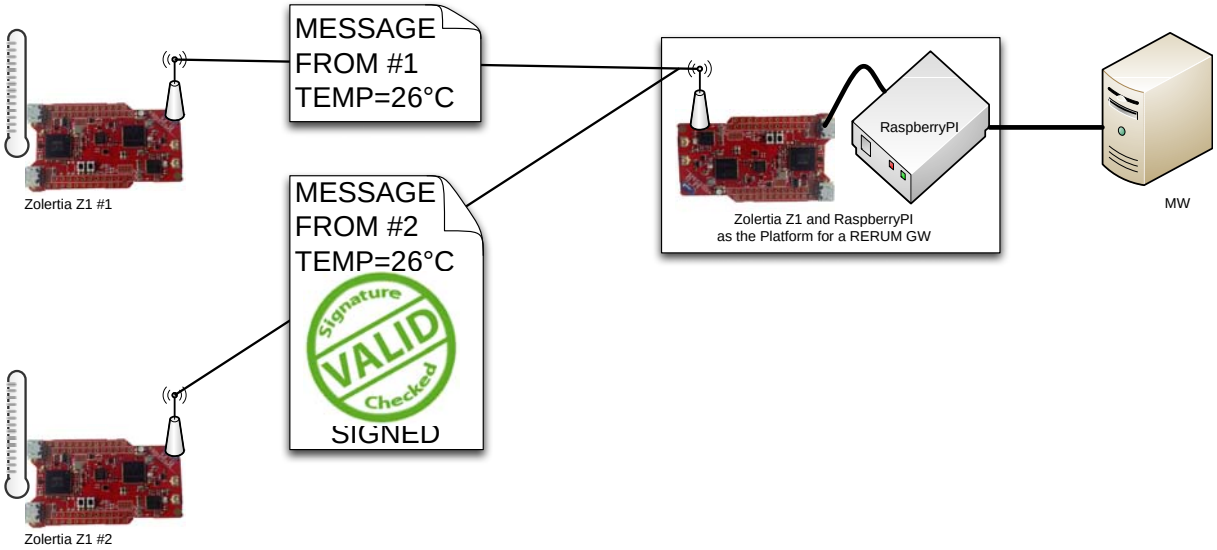


Figure 22: On-Device-Signatures prototype implemented using Zolertia Z1's by UNI PASSAU

The Zolertia Z1, compared to the RERUM Mote, Zoertia's Re-Mote[40], has limited memory (RAM and ROM). As a starting point RERUM used ecc-light[41]. This Contiki library was developed by employees of the National Institute of Standards and Technology (NIST), and others, that has been contributed to the public domain. From this available library, the core ECDSA implementation was singled out and the remaining code removed. Regarding the keys for the p160 NIST curve, the private key and a public certificate was generated on one Z1 and exported. Then the private key was hardly encoded into the signing Z1 (#2 in the above figure) and the public certificate was hard-coded into the Z1 that verifies (the one connected to the Raspberry PI in the above figure). During the prototyping the functionality for signing and verification was more and more stripped down. Then the JSON canonicalization method was implemented in Contiki. The prototype for RERUM's on-device-signatures was presented at Net Futures 2015[42]. RERUM currently gathers the results and plans to publish them together with the lessons learned when implementing certificates and JSON signatures to the academic community. Note, that this research prototype is limited to SHA256 for hashing and a to a 160-bit NIST curve. RERUM foresees to not continue implementing this on Re-Motes. RERUM already for the DTLS decided to use other ECC curves, i.e. Curve25519. The implementation of those is part of the DTLS mechanism. RERUM will continue to implement on Re-Motes, thus Z1 development will not actively continued. Future development will be on implementing digital signature schemes with a focus on privacy in D3.2, research prototypes are expected to be on the Re-Mote as part of the cryptographic development activities in WP3 and WP5.

[40] http://zolertia.io/product/hardware/re-mote

[41] https://github.com/nist-emntg/ecc-light-certificate

[42] http://ec.europa.eu/digital-agenda/en/news/net-futures-2015

3.2.5 Summary of Profile

Layer	Cross-Layer
Security Association provided between	RD and any other (RD, GATEWAY, APP): • RERUM Device to RERUM Device (e.g. RD1-to-RD3 in Figure 2) • RERUM Device to RERUM Device – End-to-End (e.g. RD1-to-RD3 via RD2 in Figure 2) • RERUM Device to RERUM Gateway (e.g. RD3-to-RGW in Figure 2) • RERUM Device to RERUM Gateway – End-by-End (e.g. RD3-to-RGW via RD2 and RD1 in Figure 2) • APP to RERUM Gateway (e.g. APP-to-RGW in Figure 2) • APP to RERUM Device (e.g. APP-to-RD3 in Figure 2)
Provides Confidentiality	No
Provides Integrity	Yes
Provides Origin Authentication	Yes
End-2-End	Yes
Hop-2-Hop	Yes
Needed Key Material	CERT_SC, sk_{sig}, pk_{sig} for classical ECDSA and possible others for other signature schemes

3.3 Profile 802.15.4 Security

3.3.1 Introduction, Motivation and Link to User Requirements

It is of no doubt that state-of-the-art communication networks cannot be deployed in a real world without any form of security support. IoT ecosystem is not any different, and as proven, e.g., in this section, there exists many different profiles that one can successfully apply to provide such a security support for IoT solutions. This section presents another profile in which confidentiality, integrity and origin authentication is provided, but the profile itself targets only layer 2 of OSI model, so-called link-layer. In general, a link-layer provides a node-to-node communication link (a link between two directly connected points in a network). Such a link extended with security mechanisms, in consequence will provide a secure connection between these two points. It is essential to note that security mechanism is only between directly connected nodes, and any security connection between more distinct nodes

(those which are not connected directly with each other in link-layer sense) will be possible only in hop-by-hop fashion (in contrast to descried in Section 3.1 DTLS profile). Although a hop-by-hop secure connection introduces a risk of communication being intercepted by a malicious node, the benefits come with a better efficiency and simplicity.

In 2003, the IEEE published the first version of IEEE 802.15.4 [IEEE06], a specification for the physical and link layer operation for low-power radio communication. The document also specifies a set of security services, which aim to secure the communication between devices.

Described 802.15.4 security profile is applicable to any devices in RERUM architecture that uses IEEE 802.15.4 standard to communicate within each other. The profile targets low level OSI layer which implies applicability to all four presented in D2.1 use case scenarios and directly addresses the following user requirements:

- UR-7: User requires his messages to not be forged by malicious users;
- UR-21: User needs to have reliable and energy efficient networking connectivity at his devices and
- UR-25: User requires open solutions for authentication between devices, ensuring the integrity of their data as well as the confidentiality.

i.e., user requires his messages to not be forged by malicious users; user needs to have reliable and energy efficient networking connectivity at his devices; user requires open solutions for authentication between devices, ensuring the integrity of their data as well as the confidentiality.

3.3.2 Overview of Layer 2 Security in IEEE 802.15.4

In order to achieve secure communication, the standard specifies that all the security services will use the AES algorithm with 128-bit keys. It is also important to note that the IEEE 802.15.4 standard allows the use of group keys, where a common key is used from a group of nodes (devices) mainly for multicasting and broadcasting. As mentioned in the standard, when a shared group key is used the provided protection is only against the outsider nodes and not against malicious nodes in the specific group, sharing the same key.

The specification does not provide details on how the keys are created or disseminated. It only mentions that the keys are provided by processes on higher layers and assumes that the keys are stored securely. The specification also does not discuss what kind of authentication policies can be applied.

3.3.3 Security Suites

The standard defines 8 different security suites, which can be used to provide different combinations of confidentiality, integrity and origin authentication. One of the suites provides no security, whereas the remaining 7 suites use one of the following AES modes of operation:

- Cipher Block Chaining Message Authentication Code (CBC-MAC), with Message Authentication Codes (MAC) of three different sizes
- Counter (CTR)
- CCM mode (Counter with CBC-MAC), with three different sizes of MACs

The table below presents an overview of those security services:

Security Level	Suite	Data Confidentiality	Data Integrity	Origin Authentication	Length of MAC (bits)
0	No security	No	No	No	N/A
1	AES-CBC-MAC-32	No	Yes	Yes	32
2	AES-CBC-MAC-64	No	Yes	Yes	64
3	AES-CBC-MAC-128	No	Yes	Yes	128
4	AES-CTR	Yes	No	No	N/A
5	AES-CCM-32	Yes	Yes	Yes	32
6	AES-CCM-64	Yes	Yes	Yes	64
7	AES-CCM-128	Yes	Yes	Yes	128

3.3.4 IEEE 802.15.4 Security with the Contiki OS

As of October 2014, the Contiki Operating System provides off-the-shelf support for IEEE 802.15.4 security services. In order to enable support for those security services, the user simply needs to set a configuration directive at build-time. For example, in order to select the AES-CCM-64 suite, the user would have to specify the following in the project configuration file:

```
#define NETSTACK_CONF_LLSEC        noncoresec_driver
#define LLSEC802154_CONF_SECURITY_LEVEL  6
```

The first line enables the Contiki's layer 2 security sub-system, whereas the second line selects the AES-CCM-64 suite. In order to switch between among suites 1 and 7, the user simply has to change the value of LLSEC802154_CONF_SECURITY_LEVEL to the desired security level.

Contiki's source tree includes a software implementation of the AES algorithm, but it also provides drivers for some AES acceleration hardware, such as the CC2420.

A detailed description of Contiki's security services and configuration system is out of the scope of this deliverable, but interested readers may find more information in Contiki's blog and wiki pages[43, 44,45].

Currently, the implementation uses pre-shared keys, but efforts have been made to add support for key establishment mechanisms and group communication schemes [KRM13, IOT13].

3.3.5 Summary of Profile

This profile provides Hop-2-Hop confidentiality, integrity and Origin Authentication at the Device Communication Layer, in the case where devices communicate with one another using IEEE 802.15.4

[43] https://github.com/contiki-os/contiki/pull/557

[44] http://contiki-os.blogspot.se/2014/10/a-big-step-for-contiki-built-in.html

[45] https://github.com/kkrentz/contiki/wiki

wireless radio links. Keys are symmetric and they always have a length of 128 bits. All cryptographic operations use the 128-bit AES algorithm.

From a RERUM perspective, this functionality constitutes a protective mechanism that can be used to mitigate threats Threat#01 to Threat#09 inclusive, as listed in Section 3.6 of D2.1. The functionality is transparent to the system's user as well as to citizens. The functionality is relevant to all RERUM Use-Cases.

Layer	Device Communication Layer
Security Association provided between	RERUM Device to RERUM Device – Single Hop Only (e.g. RD1-RD3 in Figure 2) RERUM Device to RERUM Gateway – Single Hop Only (e.g. RD1-RGW in Figure 2)
Provides Confidentiality	Yes
Provides Integrity	Yes
Provides Origin Authentication	Yes
End-2-End	No
Hop-2-Hop	Yes
Needed Key Material	NK_PAN

3.4 Profile Lightweight and Secure Encryption Using Channel Measurements[46]

3.4.1 Introduction, Motivation and Link to User Requirements

Wireless sensor networks are the building blocks of the Internet-of-Things architectures, conveying critical and often sensitive and private information. As these networks often consist of severe resource constrained devices, lightweight encryption mechanisms are of paramount importance for achieving energy efficiency. However, the majority of the proposed algorithms do not fully fulfil the requirements for energy efficiency. Furthermore, key distribution schemes are necessary for their proper operation, making the network vulnerable to adversaries that manage to capture the keys during key exchange. In this section, we describe a scheme where key extraction is performed using channel measurements, thus there is no need for any key distribution mechanism. The derived keys are used for encryption/decryption using the primitives of the compressed sensing (CS) theory [CW08], which allows encryption and compression simultaneously. The evaluation results show that legitimate nodes experience a very low reconstruction (decryption) error. Of course, a given error can be justified depending on the specific application, as some applications can tolerate a higher error and other cannot. At the same time, adversaries located at a distance greater than half of the carrier frequency's wavelength, experience a higher error, thus being unable to capture sensitive information.

[46] Part of this work has been published in Wireless Vitae 2015 conference in Aalborg, Denmark [FTT14].

A variety of algorithms for secure communications in several layers of the communications stack has been proposed, for example, the Elliptic Curve Diffie-Hellman, ContikiSec, RSA, etc. Most of these algorithms have a number of important inefficiencies:

- They are not optimized to be energy efficient. This is critical as sensors are severe resource constrained devices in terms of memory, CPU, and processing;
- Key distribution schemes are required. This consumes valuable energy, and there is also the risk of information hijacking (by an adversary) during the key exchange and
- Keys have to be pre-stored in the sensor device, usually during manufacturing. This poses a significant security threat as sensors can be easily compromised when placed in outdoor environments.

Here, we address all these issues by proposing an algorithm that does not require any key distribution scheme. The encryption key between two communicating parties is created using channel measurements, and more specifically, utilizing the Received-Signal-Strength-Indicator (RSSI), available in all off-the-shelf wireless transceivers. The algorithm we consider for encryption uses the principles of the relatively new theory of compressed sensing (CS). CS-based encryption is lightweight and provides acceptable security against adversaries [OAS08], [RB08]. Also, CS performs a lossy compression on the encrypted data; hence encryption and compression take place simultaneously.

Related work contains several significant contributions. Mathur et al. [MVTM11]utilize the change in phase of a transmitted signal for secret key generation. They show that [RB08] the speed with which nodes securely communicate depends on the variations of the monitored channel. This method however has two disadvantages for use in a WSN: (i) it requires an external signal source (e.g. an FM transmitter) to emit energy, and (ii) it uses the phase of the signal; a metric that is not available in commodity hardware like the sensors. Other related contributions [OPJC++K13], [LXMT06] consider the Received-Signal-Strength or the channel amplitude for key generation. All the proposed techniques generate keys that are suitable for common encryption algorithms (e.g. RSA, AES, etc.). Our work focuses on key generation for CS-based encryption. CS has the major advantage of performing lightweight encryption and compression simultaneously, so energy-efficiency becomes feasible. To the best of our knowledge, the only work that considers key generation using channel measurements for CS-based encryption is described in [DT13]. Our work differs in two areas: (i) we use experimental data collected from a real WSN, and (ii) we show how the reconstruction error is affected using different quantization levels.

The algorithm described here fulfils the security requirements of the innovation table for *RSSI-based CS encryption keys* (described in RERUM Deliverable D2.1). More specifically, the user requirement linked to this work is

- UR-17: «The end user (or the service provider) needs to install a CS key for each one of the *deployed devices. The user needs the key to be changed dynamically and not be pre-stored on the device. The user also needs to avoid manual installation or update of the key*.» The compression scheme described in this section fulfils these requirements as: (i) it does not require any key pre-storage in RDs, (ii) it does not require any key distribution mechanism, and (iii) keys can be updated dynamically.

The advantages of the proposed RSSI-based scheme (energy efficiency, and secure communication) make it ideal for use in three of the proposed RERUM use cases (described in RERUM Deliverable D2.1). Lightweight encryption and compression, as well as on-the-fly secret generation can be used for the environmental monitoring where a large number of unattended sensors may be used. In this case, a centralised secret key distribution scheme may not be feasible. Furthermore, energy efficiency is important as sensors' battery replacement operations can become very difficult when the sensors are deployed in harsh environments.

Regarding the home energy management, and the comfort quality monitoring use cases, apart from the energy efficiency of the deployed algorithms, privacy is a major concern as part of the monitoring data (e.g. energy consumption, home appliances used, etc.) can reveal sensitive data (e.g. daily habits, etc.) to an outsider. The algorithm proposed here can highly secure the collected data, as it has been shown that compressive sensing achieves high secrecy.

The simultaneous encryption and lightweight operations using RSSI-based key generation, as proposed here, can be used in all RERUM components for single-hop wireless communication. For example, it can be deployed for data encryption and compression between the Resource Monitor and the Alert Processor, between the RERUM Devices and the Gateway, etc.

3.4.2 Compressed Sensing Background

The recently introduced theory of CS exploits the structure of a signal in order to enable a significant reduction in the sampling and computation costs. CS has been used in many research areas, like in wireless intrusion detection [DT13], energy-efficiency [FRAAT13], indoor localization [FNT12], etc. Assume that $\mathbf{x} \in \mathbb{R}^N$ refers to information collected by a sensor. According to CS theory, if $\mathbf{x}$ is sparse in some domain, it can be reconstructed exactly with high probability from M randomized linear projections of signal $\mathbf{x}$ into a measurement matrix $\mathbf{\Phi} \in \mathbb{R}^{M \times N}$, where $M \ll N$. A signal is called sparse if most of its elements are zero in a specific transform basis. Signal $\mathbf{x} \in \mathbb{R}^N$ can be expressed using a dictionary $\mathbf{\Psi}$ of $N \times 1$ vectors $\{\boldsymbol{\psi}_{i=1}^N\}$ such that $\mathbf{x} = \mathbf{\Psi b}$, where $\mathbf{b} \in \mathbb{R}^N$ is a sparse vector with S non-zero components ($\| \mathbf{b} \|_0 = S$). The general measurement model is expressed as follows:

$$\mathbf{y} = \mathbf{\Phi x} = \mathbf{\Phi \Psi b} = \mathbf{\Theta b} \quad (1)$$

where $\mathbf{\Theta} = \mathbf{\Phi \Psi}$. The original vector $\mathbf{b}$, and consequently the sparse signal $\mathbf{x}$ are estimated by solving the following ℓ_1-norm constrained optimization problem:

$$\hat{\mathbf{b}} = \arg\min \| \mathbf{b} \|_1 \quad s.t. \quad \mathbf{y} = \mathbf{\Theta b}. \quad (2)$$

Finally, the reconstructed signal is given by $\hat{\mathbf{x}} = \mathbf{\Psi} \hat{\mathbf{b}}$.

Observe in (1) that information $\mathbf{x}$ is multiplied by the measurement matrix $\mathbf{\Phi}$, producing signal $\mathbf{y}$ that is a compressed and encrypted version of $\mathbf{x}$. Consequently, CS unifies compression and encryption in a simple linear measurement step, using $\mathbf{\Phi}$ as the encryption key. But how secure is such an encryption scheme? Rachlin et al. [RB08] have demonstrated that although CS-based encryption does not achieve Shannon's definition for perfect secrecy, it can however provide a computational guarantee of secrecy. In [OAS08], the authors study the security implications of CS by considering brute force and structured attacks. The structured attacks refer to attacks based on the symmetry and sparsity structure of the measurement matrix $\mathbf{\Phi}$. Their results show that the computational complexity of these attacks renders them infeasible in practice.

Cryptography using CS is a symmetric operation as the same key $\mathbf{\Phi}$ is used for both encryption and decryption. Decryption is performed by solving the optimization problem shown in [CW08]. A key difference with other cryptographic methods is that CS does not require both communicating parties to hold exactly the same key. The reconstruction error of the CS-based encryption depends on the similarity of the keys owned by those parties.

3.4.3 Key Generation

Secret key establishment is a fundamental requirement as WSNs carry sensitive and private information over unattended environments. As sensors are severe resource constrained devices, energy efficient cryptographic algorithms are a necessity. An approach for key generation but without involving key distribution is to enable the interested communicating parties to extract cryptographic keys (in our case the matrix $\mathbf{\Phi}$) directly from the wireless channel. Following the notation used in similar works, we assume that there is a legitimate transmitter named Alice, a legitimate receiver with the name Bob, and an adversary named Eve; a passive eavesdropper. The distances between Alice-Bob, Alice-Eve, and Eve-Bob are denoted by d_1, d_2, and d_3, respectively (Figure 23).

Alice aims to extract key $\mathbf{\Phi_1}$ using channel measurements. Afterwards, she will use this key in order to encrypt plaintext $\mathbf{x}$, producing ciphertext $\mathbf{y}$ using (1), and finally, she will transmit $\mathbf{y}$ to Bob using an appropriate communication protocol. Bob, in order to decrypt $\mathbf{y}$, he needs key $\mathbf{\Phi_1}$. As there is no key distribution mechanism, he will attempt to extract the decryption key $\mathbf{\Phi_2}$ from the wireless channel, following the same operation as Alice. After key extraction, Bob will use (2) to decrypt ciphertext $\mathbf{y}$. As this is a symmetric encryption/decryption operation, the reconstruction error (the fidelity of the decrypted message) depends on the similarity between $\mathbf{\Phi_1}$ and $\mathbf{\Phi_2}$. The challenge here is to employ a mechanism that enables Bob and Alice to extract almost the same key from the wireless channel, while making Eve unable to extract such a key that will further allow her to decrypt the ciphertext with high fidelity.

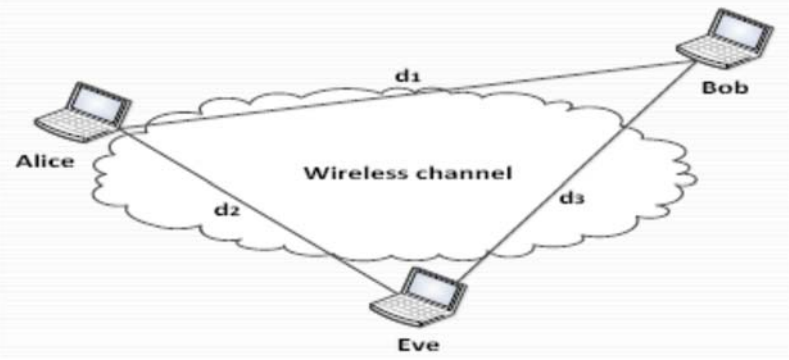


Figure 23: Key extraction from the wireless channel with the presence of an eavesdropper

3.4.4 RSSI Sampling

The wireless channel can be regarded as a time and space varying filter that any point in time has the same filter response for transmissions between two wireless nodes [PJC++K13]. The signal transmissions between Alice and Bob, and vice-versa, are modified by the channel in a manner that is unique between these two. The multipath properties between Alice and Bob (gains, phase shifts, and delays) are identical on both directions of the communication link. This is usually called as *channel reciprocity*. How fast these properties change depends on the *coherence time* (CT) of the network, defined as the minimum time channel properties are invariant. Based on these, Alice and Bob can extract a common key from the wireless channel by considering one or more channel properties at intervals greater than CT. A question that arises here is if Eve is able to extract the same (or almost the same) key from the channel. It is shown in [JAW74] that the received signal rapidly de-correlates over a distance of roughly half a wavelength, therefore if $d_2, d_3 > \frac{\lambda}{2}$ (Figure 23), where λ is the carrier frequency's wavelength, Eve will extract a completely different key.

RSSI is a popular statistic of the radio channel made available by the majority of the off-the-shelf wireless transceivers, and for this reason we use it in order to extract the secret key from the channel. We set up the experimental network shown in Figure 23 using three wireless Z1 sensors [SENSO] in a room with several objects, and people freely moving around. The distances between the nodes are as follows: $d_1 = 120cm$:, $d_2 = 100cm$, and $d_3 = 100cm$. For transmission we use the IEEE 802.15.4 MAC at the 2.4 GHz. At this frequency, $\frac{\lambda}{2} = 12,5cm$, so Eve is at a distance that will not hopefully allow her to extract the same key as Bob and Alice.

For RSSI collection, Alice periodically transmits UDP packets to Bob. As soon as Bob receives a packet from Alice, it records the corresponding RSSI value. At the same time, when Alice receives the ACK packet from Bob, acknowledging a correct packet reception, she records the RSSI value for that packet.

3.4.4.1 Quantization

The raw RSSI measurements cannot be directly used for key extraction. Firstly, we filter out of the collected RSSI its mean value, as RSSI is a predictable function of the distance, so an adversary that knows Alice and Bob's positions could estimate their recorded RSSI values. Next, we split RSSI into several blocks of length B_l and then we quantize its values per block. A quantization function *Q*(.) takes as input an RSSI vector of length B_l and produces a binary stream. Supposing $RSSI_{B_l}$ is the vector that contains the RSSI values, function *Q*(.) first computes $p = \max RSSI_{B_l} - \min RSSI_{B_l}$. Then, given $n \in \mathbb{R}^+$ the number of bits used for quantization, we split the recorded $RSSI_{B_l}$ into $\frac{2^n}{p}$ equally-sized regions, and for each region we assign an n-bit binary word. For example, if $RSSI_{B_l} = \{20,22,23,24\}$ and n=2 bits, then *Q*(.) gives {(00), (01), (10), (11)}. The total number of bits after the quantization process equals $B_l \times n$. Furthermore, we encode *Q*(.)'s output using the Gray code [FNT12] as this type of encoding ensures that small RSSI discrepancies cause no more than a single bit error. This is important as the possible key mismatch between Alice and Bob has to be minimized.

3.4.4.2 Key Uniformity

As mentioned previously, the measurement matrix $\mathbf{\Phi}$ is used as the encryption/decryption key in our CS-based scheme. The performance of CS (in terms of the reconstruction error) heavily depends on two factors: (i) sparsity, and (ii) coherence. Sparsity gives a measure of how many non-zeros of $\mathbf{x}$ exist when projected into dictionary $\mathbf{\Psi}$. On the other hand, coherence measures the largest correlation between any two elements of $\mathbf{\Phi}$ and $\mathbf{\Psi}$. The smaller the coherence, the higher CS performance is. Several works have shown (e.g.) that when considering measurement matrices built using values selected independently from certain distributions, exact signal recovery can be achieved with high probability. One such choice is the uniform distribution used in several works (e.g. [FRAAT13]).

As uniformity gives higher CS performance, we hash *Q*(.)'s output with an appropriate secure hashing function, targeting to achieve uniformity for key $\mathbf{\Phi}$. We utilize *Uquark* [AHP13], a lightweight hash algorithm which takes as input a 64-byte block and produces a 17-byte hashed output. This algorithm has a very low number of collisions; hence, high probability of achieving uniformity.

3.4.4.3 Information reconciliation

Both Alice and Bob follow the previously described steps and have now created their own keys: Key_A, and Key_B, respectively. As Eve is fully aware of the key generation algorithm, she has created Key_E. Actually, taking into account that RSSI is initially split into equally-sized blocks of length B_l, these keys are supersets of one key per block, e.g. with $i \in [1,k]$. The same holds for Bob and Eve. There are two challenges here: (i) to find the appropriate block id such as $\mathrm{key}_{\mathrm{A_i}}$ is very close to $\mathrm{key}_{\mathrm{B_i}}$, while $\mathrm{key}_{\mathrm{E}}$ is as different as possible. In fact, we are seeking for a mechanism that will enable Alice and Bob to agree on the appropriate block id, without exchanging private information that could be captured by Eve.

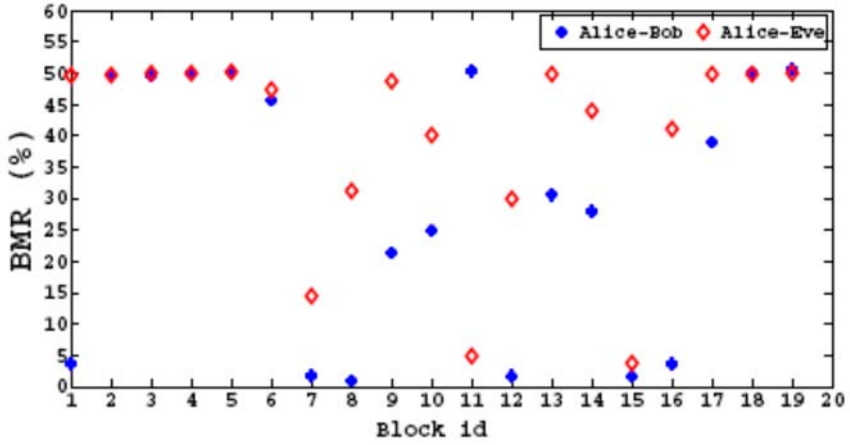


Figure 24: Bit mismatch rate between the keys of Alice-Bob and Alice-Eve

We run a number of experiments using the topology shown in Figure 23. Alice transmits a single UDP packet to Bob every 1 msec. Based on this traffic, RSSI samples are collected by all nodes, and the corresponding keys are derived using a 3-bit quantization. Figure 24 shows the bit-mismatch-rate (BMR) between $\mathrm{Key}_{\mathrm{A}}$ and Key_B, and $\mathrm{Key}_{\mathrm{A}}$ and $\mathrm{Key}_{\mathrm{E}}$, for the different blocks. BMR gives the number of bits that are found to be different when comparing two keys at the bit level. BMR can significantly vary between the blocks because measurements are taken in a dynamic environment where multipath fading causes significant RSSI fluctuations.

Observe that Alice and Bob achieve a low BMR ($\approx 0.9\%$) for the block id=8. BMR cannot be zero as there are RSSI variations due to noise, interference, and hardware limitations. For the same block id, the BMR for $\mathrm{Key}_{\mathrm{A}}$ and $\mathrm{Key}_{\mathrm{E}}$ is over 30%. Nevertheless, as Alice does not know $\mathrm{Key}_{\mathrm{B}}$, she cannot estimate BMR, so she is not able to select the appropriate block id. Moreover, Alice and Bob cannot exchange keys as Eve is able to capture them.

In order to address those limitations, we propose the use of a cryptographic primitive called as *secure sketch* (SeS). Using a SeS, Bob can produce a public information $\mathrm{SeS}_{\mathrm{B}}$ about its key $\mathrm{Key}_{\mathrm{B}}$ that however does not reveal $\mathrm{Key}_{\mathrm{B}}$. Alice, upon receiving $\mathrm{SeS}_{\mathrm{B}}$ from Bob, can extract $\mathrm{Key}_{\mathrm{B}}$ using $\mathrm{SeS}_{\mathrm{B}}$ and her own key ($\mathrm{Key}_{\mathrm{A}}$). Exact recovery is possible if the BMR between $\mathrm{Key}_{\mathrm{B}}$ and $\mathrm{Key}_{\mathrm{A}}$ is not too high. This process is often called as *information reconciliation*. For generating the SeSs we use the algorithm proposed in [DORS08]. Note that Bob computes the SeSs that are further transmitted to Alice. Alice then performs information reconciliation extracting $\mathrm{Key}_{\mathrm{B}}$, and now she is able to compute BMR and find the block id where it takes its minimum value. Finally, it transmits in clear that block id to Bob, so he knows which key to use for decryption. On the other hand, Eve is able to capture both the SeS and the optimum block id. However, as her key significantly differs from Bob's key, she will not be able to accurately compute $\mathrm{Key}_{\mathrm{B}}$ using the SeS. Furthermore, as she does not hold the correct key $\mathrm{Key}_{\mathrm{B}}$, the capture of the optimum block id does not provide any useful information to her.

The algorithm shown below summarises the steps for key generation that both Alice and Bob follow, except the *Transmit_Sketch* function that is executed by only one of them. So Bob transmits his SeSs to Alice, then Alice performs the information reconciliation process, and sends back to Bob the optimum block id.

Key generation algorithm:

Variables:

1) *RSSI*: the RSSI samples over a time window
2) *bl*: the length of the equally-sized blocks that comprise RSSI
3) *L*: the number of the equally-sized blocks that comprise RSSI
4) $RSSI_{bl_i}$: the RSSI values that belong to block *i*
5) *n*: the number of bits per quantisation
6) Q_i: the quantised values of block *i*
7) H_i: the hashed values of block *i*, after quantisation
8) SeS_i: the secure sketches of block *i*

Functions:

1) *Quantise_RSSI*: the function that quantises a block of RSSI samples , given the number of bits
2) *Create_Hash*: the function that hashes the quantised RSSI block
3) *Create_Sketch*: the function that computes the secure sketches
4) *Transmit_SKetch*: the function that transmits the secure sketches to the node that will perform the information reconciliation task

Algorithm:

for i=1 to i=L do

Q_i=*Quantise_RSSI(*$RSSI_{bl_i}$, *n)*

H_i=*Create_Hash(*Q_i)

SeS_i=Create_Sketch(H_i)

Transmit_Sketch(SeS_i)

end for

3.4.5 Performance evaluation

In order to evaluate the proposed key generation algorithm, we consider collected RSSI samples using the topology shown in Figure 23 Key extraction from the wireless channel with the presence of an eavesdropper. Also, we make the following assumptions about Eve: (i) she can overhear all the packets exchanged between Alice and Bob, (ii) she is fully aware of the key generation algorithm and its configuration parameters, (iii) she does not interfere with Bob or Alice, and (iv) she does not masquerade as Bob or Alice.

We proceed with the evaluation by considering the following scenario. Alice has collected a number of ambient temperature measurements that wishes to encrypt and transmit to Bob. Actually, we feed Alice with the ambient temperature measurements provided by a real WSN [SENSO]. As CS also performs compression, we select the compression ratio as 50%, so $\frac{M}{N} = 0.5$. Next, both Alice and Bob perform the key generation algorithm, deriving $\mathrm{Key_A}$ and $\mathrm{Key_B}$, respectively. While Eve is overhearing the whole communication, she generates $\mathrm{Key_E}$.

We compute the reconstruction error of the collected measurements, defined as, where $\mathbf{x}$ and $\hat{\mathbf{x}}$ are the original and reconstructed temperature measurements, respectively. During the evaluation, Alice

encrypts data with $\mathrm{Key_A}$, while Bob and Eve decrypt them with $\mathrm{Key_B}$, and $\mathrm{Key_E}$, respectively. The decryption process is often referred as reconstruction in CS terminology. For decryption we use the OMP algorithm [TG07]as it is computationally very efficient. The length of the equally-sized RSSI blocks is set to 25000 (line 3 in Algorithm 1). As RSSI sampling is performed every 1 msec, this results to a key generation every 25 secs that is an adequate time for key refreshment.

Figure 25 shows the cumulative density function (CDF) of the reconstruction error when information is decrypted by Bob, for an increasing number of quantization bits (line 6 in Algorithm 1). Observe that as the number of bits increases, the error also increases. This is because BMR increases for the keys of Alice and Bob when the bit length of the keys increases (Figure 26). However, the error even for the 4-bit quantization is less than 0.05 for the 80% of the encrypted information (temperature measurements are split into blocks and encrypted separately). For a 3-bit quantization, the error is less than 0.05 for more than the 90% of the encrypted blocks.

When Eve decrypts information using $\mathrm{Key_E}$, the reconstruction error is extremely high as shown in Figure 27 (note that this figure has a different x-axis scale). Even for the 1-bit quantization, Eve experiences an error of more than 0.6, meaning that the decrypted data differ by more than 60% with the original data, thus making Eve unable to steal potential sensitive information.

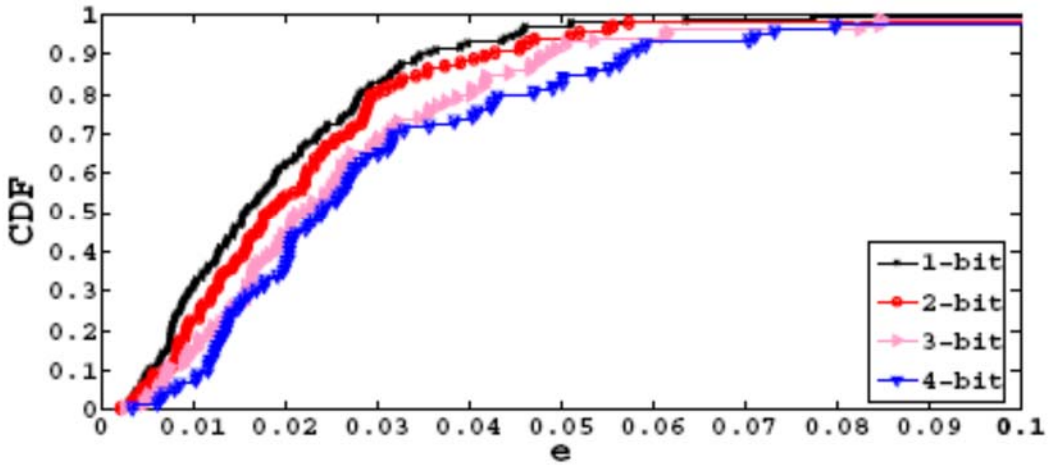


Figure 25: Reconstruction error at Bob or Alice for different quantization levels

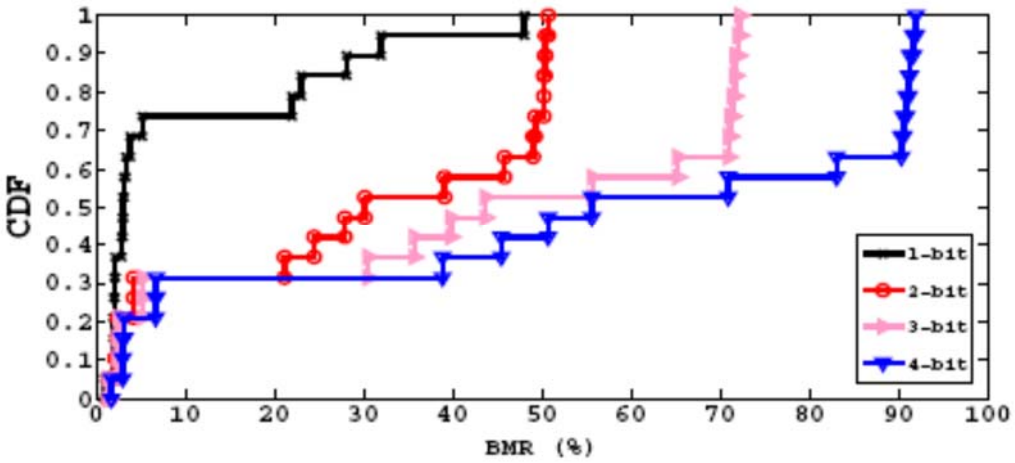


Figure 26: Bit mismatch rate for different quantization levels

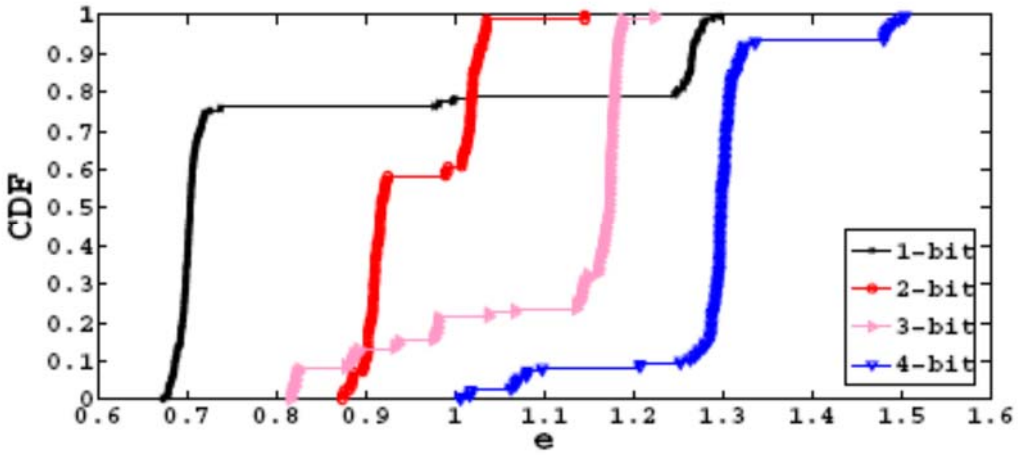


Figure 27: Reconstruction error at Eve for different quantization levels

3.4.6 Summary of profile

Layer	Application
Security Association provided between	RERUM Device to RERUM Device – Hop-by-Hop (e.g. RD1-to-RD3 via RD2 in Figure 2)
Provides Confidentiality	Yes
Provides Integrity	No
Provides Origin Authentication	No
End-2-End	No
Hop-2-Hop	Yes
Needed Key Material	CS-based key material

4 Authorization in RERUM

4.1 Introduction, Motivation and Link to User Requirements

This chapter explains the mechanisms for authorizing requests in RERUM. That is, deciding whether to allow or reject a request to a RERUM service or resource. Authorization may be used for enforcing both security and privacy requirements. As this document is focused on security rather than privacy, this chapter is focused on the authorization mechanisms for security only. Note that many of the said authorization mechanisms will be reused to protect privacy as well. The differences of how these authentication mechanisms are used for such protection will be provided in Deliverable D3.2.

The components explained in this chapter correspond with the authorization component stated in D2.3 as shown in Figure 28 extracted from it:

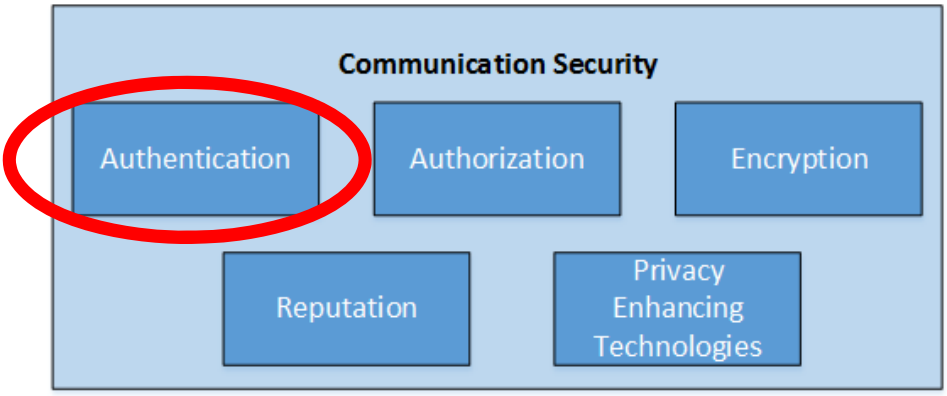


Figure 28: Security components in the communication layer from D2.3

Specifically, in this chapter we explain the design of an Attribute Based Access Control (ABAC) mechanism and its integration with RERUM, as stated in the table of innovations of RERUM in D2.1, i.e., for Contribution 10: Integration of ABAC in IoT with support for specific business data contained in the request'. This contribution is linked to the following user requirement:

- UR-12: The user requires being able to define specific access criteria so that the system can make decisions based on the attributes of the user that is issuing the request, and the context of the request.

One of the main concerns that is hindering the adoption of IoT is the lack of trust in this technology due to insufficient security support. Access control is an essential security feature to control who is accessing to the resources and services of the system. The ability to take access decisions based on any attribute not only from the user, but also from the request, is the most flexible and powerful mechanism of above-mentioned ABAC mechanism, because it is the one that allows considering the most information available in the process of controlling the access. The need of configuring the access to the system is universal in IoT and therefore is applicable to all use cases in the project and will be validated in all trials.

Besides, RERUM also addresses an issue that is not often addressed by generic platforms. Sometimes it is necessary to define access criteria based on information that is completely specific from the service that is being invoked. For example in UC-O1: The administrator might decide to enable certain service only for user's who hold the attribute 'police', which is only provided for user that are to members of the local police. A regular citizen, i.e., Alice, making an attempt to get information about the position of a citizen's car will be rejected, because only policemen are allowed to get that information, i.e.,

users whose type equals to 'police'. Instead, Alice is just allowed to get the information of an information post, whose type is 'public'. For doing this kind of check, it is necessary to know in advance the characteristics of the service invoked and all the parameters it is receiving. Though there are already some technologies, such as XACML[47] that allow defining such criteria. The main issue is that they often need to specify directly in the code how to access the specific parameters that are used to take a decision. Considering that each service is different from the others, this makes generic platforms unable to deal with specific criteria. In contrast, RERUM is able to cope with this problem while still keeping the platform generic. This gives RERUM a flexibility that other platforms lack, because it allows adding new services that need specific access criteria without having to modify the code of the platform.

This chapter is organized in the following parts:

- authentication in RERUM, where an explanation of the relationship and differences between authentication and authorization is provided and how RERUM requires the authentication to be carried out,
- analysis of authorization options, where a general analysis of the main trends in authorization available as well as their pros and cons in RERUM context is described, and
- explanation of the design of authorization components where the technical details of how the authorization components defined in the architecture document are designed in RERUM.

4.2 Service level authentication in RERUM

First of all, authentication is not the same as authorization. In general terms, authentication is a process of making sure that any entity accessing the system is he who claims to be. On the other hand authorization is a process of deciding whether to grant access or not to a concrete resource to that concrete entity, possibly based on the attributes of that entity. That is, authentication is not the same and is even not the part of the authorization, but the authorization heavily relies on the authentication.

When the authentication is carried out to prove that a device accessing any system corresponds to a device already known by the system addressed by said device, we name it authentication at device level. This was previously discussed in Sections 2.2.2 and 2.3.4.

When the authentication is carried out that the request is being made on behalf of a user that has previously defined in the system, it is named authorization at service level.

In the concrete case of RERUM, service level authentication means that any application accessing the system is acting on behalf of the human user that it is claiming to act on behalf of. And for the authorization it means deciding whether to grant or reject access to RERUM services and resources to that application based on the attributes of the human user that the application is acting on behalf of.

As explained in D2.3, service level authentication is not part of RERUM, authentication is not part of RERUM, and therefore it is not in the scope of RERUM to provide any design or implementation of an authentication system. But authorization is indeed in the scope of RERUM, and RERUM performs this authorization based, among other things, on the attributes of the requesters that try to access RERUM resources or services. However, authentication can be carried out in different ways, and the way to access these attributes can be carried out in different ways as well. For this reason, it is very important for RERUM to define how it expects this authentication to be carried out and how these attributes are provided or otherwise the RERUM authorization components might try to access these attributes on a way that were different from the one provided by the Authentication. This section does precisely that:

[47] XACML stands for eXtended Access Control Markup Language and is a standard language for defining access control criteria from the OASIS standardization body.

it specifies the minimum RERUM needs of how the authentication should be carried out and how the attributes of the user should be provided to RERUM.

Authorization is normally carried out based on the attributes of the entity issuing a concrete request. In the concrete case of RERUM, the originator of a request may be either human users or applications acting on behalf of a human user, which means that the device issuing the request should be able to authenticate itself on behalf of such user. In any case, it is necessary that the authentication layer provides an unified interface to the authorization components to access the attributes of the requester, let it be a human being or a service installed on a device acting on behalf of a human being.

The idea is that any access to the system must be carried out on behalf of a valid user previously registered in the system, even an anonymous user, in order to let the security policies to make decisions based on the identity of such user. The security policies will then be able to make decisions based on the attributes of the user that is issuing the request, even for anonymous users.

RERUM proposes that any device trying to access the system must provide first a security token obtained from an identity platform when logging in it. The identity platform might not necessarily be part of the RERUM system. This security token will have to be associated with a user registered in the platform and signed by the identity platform to ensure it is not tampered.

RERUM proposal envisages three types of access:

(a) Access from unknown devices external to the RERUM installation that do not have any specific user associated: the only way these devices will be able to access the RERUM system will be by using an anonymous user. This anonymous user will have its operations restricted to only those that the system administrator has decided to be open publicly. For instance, the administrator of the municipality might decide to let citizens to have access to the measures of pollution using this user;
(b) Access from VRD belonging to the RERUM system. These devices will work with special internal users of RERUM. As these devices have been previously incorporated to the RERUM installation by the administrator, they are considered to be trusted, and their permissions should be set accordingly
(c) Access from VRDs that have a valid user registered in the system different from the anonymous and internal ones. Note the administrator of RERUM will be one of these users as well, with the only difference that it will be granted access to all administration tasks in RERUM

In all these cases, there must be a user registered in the platform to be able to log in it. The registration process has the following operations associated in the same order they are numbered in the list:

1. A requesting device asks for a new user and provides the necessary means to authenticate itself from the ones that the identity platform API supports. These means might be, for instance, a user and password or, more commonly, a valid user certificate signed by a CA. The requesting device may also provide additional information about itself or the human being that it is acting on behalf of;
2. The identity platform associates the new user with the means provided by the requesting device and performs a partial and provisional assignment of the values of the attributes of the new user based on the data provided by the requesting device and their internal criteria and assign it an status of ‘pending to be confirmed;
3. The identity platform stores the new user and notifies its administrator of a new user pending to be confirmed and

4. The administrator checks the provisional attributes set to the new user and fills in the values for the remaining ones. These attributes will be utilized later for granting access to the RERUM service. Finally the administrator sets the state of the new user to 'valid' and saves it.

Figure 29 shows the complete registering sequence:

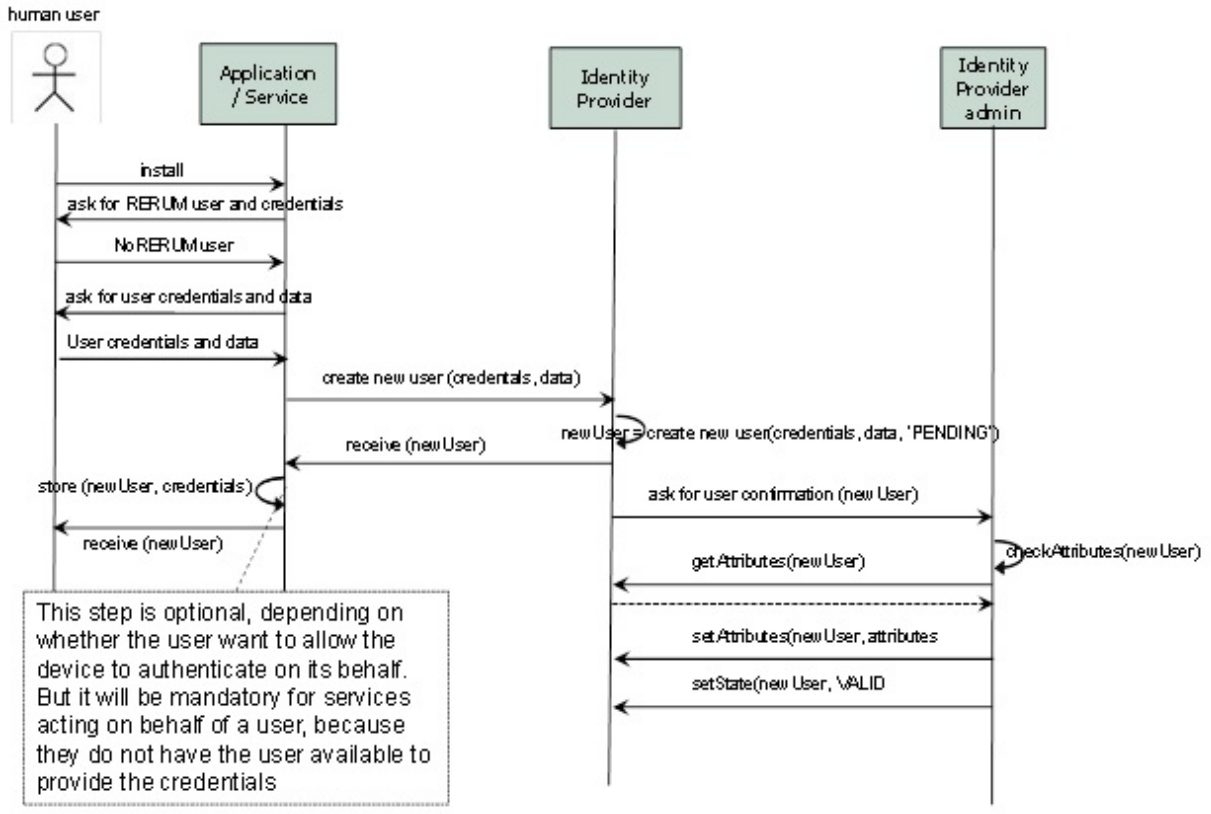


Figure 29: Registering users

According to the three types of access stated above, it will be necessary to perform the following tasks.

In case the system supports access (a) it will be necessary to create an anonymous user from any requesting device and have the administrator of the identity platform to fill their values the way he see fit. Once this user is created, external unknown devices will be able to log in the system using this user.

In case of (b) it will be necessary to create at least one internal user for RERUM Devices and have the administrator of the identity platform to fill their values the way he see fit. Note, however, though these users might be perfectly a single one, this is not necessarily true. For instance, a VRD of the indoor scenarios may possible be more trusted than the ones from the outdoor scenarios because they could be more difficult to be attacked. Hence, it can make sense to have different internal users with different permissions for these different kinds of VRDs. These devices will work always using these special users, so they are capable of accessing other devices of the same RERUM system For this reason and to enhance performance, it is reasonable that the installation process for these devices gets a security token that never expires for them, possibly from an internal RERUM IdP (they are special non-public users)

Also note that depending on whether there is a negotiation phase with the authorization layer for asking the attributes of the user, the VRD running the authorization layer will require an internal user as well.

In case (c) any device trying to access the system will have to have a user previously registered. Hence, if it does not have any user registered yet, it will have to ask for a registration by a user using the following procedure.

Once the users have been created in the system, the responsibilities of the involved components are:

- Identity platform needs to:
  - check the authenticity of the credentials of the user when logging into the system;
  - include a security token in a header of the http session that includes at least a reference for the user and sign this token;
  - provide an API for the authorization layer to negotiate the values required for the authorization. This negotiation might be as simple as the authorization layer asking for the each needed attributes every time. Alternatively, the identity may include all the attributes of the user in the token at once;
  - assign an expiration hour for the token possibly depending on the attributes of the user, and
  - sign the token to prevent it from being tampered later.
- Requesting VRDs need to:
  - obtain a valid security token from the Identity Provider. For access in case (b) and (c) this will be performed when trying to access the system. However, for components meant to be permanently connected to the system, such as the VRDs registered in the system, it is conceivable to provide them directly with the security token obtained at the time of registration of their internal users, and
  - provide this security token in a header of the http request.
- The authorization layer needs to:
  - retrieve the security token from the header of the http request;
  - check the integrity of the security token;
  - check the expiration date of the security token;
  - negotiate the needed fields with the Identity Provider, and
  - fulfill all other responsibilities that are inherent to the authorization operation, including, of course, granting or rejecting access to the request based, among other things, on the attributes of the user corresponding to the one referenced in the security token

Figure 30 shows the process for retrieving the security token for a service or for any device that acts on behalf of a user without requiring him to authenticate each time.

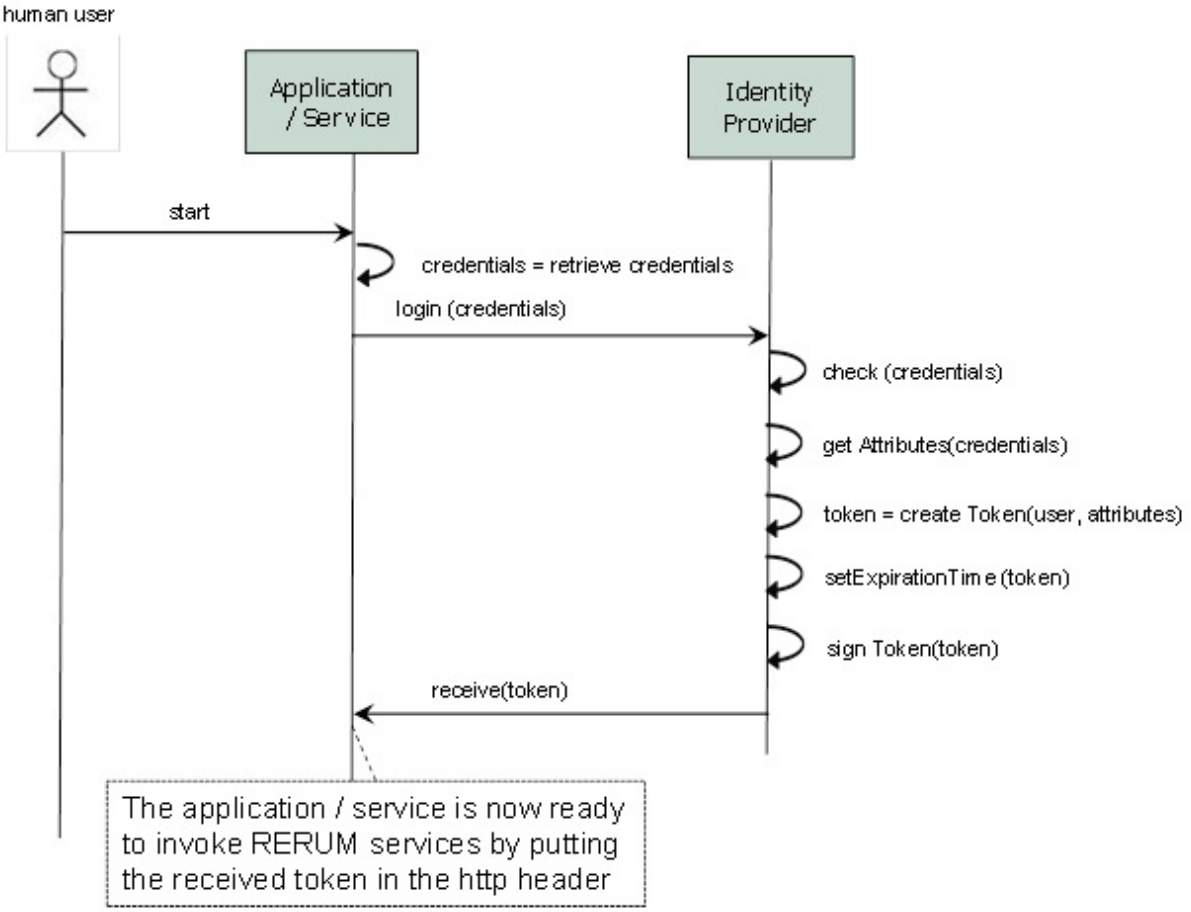


Figure 30: Retrieving a security token for later use

Figure 31 shows the process for retrieving the security token for an application that does not stores any user credentials and must ask the user for them each time:

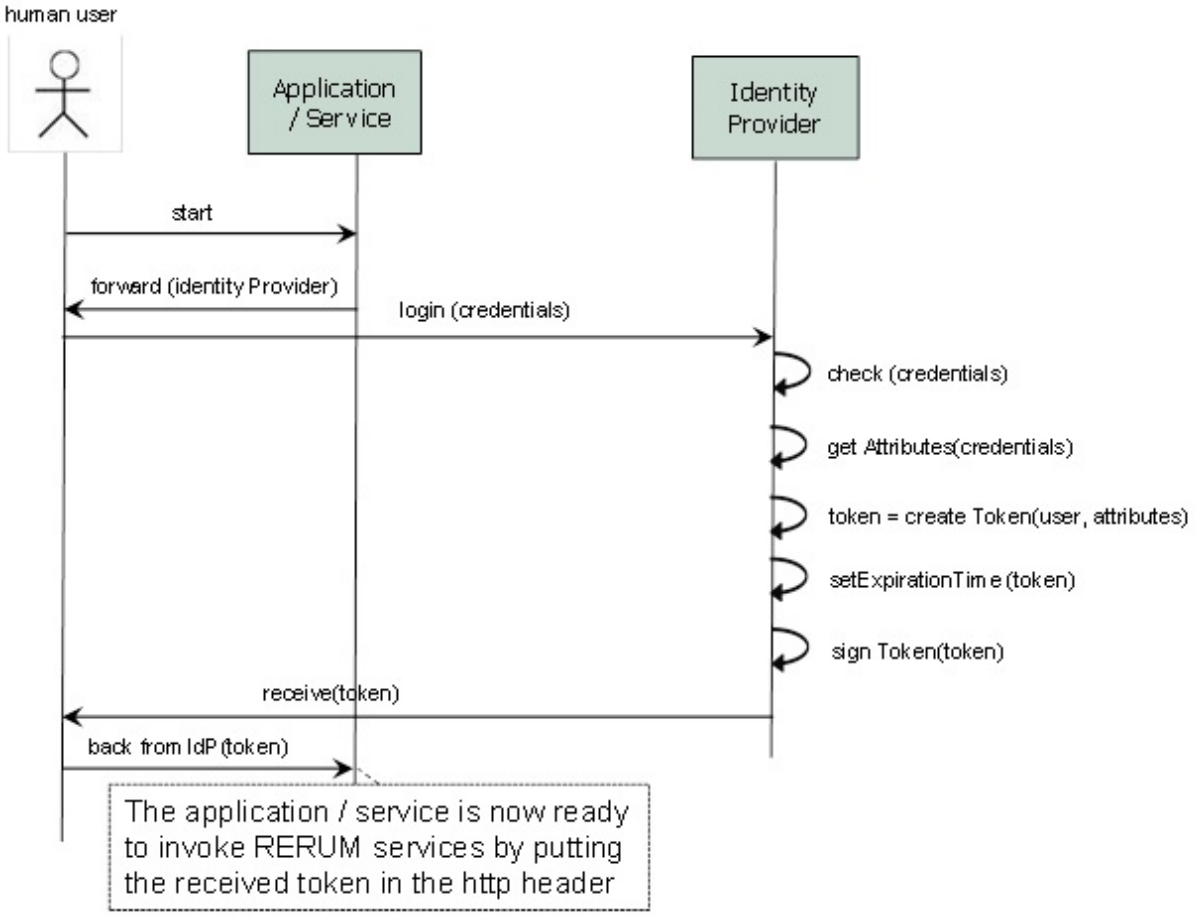


Figure 31: Authenticating users on each startup of the application

Basically, the authorization layer could do the forwarding for the application, but this way the application would have a better control of the execution flow.

Figure 32 shows a simplified view of how the security token is delivered to the authorization layer and is used by it after the application / service has already got the security token:

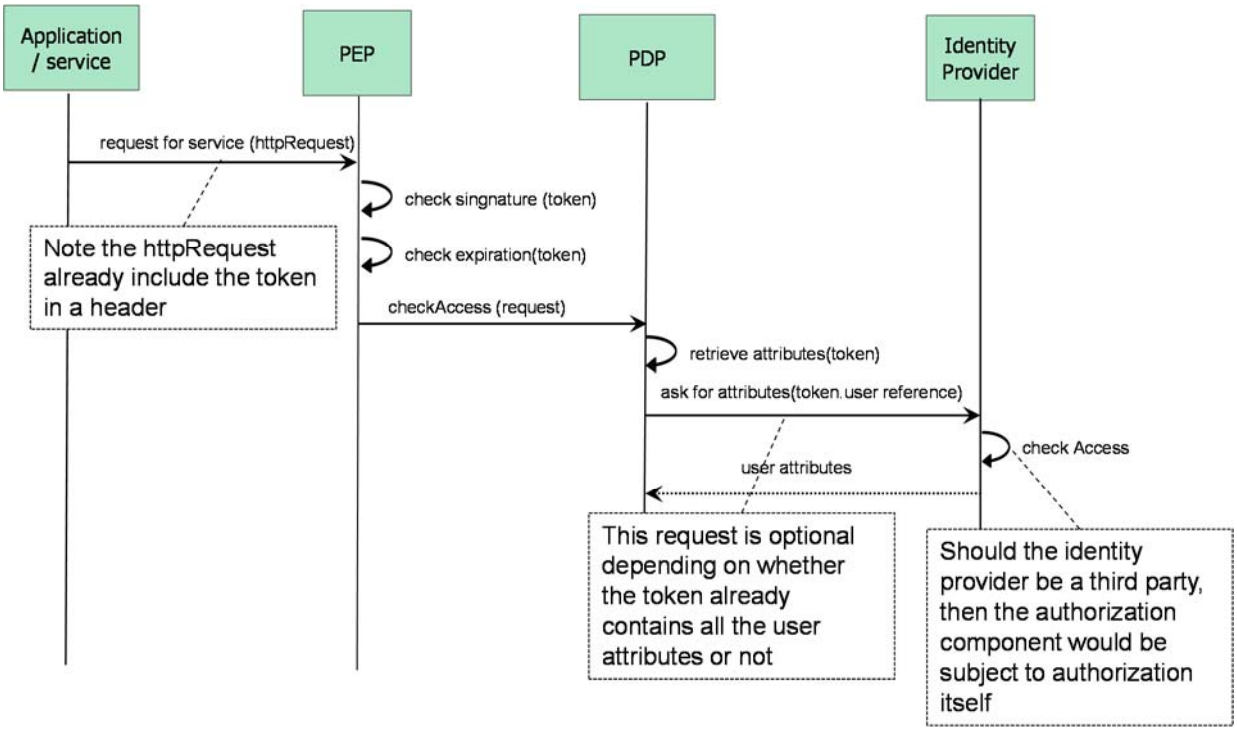


Figure 32: Reusing a security token already retrieved

Example 1: May the administrator's name be Juan. Juan installs the RERUM traffic application and starts it. The installation process did not install the user credentials of Juan. When the RERUM app initially communicates with the authentication layer in the GW instance, the authentication layer cannot locate the security token in the http header. Hence, it responds with an unauthorized code so the application can redirect to a login page of the identity platform where the user authenticates him with the credentials and user obtained in the registration process. The identity platform then fills the http header with the content of Juan's attributes and signs it. Now the authorization layer detects the availability of a token. The token contains Juan's attributes and the authorization layer can check the signature to ensure the token is valid, non-obsolete and generated by a trusted entity.

Example 2: A RD needs to access another one. The RD has previously received a security token from the identity platform for user 'internal' using, whatever authentication mechanism the application designer might see fit for internal RDs. Alternatively, the RD may have been provided with a security token during some startup-phase from the middleware. When the RD needs to access another one, it includes his own token in the request. This token can then be used as any other in the authorization layer.

Example 3: A RERUM Device from the electricity operator (EO) wants to access the electricity counter of my home. For this purpose, the EO has already received his own user from the certificate of the server that will be used to authenticate the EO server. Now when the EO server wants to access the RERUM system, it must get his authentication token associated to his user by authenticating itself by providing the certificate corresponding to the user associated with the municipality. The identity platform will generate the security token with the information of the associated user (that was previously filled from the SO server), sign it, and include it in the http request. Now the SO server can access the RERUM system with this token and the authorization layer will be able to read the attributes of the SO server and check the signature of the security token.

Example 4: An unknown RERUM Device wants to access the system. In this case the unknown RERUM Device can only try to access the system by authenticating itself in the identity platform as 'anonymous'. Previously, the administrator will have to define the allowed actions for user with attributes that correspond to an anonymous user. When such a request arrives at the authentication

layer the request will be evaluated according to the rules. Hence, an anonymous user can do only actions that have been previously defined by the administrator.

Note: A valid digital certificate should be deployed in every GW instance to let the connecting RD to check it.

4.3 Analysis of authorization options

This section analyses the three most widely used options for authorization and its suitability for RERUM. With this purpose, we will first make a brief introduction about how they work and then explain whether they fit or not in RERUM. This analysis aims to be easy to understand, and for that reason it tries to avoid complex technical details and keep at a conceptual level as far as possible. Complete technical details will be exposed later in Section 4.4 for the chosen option.

Authorization is not at all the same as authentication. Authentication is the action of checking that any requester is he that claims to be, while authorization is the action to decide whether or not to grant access to a given service or resource. Although authentication is an essential prerequisite for authorization, the goal of this section is not discussing authentication but it assumes that it has been achieved instead. RERUM's proposal for authentication is explained in detail in Section 4.2. Therefore, this analysis explains authentication processes only when they refer to non-RERUM procedures that are tied to the different authorization options explained.

Note that this analysis covers several authorization mechanisms that already exist outside RERUM. For that reason, it needs to refer some concepts that are not specific from RERUM but whose names are the same as the ones used for RERUM. As such, and only for this analysis, these terms must be interpreted in a general way, not necessarily constrained to a RERUM scope. For helping to differentiate these concepts from the ones used in RERUM throughout the rest of documents, they are presented in *Italic font.* These concepts are:

Application: Any piece of Software capable of processing data, either directly or by invoking other pieces of Software, which may not be necessarily part of it.

Resource: Any functionality or piece of data that can be accessed by an *application*

Service: Any running *application* that exposes *resources*. Hence, note that a *service* is a special kind of *application* itself, but an *application* is not necessarily a service as it may not expose any resource to other applications.

GUI (Graphical User interface): A special kind of software specialized only in interacting with human users to present and obtain information from them. Though, basically, nothing impedes a *GUI* to process data on its own, this is not part of its definition. Instead, pure *GUI*s are supposed not to make any process on data themselves, but delegating this process to service. Hence, such pure *GUI* are not considered to be applications.

4.3.1 Option 1: Ad-hoc authorization provided by the application when registering in the system

The authorization process relies in proper authentication of the requester. This authentication may be carried out for each operation of once for all the session. As the authentication is subject to be a heavy task, both in computer and human effort, the approach usually taken is usually the latter. That is, the requester provides her credentials at the start of a session and gets authorized for the rest of the session or until this authorization expires. The process of providing the credentials to get authenticated and gain access to the system is named 'logging in the system'.

This option is the oldest and simplest one, which comes even from before the client – server paradigm. It basically consists on the addressed *application* checking all the permissions of the logged user each time the user starts it. This usually involves retrieving these permissions from a storage associated to the logged user, such as a database query for the user and making these permissions available from the *service*, usually in a variable stored in memory. From this moment on, the *application* may simply disable the proper actions to make them not accessible or preferably making the *services* checking the variable that contains the permissions previously retrieved.

This option has the advantage of letting the authorization process to be as complex and powerful as needed, because it has access to all the information treated by the *application*, and it is also very fast because the permissions are calculated only once per time the user logs in the *application*.

Though the ad-hoc authorization suits with client-server models, it couples tightly the authorization process with the *application* it is protecting. That is, the SW in charge of the authorization needs to have a deep knowledge in advance of the *Application* it is protecting and the way it stores and access its data. Besides, modern Internet applications usually decouple *services* from *GUI*s, running *GUI*s on the device utilized by the human user and *services* often run on different machines, bringing the need to transmit the permissions and their integrity. Moreover, the segregation between *GUI*s and *services* also make possible the need to split the authorization decision between the different *services* accessed by the *GUI*.

Finally, in modern *applications* where *services* and *GUI*s are split from each other, it is possible that a single *service* has to serve several *GUI*s and / or other *services*, and they may not necessarily need to access the same functionalities of the *service* and therefore they do not necessarily need the decision for the same set of operations available.

4.3.2 Option 2: OAuth

OAuth copes with the coupling problem of the ad-hoc procedure by letting any server produce a standard token with a set of permissions that can be reused among different *applications*. For this reason, and because its authorization times are still very fast, It is currently an authorization technology very widely used in Internet. Besides, it is the authorization counterpart for a very popular authentication mechanism, which is OpenId [OpId12].

In short, OAuth provides the requesting *applications* a security token that already contains the result of the authorization evaluation for all *resources* to be accessed in the system.

In short, OAuth consists in the authorization being resolved for the *resources* to be accessed in the *system* and being able of reusing this authorization later. This is carried out in the following way:

When an *application* tries to access any URL / operation on behalf of a user, it must get an OAuth security token[48] for that user / *application*. If the *application* does not have it, then it needs to ask it to an authorization server, which will redirect the request to the Identity Provider so it can check the authenticity of the user and decide on whether to provide the needed attributes for the authorization or not. If the user gets authenticated, then the authorization server generates an access token for the id of the user that is logged on. The access token contains a given expiration date. This access token will contain the permissions associated to the requested *service* plus the information needed for the service to check the identity of the user. From now on, the *application* may store internally and use it repeatedly to access the *service* till the validity of its ticket expires. Though it is not mandatory to reuse this ticket, the process to obtain it involves authenticating the user / *application* each time the security

[48] Actually, the technical name used in OAuth for the security token is 'access token', but we prefer the term 'security token' for keeping consistence with the rest of the document, which uses a similar concept with the name 'security token'

token is requested. And the authentication operation may not be necessarily a lightweight operation. For that reason, OAuth makes special sense when the *application* asks for a security token for all the needed *resources* to be accessed at once and it reuses this token later once and again.

When an *application* tries to access any URL of a system on behalf of a user, It must provide the access token and the system must first check first the validity of the token and whether it grants access to the requested URL. After this, the access to the *resource* is granted according to the permission in it. If the security token did not exist the system returns an http code 403 meaning that it is necessary to authenticate the user first because the security token also includes the credentials to access the system. If the security token did not include permission for the requested URL, then the access is rejected.

Although it is theoretically possible to obtain a security token accessing any content of the request, in order to do so, the Application would have to ask for the security token for each request because the concrete values of the request will not be known until the request is issued. For example: A video store would like to check the age of the user before selling a video that is rated adult-only. The store would have to access both the age of the user and the rating of the requested video. Their values will be only available on the request itself. Hence, for checking this it would be necessary to create a security token each time a new request for a video with age restriction arrives.

This is feasible, but it removes the main advantage of OAuth, which consists in authorizing once and accessing multiple times, while it still keeps its main drawback, which is the need to authenticate the user / *application* each time the security token is issued. That is, it does not make much sense to use OAuth asking for a new security token per each request, because there are other technologies that allow doing it without having to authenticate the requests each time. For that reason the rest of this analysis is aimed only to the case where the same security token is reused for several requests.

Under this assumption, OAuth may suit some Internet *applications* provided that:

- The set of available URL / operations to be granted is previously known and not very large;
- The access to them can be easily granted basing only on the attributes provided by the identity platform and
- The complexity of the *application* itself is small enough to hold all the authorization logic in a single point

If the *application* covers these conditions, then OAuth has the advantage of being quite fast and lightweight, because the authorization decisions are evaluated only once for all the *application*, with no need to execute an authorization process each time a URL is accessed.

But OAuth is not suitable for RERUM because:

- OAuth does not support checking the purpose of the request or the subject being accessed due to this can only be known when request is issued;
- For the same reason, OAuth has no mean to base the access decision on any information coming from the request, which makes almost impossible to use business specific logic, because non-trivial specific logic for any *service* or *resource* will depend on arguments included in the request. Once OAuth has evaluated the access to a given URL, it has no mean to check the privacy of the resulting data, because this result is not available unitl after the processing of the request. For instance, if a query returns a set of records that each of them needs to be checked, there is no way that OAuth provides this; and
- Finally, though theoretically the OAuth specification does not impose any kind of authentication provider, in practice, almost all OAuth implementations only work with OpenId Authentication, which is a specification for an Authentication protocol. Although it is followed by many of the most important authentication providers, it is not universally accepted yet. As

a consequence, using OAuth in RERUM would in practice remove the RERUM flexibility to work with any authentication provider that complies with the authentication procedures defined in Section 4.2.

4.3.3 Option 3: Policy based access proxy

This option is not necessarily tied to a concrete technology, but to a general concept instead. In this option, the decision of granting access to a given URL / operation is taken each time the operation is requested and based on:

- some security criteria previously defined to be applicable to that operation requested,
- the identity of the user issuing the request, including the attributes of the user,
- information contained in the request itself.

These security criteria are normally defined in a formal language specific for defining access criteria and stored in a file named policy, such as XACML [XACML 13], and evaluated with software that is able to obtain the information of the request and combine it with the applicable policies to make an access decision.

Note that accessing the attributes of the user is not a trivial issue. The easiest option would be to dump all of them on the request after the authentication of the user, but this could lead to privacy issues, due to accessing all the attributes unnecessarily. Another alternative could be to ask only for the needed attributes, but this would require a negotiation with the Identity Provider whose message could last more than the evaluation of the request itself. In any case, the concrete way to deal with this issue depends on each system.

The advantages of this option are:

- Completeness: It is possible to define security criteria that are based in any information that is included each time on the request and the attributes of the user, which can allow these security criteria to be defined even on business specific logic;
- Online refreshment of the attributes of the requester: This approach is not affected because of the attributes of the user being changed, because they are provided per each request;
- Ability to implement policies specific for privacy criteria: The security policies may include decisions based on the purpose of the request, the identity of the human user whose data are being accessed and the identity of the requester, which are needed to take a decision on privacy constraints. Moreover, as this option is not necessarily constrained to evaluate the access decision only before the *service* is reached by the request, it still can check the privacy policy against the result of the request, making possible to evaluate this policies also on the result itself. For instance, if the request asks for a list of cars that passed through a crossing, the initial evaluation of the policy could grant access, but if there are some cars in the list that deny access to its position, a policy based access proxy could check this in the response, that is, after the service has been executed and return a security violation page instead of the response itself;
- Safety: Security good practices encourage checking access to a *resource* each time it is accessed and locate the decision as close as possible to the *resource* accessed as possible and
- The criteria for granting access to a *resource* can be decoupled from the implementation of the *resource* itself.

The drawbacks of this approach are:

- The processing of the security policies require a relatively high computational cost (especially in an IoT environment), which might be relevant if we are executing them for each request and
- The evaluation of the access decision of a *resource* could take far more time if it requires accessing the Identity Provider for retrieving the attributes of the user for each request and very especially if it required communicating with the Identity Provider per each attribute to be retrieved

4.3.4 Conclusions of the Analysis

The only option that seems suitable to cover the very strict privacy requirements addressed by RERUM is the Policy based access proxy. However, due to the very constrained environment of IoT, it is very advisable to take some compromises to try to minimize the impact of evaluating the access of each request, especially in the part that could delay it most, which is the negotiation of the attributes to be retrieved from the Identity Provider. For this reason, RERUM design will have to focus on trying to minimize the messages with the Identity Provider for retrieving the suitable attributes of the user.

Moreover, the set of attributes retrieved will need to be reduced as well, in order to comply with the principle of privacy by design of using only the information of the user that is needed for the operation.

However, the concrete set of attributes applicable to authorize the requests may vary from one request to another. Hence, a minimum set of attributes per request would require negotiating them for each request, which would bring the overload of communicating with the Identity Provider that we aim to avoid. Thus, RERUM's design will have to leverage the balance between an absolutely minimum set of attributes and asking on advance a more complete set for minimizing messages with the Identity Provider.

Finally, the policy-based access proxy ability to access each request also makes it the only one capable to provide security criteria based on business specific logic dependant on the content of the request.

4.4 Design of Authorization Components

Before explaining the design of the authorization components, it is very important to recall that this section is focused on explaining those components that enforce only the security criteria, but not the privacy ones. Although the whole design supports both security and privacy criteria, this design is the result of a first step that focuses only on the security criteria. The full explanation of the remaining components that cover the privacy criteria as well is contained in deliverable D3.2.

The conclusions of the options analysis explain why it is necessary to provide a negotiation to reduce the set of attributes used by the authorization and the number of times it is requested. Although RERUM's design addresses this issue, it is a privacy related one, which is not the scope of this document. The components described in this documents assume that they are provided with all the attributes needed to make the decision. The concrete components that negotiate and gather these attributes are explained in deliverable 3.2.

Figure 33, taken from D2.3, recalls how the main authorization components are meant to interact with each other:

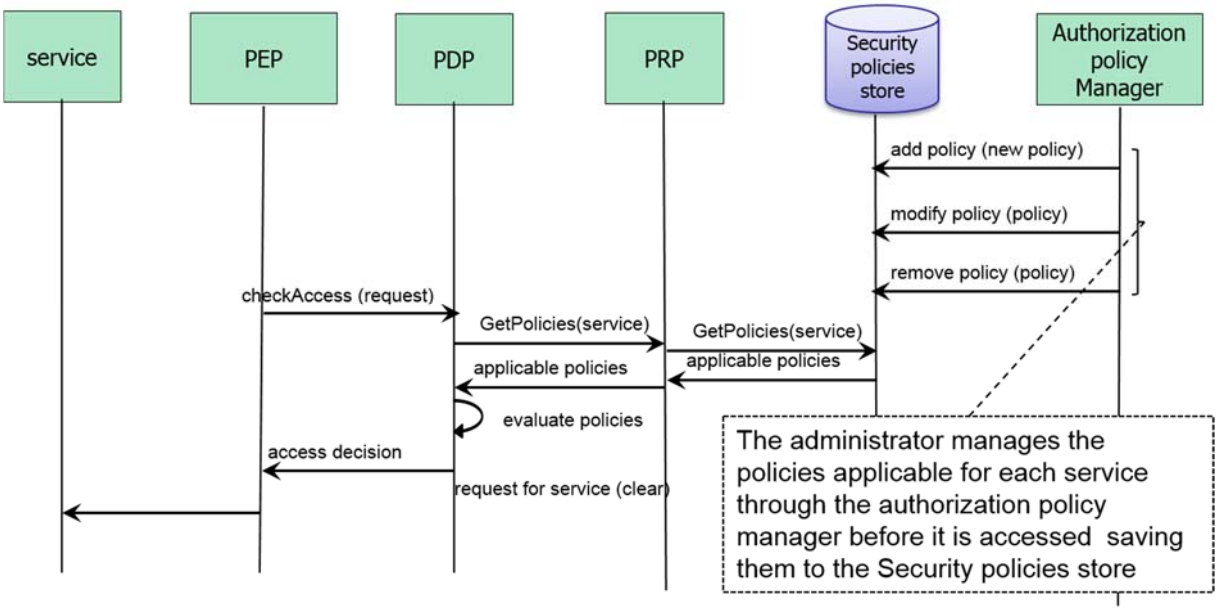


Figure 33: Managing security policies

In short:

0. Security criteria are previously defined by the administrator of the system and saved as security policies through the Authorization policy manager
1. Any request of the system is intercepted by a Policy Enforcement Point (PEP)
2. PEP delegates the authorization decision on a Policy Decision Point (PDP)
3. PDP asks a Policy Retrieval Point (PRP) for the policies applicable to the requested service
4. The PRP retrieves the applicable policies from a Security Policies Store
5. The PDP evaluates the policies and decides whether to grant access or not to that service
6. The PEP allows the request to reach or not the service according to the decision taken by the PDP

The security criteria will be defined in a standard language for specifying security criteria. In concrete, the project has chosen XACML because it is a standard for defining access policies specified by the OASIS [OASIS] consortium widely accepted in the Internet.

4.4.1 Policy Enforcement Point (PEP)

As stated in D2.3: 'The Policy Enforcement Point is responsible for intercepting incoming service requests in the system and granting or denying access to them. To do so, it will retrieve all necessary input from the request in a protocol independent manner and invoke the Policy Decision Point (see below) to know whether to grant or reject the decision. Depending on that, the PEP will let the request pass or it will reject the request. The sensitivity and the requirements to access a certain set of data are described in the access policies. The input to the PEP will be the request itself, which must contain a security token with the information related to the attributes of the user issuing the request. The output of the PEP will be either letting the request to pass or rejecting it and returning a security error page'.

Hence, the enforcement action consists in letting pass or rejecting an intercepted request. But intercepting a request is not trivial and can be carried out in different ways, including, but not limited to the following three options:

Option (1) Deploying all services and resources in an application server and installing the PEP as a filter for the applications;
Option (2) Designing the PEP as a proxy installed in the same host that is running the services and resources and intercepting all traffic that goes to the ports where they are listening and
Option (3) Installing a proxy that is running on a separate proxy.

Option (3) is not contemplated in the architecture and option (1) might need a lot of resources that might not be necessarily be available on a GW instance. For this reason, the RERUM PEP will be designed according to option (2). That is, it will be an independent application installed on the GW instance itself that will be listening in the same port where the services and resources are meant to be exposed and will grant or reject access to the corresponding services depending on the decision taken by the PDP.

However, though RERUM implementation will be based on option (2), the design will be prepared to support the three options, by separating the authorization logic from the intercepting one. Hence, we will have:

- Intercepting logic: Proxy part
  - Obtain destiny host and ports (for possible redirections of case 3)
  - Obtain listening port (for intercepting requests)
  - Listen on intercepted port
  - Listen on intercepted response
- Authorization logic
  - Obtain security (authentication) token from the request
  - Gather relevant information of the request, including the token, and place it in protocol independent structure
  - Evaluate access for the request, by invoking the PDP passing it the information gathered before
  - Evaluate access for the response, by invoking the PDP

Note, however, that actually the evaluation of access for the response is only necessary for privacy issues, and hence, it will be discussed in D3.2.

In any case, the PEP needs to have two subcomponents. The first subcomponent, which will be named Security Interceptor, will take the responsibility for carrying out the intercepting logic. As mentioned, this logic can either be implemented as a proxy or as a filter installed on an application server. The second subcomponent will be the authorization logic stated for the PEP and will be named ‘Authorizer’.

The Authorizer itself will delegate the decision in the PDP, which is explained in the following section. For collecting the information from the request and passing it to the PDP, the PEP uses an object named XACMLContext, which is an XACML standard way to pass information to a PDP. In practice XACML Context is an object that constructs a valid XML content compliant with the XACML specification for providing input to a PDP from the information contained in the request, whose name in the XACML specification is Access Context. It is also able to interpret the decision of the PDP itself. The only reason for naming this object ‘XACMLContext’ instead of ‘AccessContext’ is only to differentiate it from the internal objects that the XACML libraries uses internally in the PDP.

Figure 34 shows the relationship of the distinct components mentioned.

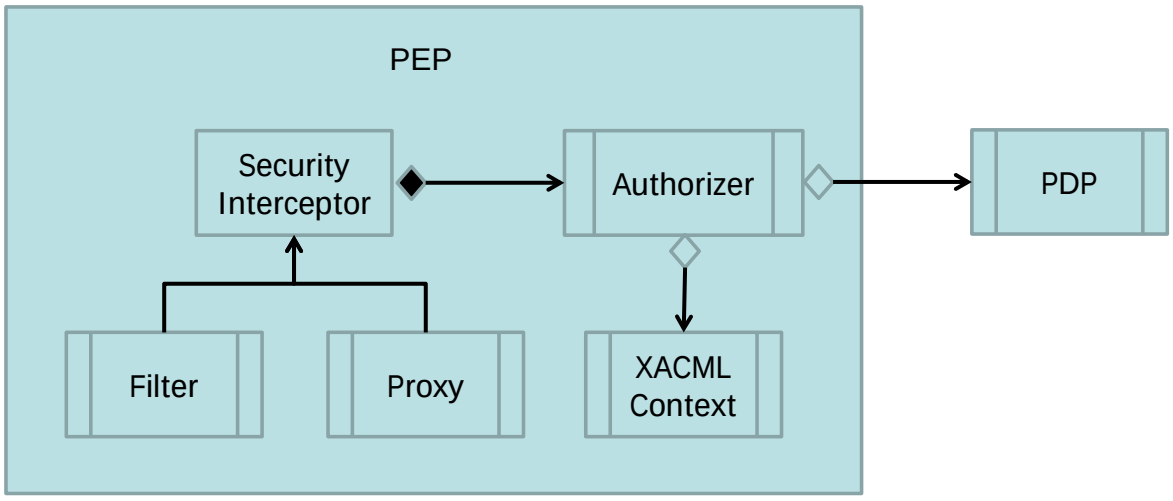


Figure 34: PEP components and their relationship with PDP

As the figure shows, the PEP is comprised of a Security Interceptor and an Authorizer. The Security interceptor may be a Filter to be installed on an application Server or a classic intercepting proxy. The Authorizer carries out the authorization process by invoking a PDP. Figure 35 show the processing of a request.

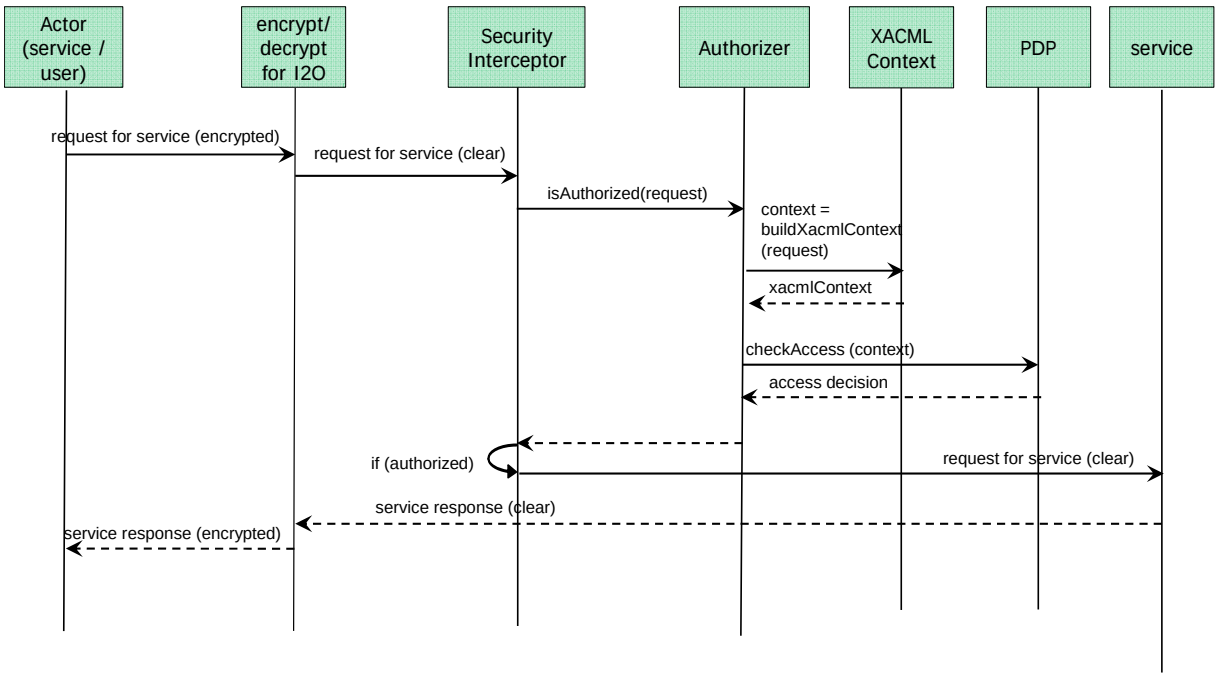


Figure 35: Interaction of PEP components with the rest of the system

The XACML context is paramount for one of the innovations in RERUM. Although XACML itself already supports the evaluation of business specific criteria by including request parameters in the resource part of the policy, the problem with this approach is the following: it is necessary that the PEP that will invoke the PDP with the XACML libraries provide the values for these request parameters, and for providing these values, the PEP itself needs to know where to obtain them. In practice, this is usually dealt with by hard coding in the PEP where to obtain these values. But the problem with this approach is that: it makes the PEP dependent on each service to protect, because it needs knowledge about how the service work, and this makes impossible that the same PEP can serve other services it does not know. In short, supporting business specific logic usually comes at the cost of lack of adaptability. But in RERUM we follow a different approach. In concrete, RERUM assumes that all the information contained in the request will follow a given usual nomenclature in case it is really there, and dump this

information in the XACML content following this nomenclature without needing to have any previous knowledge of the nature or structure of the service.

The exact nomenclature assumed by the XACML Context is the following:

Request content	`Entry name`	**XACML context section**
Protocol version	`REQUEST_LINE_protocol_version`	<resource>
Request line method	`REQUEST_LINE_METHOD`	<resource>
header	`HEADER_<headerName>`	<resource>
uri	`REQUEST_LINE_protocol_version`	<resource>
Security token: user attribute	`<Atribute_name>`	<subject>
Uri.query	`REQUEST_Query`	<resource>
Uri.authority	`REQUEST_Authority`	<resource>
Uri.fragment	`REQUEST_Fragment`	<resource>
Uri.host	`REQUEST_Host`	<resource>
Uri.path	`REQUEST_Path`	<resource>
Uri: resource identifier	`ACTION_action_id`	<action>
URL: named_parameter	`ACTION_<param_name>`	<action>
URL: unnamed_parameter	`ACTION_PARAM_<param_order>`	<action>
Body of type HTML. POST field	`POST_<fieldName>`	<resource>
Body of type Form: POST field	`POST_<fieldName>`	<resource>
Body of type XML: tag value	`<xpath name of the tag>`	<resource>
System.CurrentTime	`urn:oasis:names:tc:xacml:1.0:environment:current-time`	<Environment>

Figure 36: Nomenclature for request contents to be included in the XACML context

Following this nomenclature, a policy administrator, who knows the structure of the services of the system, may create policies according to this nomenclature, which, together with the ability of the PEP to include each value of the request in the XACML context following the same nomenclature, will allow the PDP later to properly evaluate the XACML policy.

4.4.2 Policy Decision Point (PDP)

As stated in D2.3: 'The Policy Decision Point is responsible for deciding whether a request is granted or not, based on the information provided by the PEP and the security policies applicable for that request. The PDP utilizes its own specific data structure and it does not reuse the original request, aiming to be independent from the protocol that the request was originally carried on. The input of the PDP will be a protocol independent description of the request containing the information necessary to evaluate the security policies corresponding to the requested resource. The result of the PDP will be a protocol independent response stating whether to accept or not the request'.

In the case of RERUM, the PDP evaluates XACML policies applicable to a request for a concrete XACML content, which is the protocol independent description stated in D2.3. For doing so, the PDP makes use of libraries that evaluate the XACML policies themselves. It might be argued that if such libraries already exist, then there is no need for the PDP object itself. However, those libraries have the problem that they do not offer a common API and the version 2.0 of XACML is not compatible with version 3.0. Hence, for a matter of maintenance and versatility, the PDP object wraps the access to these libraries for the PEP in a way that is independent from the library used and the version of XACML being used.

Besides, a given request can be applicable to many policies. For this reason, it is necessary to previously obtain the policies applicable. This task is delegated in the Policy Retrieval Point explained in the next section.

Figure 37 shows the classes involved in the process and how they work with each other.

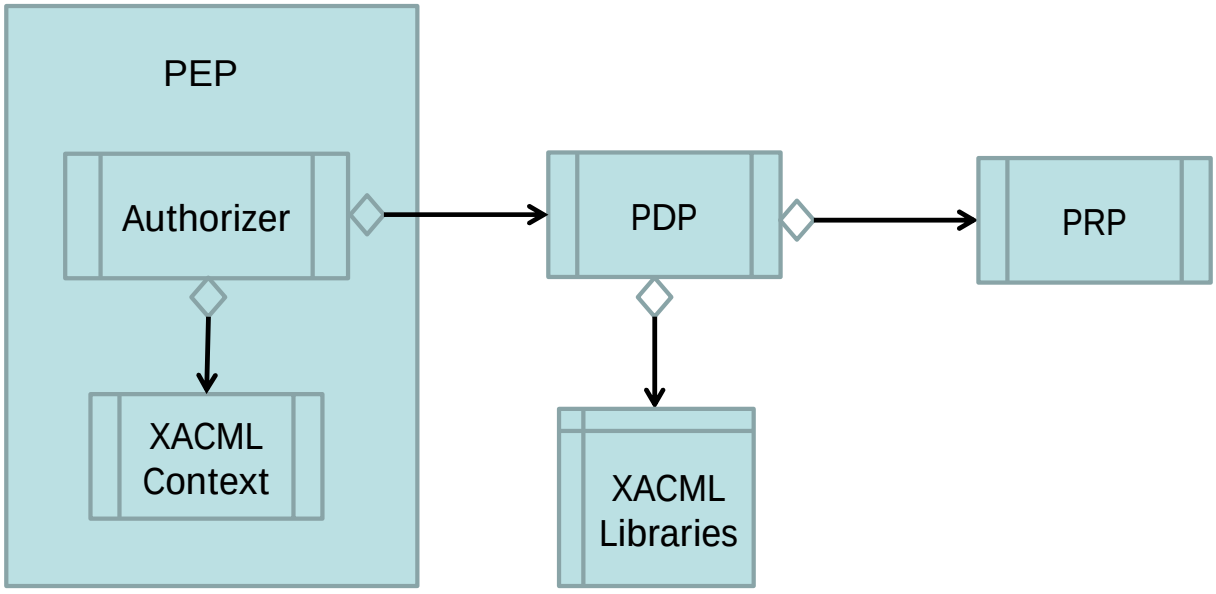


Figure 37: PDP related classes

4.4.3 Policy Retrieval Point (PRP)

As stated in D2.3: 'The Policy Retrieval Point provides the security policies applicable for the requested services so that the PDP can evaluate them. The input of the PRP will be a protocol independent request including at least the data referring to the resource whose access wants to be decided. The output of the PRP will be the content of a set of policy files / rules applicable for the resource requested'.

There are 2 possible ways to achieve this:

a) Let the PRP itself filter policies that have to be applied in base of the URL requested, possibly according to some kind of configuration and

b) Letting the XACML library itself to decide whether a policy is applicable or not by evaluating its `<target>` section, whose purpose is precise.

Note that actually a) and b) do not necessarily exclude each other, but they can be combined together, instead. In fact, this is the approach of the RERUM PRP. That is, the RERUM PRP first make a selection of applicable policies and nothing impedes these policies to include their own <targets> sections that the XACML library will evaluate to constrain the set of applicable policies further.

In a first approach, the PRP could make a direct association between the resource to be authorized and the policy to be applied. However, this would force to have at least one policy for each resource to be authorized, making almost impossible to reuse the same policy for a whole set of resources. But in practice, RERUM resources might be grouped according to his URL. For instance, the indoor comfort management use case could state that only the members of a given floor can adjust the temperature of the floor, but this is applicable to all the temperature actuators of the same floor.

The actuators are being exposed as REST services with the given structure:

`https://<anyRERUMhost>/temperature/floor/set/?floorNumber?newTemperature`

In this case, we are interested in reusing the same policy that check the number of the floor, so we could be interested in setting up a policy for all URLs that match '`https://myRERUMhost/temperature/floor/set`', and this can be achieved easily with a regular expression.

And this is exactly what the PRP does. The PRP reads its configuration from a file, which assigns regular expressions in java format to a set of applicable policies, including their corresponding relative paths for them, which allows defining and storing these policies in a hierarchical way. Moreover, as a given URL may match more than a single regular expression, it is even possible to define overall regular expressions that point to policies defined by the overall administrator of the system and more specific regular expressions that point to finer grained policies.

Upgrading the previous example, an overall administrator policy could state that only users that are active can access the system. In practice, this would lead to a configuration like:

```
https://myRERUMhost/.*:active_user_policy
https://myRERUMhost/ temperature/floor/set .*:temperature_setting_policy
```

The first line will match all URL's starting by '`https://<myRERUMhost>`', that is, all https accesses to the host `myRerumHost`, including, but not limiting to the temperature setting ones. And it means that all operations addressed to `myRerumHost` will be applied the policy active_user_policy.

The second line will match only those urls starting by '`https://myRERUMhost/temperature/floor/set`', that is, only the temperature setting operations, and it means that all temperature setting operations will be applied the policy `temperature_setting_policy`.

Hence, for a temperature setting operation, the PRP will select 2 policies, `active_user_policy` because of the first line, and `temperature_setting_policy` because of the second line.

It is also possible that we want to apply distinct policies to the same resource that each of them evaluates a single condition. For instance, the room temperature service could have several operations for getting and setting the temperature, which are discriminated by the http method (PUT or POST). Although it is possible to have a single policy stating conditions for method = PUT and method = POST, it would be a quite complex policy and the administrator of these service could be interested in having distinct policies because he conceive these operations are distinct services. For that case, the PRP allows defining several policies applicable to the same resource. The method for doing that is separating the applicable policies by commas. In this case:

```
https://myRERUMhost/.*:active_user_policy

https://myRERUMhost/temperature/floor/.*:temperature_SET_policy,
                                         temperature_GET_policy
```

The first line is identical to the previous example and states to apply the policy to check that the user is active for all services. The second line states that all the floor operations must be applied the policies `temperature_SET_policy` and `temperature_GET_policy`. These two policies are expected to check the field `REQUEST_LINE_METHOD` for checking the proper request method as stated in Section 4.4.1.

Another interesting point is depending on the library implementing the XACML standard, it might not possible to use always the same PRP obtaining the policies directly, but the library could impose a given PRP itself, though with a different name. For instance, the sun library for XACML 2.0 in practice imposes some concrete classes for accessing the policy files, and also implies them to be accessed as physical files. Though this is acceptable as a first approach, a better one could be to at least support caching the policies in memory so they do not need to be read from the disk each time the request is evaluated.

For this reason, the PRP will be a class Factory itself, with will contain the proper `getInstance` method to retrieve the suitable PRP object. Alas, as this object will be defined by the third party library implementing the XACML standard, there is no way to enforce it will follow a given interface. This forces to define its method as Object `getInstance(enum engineType)` and later cast the returned object to each required type.

Figure 38 shows how PEP, PDP, PRP and the trust engine interact with each other to grant or deny access to the services requested:

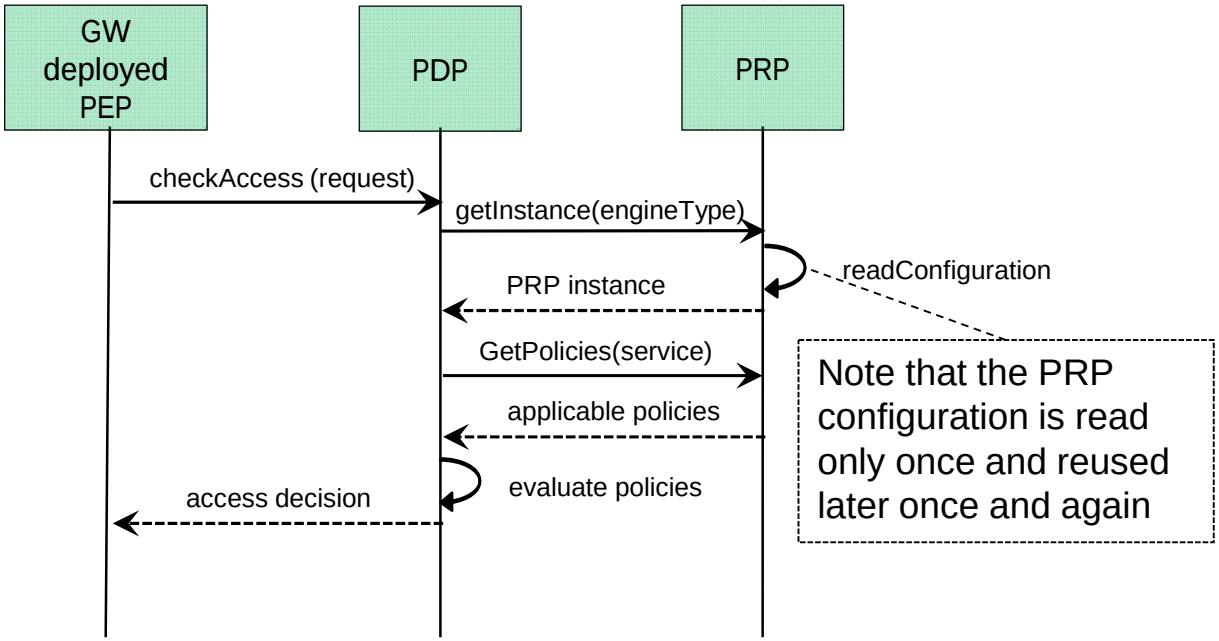


Figure 38: Interaction PDP-PRP

4.4.4 Introduction Level

As explained before in Section 4.2 for the Service level authentication in RERUM, the authorization needs to know some attributes of the user that is issuing the request and, more importantly, it needs to make sure that the requester is he who claims to be, and this is a responsibility of the Identity Provider. However, there are several ways that this provider may provide this information, each of them with his pros and cons. In concrete, the authentication provider may provide the attributes of the user when entering the system or it may provide them when explicitly asked for them. As the authentication provider is not part of RERUM, the project can hardly guarantee this. But the authorization components still need this information to be provided, and preferably always in the same way. For this reason, and to ensure the authorization components always receive the attributes of the user in the same way, RERUM includes a new component that will do this homogenization work, which is named the 'Identity Agent' (IdA). The IdA checks that the user has properly been authorized by the identity platform and collects the information needed about the user for the authorization process. Next, it includes them as a security token in a header of the request so the authorization components can use them in the decision process. As the information will be in a predefined header of the request, which already complies with the way the XACML context collects the information for sending it to the PEP, the authorization components do not be even need to be aware of the IdA making his job. They only need is to check that the security token collected from the header has been signed by the IdA.

There is another functionality / reason for the IdA. The collection of user attributes can be a heavy task, especially if RERUM is aimed to be an interoperable platform. In order for RERUM to be an interoperable platform, the retrieval of the attributes of the user should either be based on open standards, such as SAML, or implemented at least for the main Identity Providers. And in both cases the retrieval of these attributes is subject to be a heavy task, both in terms of computational resources and number of messages transmitted to retrieve the messages. Although it is possible to retrieve the needed attributes for each request, this way would imply a lot of messages for retrieving the relevant attributes for each request, which is even worse having into account that many http requests often imply additional requests associated to the main one. For this reason, it is very important that the IdA is also responsible for holding a cache of the attributes of the recent requesters so it does not need to ask for them for each request but it supports some expiry date for them after what they are removed.

This cache feature has strong implications from the point of view of privacy, because the IdA should, among other things, ask only for those attributes that are needed, ensure it is not possible to disclosure the identity of the requester and keep them updated. However, privacy is not the scope of this document and for that reason the components related with the IdA that are related with the enforcement of the privacy and their explanations are included in the D3.2 document, which is focused on privacy issues.

The signature is necessary for security reasons. The IdA and their transmissions are subject to attacks as any other component of the system and for that reason their transmissions also need protection. In more detail:

- The security token must be signed for avoiding it to be forged
- The signature of a security token should have an expiration date assigned
- The transmission of the security token must be encrypted to avoid confidential information of the user to be disclosed

There are basically three options how the IdA could provide the security token to the Authorization layer:

Option (a) The applications get authenticated once in the Identity Provider of their choice by providing it the credentials of the user and pass their authentication token to the IdA each time it is executed, which is once for each request, and the IdA talks with the Identity Provider (if necessary) each time to retrieve the user attributes;

Option (b) The applications get authenticated once in the Identity Provider of their choice by providing it the credentials of the user and pass their authentication token to the IdA just after being authenticated to have the IdA obtain the attributes from the Identity Provider once or

Option (c) The applications get authenticated once through the IdA by providing it the credentials of the user, which forwards the authentication request to the proper Authentication provider and asks it for the proper user attribute only once.

Option (a) is undesirable because it means that the IdA needs to make a conversation with the Identity Provider to retrieve the attributes of the user for each request. Options (b) and (c) do not have this problem and could both suit RERUM, but option (c) has the advantage for the RERUM applications that they only need to make a single interaction with the IdA and let it to interact with the Identity Provider instead of invoking first the Identity Provider and later the IdA. More importantly, in option (c), the IdA acts as a façade for the Identity Provider itself, offering the same interface and hiding the complexity of the process, and thus comply with the standard procedures stated in Section 4.2 for the Service level authentication in RERUM. For this reason, the option chosen is (c), which is illustrated in the Figure 39.

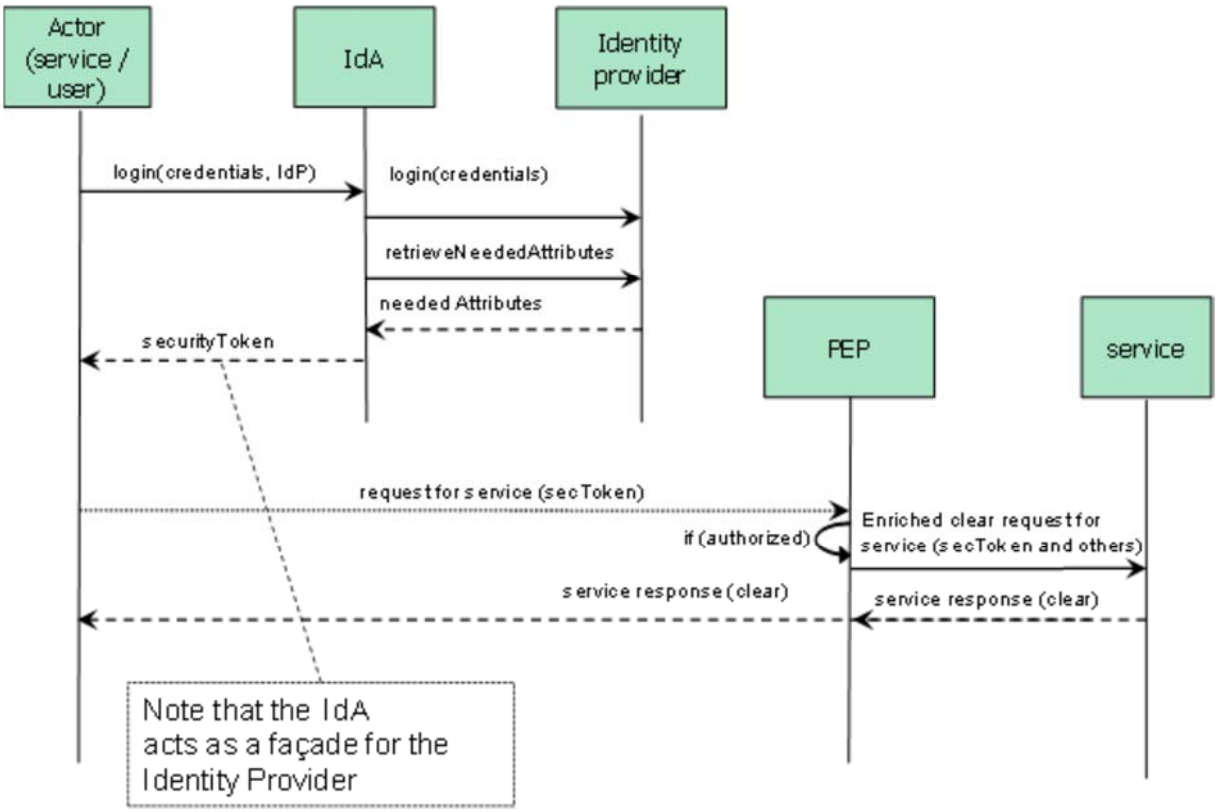


Figure 39: Obtaining security token through the IdA and using it

As explained, the IdA has to interact with the Identity Provider to get it to authenticate the request and obtain the proper attributes. However, each Identity Provider is subject to work in its own way, which leads to many possible interfaces to be implemented. For this reason, the IdA allows indicating the API he will need to use to communicate with the Identity Provider. If he is not reported, It will assume SAML by default because it is a widely used Internet standard.

4.4.5 Authorization Policies Manager

The authorization policies manager allows defining the security criteria used for deciding whether to grant or reject access to the RERUM system. These criteria will be saved in a standardized format so they can be obtained later by the PRP (see Section 6.11.1.7) to be provided to the PDP (see Section 6.11.1.6) so it can evaluate them against the request to take a decision. The Authorization policies manager supports the standard operations for adding, deleting and modifying policies. The Authorization Policies Manager takes as input the policy files provided by the administrator of the system and gives as output the Policy files to be saved to the **Security Policy Store**, which can be considered as a specific instance of the Trusted Credentials Store described above.

Figure 40 shows the relationship between the Authorization Policy Manager and the rest of the components involved in the evaluation of the security policies of the system:

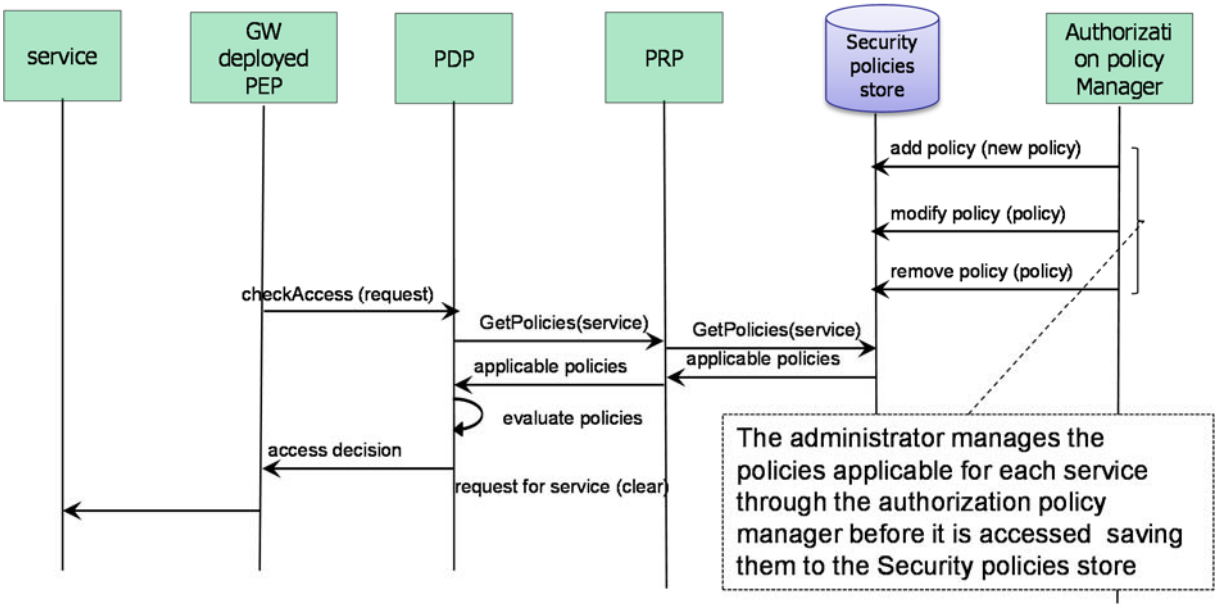


Figure 40: Managing security policies

Note: The interactions for the "modify" and "delete" operations are basically the same as for the "add" operation. That is, they modify the set of security policies applicable for a given request and these are finally supplied by the PRP.

4.4.6 Interaction with Other Modules

The authorization components that need to interact with other components of the system are:

- The IdA talks with the Identity Provider to check the identity of a user and obtain their attributes;
- The PEP let pass or rejects the requests coming from the Internet to the requested RERUM services and
- The authorization policy manager lets the administrator providing the authorization policies to the PDP;
- The architecture of the System let other components, such as the reputation engine, to provide additional information to the PEP by acting as additional proxies themselves that add more headers to the request that PDP will process later.

Basically, the Policy Retrieval Point also needs to talk with the consent manager to obtain the privacy policies to be evaluated on each request. But that is a privacy issue that is covered in 'D3.2 Privacy enhancing techniques in the Smart City applications' and hence its interface is not defined in this section. Besides, the PEP also interacts with the reputation engine, but that interaction is related with the trust engine, which will be described in 'D3.3 Modelling the trustworthiness of IoT'.

The rest of the authorization components only interact internally with each other. That is, they do not publish or use any external interface and hence they are included here.

4.4.6.1 Interaction of the PEP with the requested RERUM services

The PEP is designed to be transparent for the requested RERUM services and almost transparent for the requesting applications.

In the case of the requesting applications, it is 'almost' transparent because the PEP needs a security token issued and signed by the IdA to both check that the user has properly been authenticated and

access the needed attributes of the user for checking its access. The exact structure of the security token issued by the IdA has been previously defined in Section 4.4.4 Introduction Level.

The interaction with the services of the PEP with the services is shown in the following Figure 41:

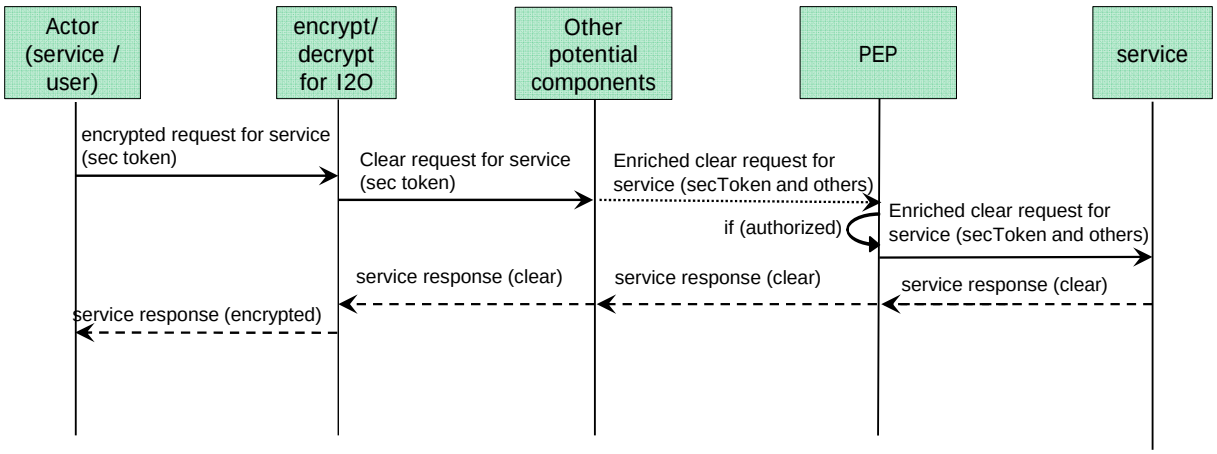


Figure 41: PEP interaction with services

As the figure shows, after encryption / decryption provided by the TLS protocol of a request that already included a security token, some other components can be hooked to make additional process and potentially modify or enrich the request. In fact the encryption / decryption components work that way and there is at least one component, the reputation engine, which will be connected this way in the future. Here the key point is the PEP gets a non-encrypted request whose headers include at least a security token previously signed by the IdA, but the other potential components may modify the request further, for instance, adding headers to the request with an evaluation of the reputation engine.

Once the PEP gets the enriched request, it decides whether to grant access or not to the requested resource. If it does, then it forwards the enriched request to the request to the service, which will also allow the service to invoke other services subject to authorization themselves.

In short: Though the PEP interacts with the requested services, this interaction is transparent for them and hence there is no additional API involved.

4.4.6.2 Interaction of the authorization policy manager with the PEP

The authorization manager exposes a Java Interface `AuthorizationPolicyManager` with a single method that is exposed below:

```
void IncludePolicy(String resourcePath, String policy, String policyId);
```

where

`resourcePath is a URL identifying the resource to be protected`

`policy is the xacml policy to be evaluated when the resource is trying to be accessed. Note that it makes sense to have several policies applied to the same resource.`

`policyId is a unique identifier for that policy and resource. Note that it makes sense to have to policies with the same policyId as long as they do not refer to the same resource.`

The policy manager is defined to allow combining several policies. However, RERUM use cases only use this feature for combining access and privacy policies. For this reason, the implementation and details of this feature will be described in D3.2

5 Secure RERUM Device Configuration

5.1 Introduction, Motivation and Link to User Requirements

This chapter details the procedures and components involved in ensuring the proper configuration of each device connected to the RERUM network and also provide mechanisms to detect and react to malfunctions or misconfigurations on those devices. This configuration includes managing the software updates of the RERUM Devices and, potentially, registering and processing events received from the whole system.

In the sub-sections below the design of the following protocols and components is explained:

- The 'Fast and Secure Network Bootstrapping' section defines the approach used in RERUM to minimize security attacks at the initial bootstrapping of RERUM Devices added to the network. This approach describes the technical contribution 'Contribution 1: Secure Credential Bootstrapping' stated in D2.1. The RERUM architecture uses several security mechanisms to provide security and privacy during operation. As the strength of these mechanisms relies on the confidentiality of the applied credentials the secure bootstrapping of security credentials is necessary for all use cases considered in RERUM. The user requirement linked to this sub-section is:
    - UR-1: A user wants to securely introduce a new device into the network, so that afterwards messages sent by his device to the gateway and to other devices of his network are kept confidential and are not modified by unauthorized people.

  The sub-section details the steps for network bootstrapping with the focus on bootstrapping of security credentials. These steps consist of "Initialization of the Security Center", "Initialization and bootstrapping of the RERUM gateway", "Distribution of a Join Key" and "Initialization and bootstrapping of RERUM Devices".
- The 'Secure and context aware dynamic auto configuration' section covers the technical contribution 'Contribution 12: Incorporating adaptability to an IoT platform using PRRS and OAP' of D2.1. This contribution allows the maintenance of the system fixing bugs or security vulnerabilities, updating software, configuring devices and in general managing remotely the whole RERUM network. This technical contribution consists in incorporating the PRRS Module and the OAP Module as parts of the SW Component Manager described in the RERUM Architecture. The user requirements (UR) linked for this section are:
    - UR-14: The user requires being able to remotely configure and upgrade the firmware and the security mechanism of his devices in an automated way, without needing any manual installation.
    - UR-15: The administrator also requires low maintenance costs and low technical administration overhead.
- In the 'Self management and self monitoring mechanisms' section contains information about the technical contribution 'Contribution 16: Lightweight framework for sensor monitoring', explaining how the devices are monitored and also how this data is processed to perform a real-time analysis of the system events by integrating a SIEM in the RERUM network, as is said in the technical contribution 'Contribution 11: SIEM in a generic IoT platform'. The SIEM introduced here is a way to implement examples of Alert Processors and Alert Reactors as they are described in the Architecture document together with a Resource Monitor that provides the monitoring information to process. All this modules are inside the Monitoring Manager Functional Component of the RERUM Architecture and linked with the user requirements:
    - UR-13: The user doesn't want to worry about reporting errors or malfunctions on the Smart Devices or Network to fix them. Those problems must be auto-detected.

- UR-19: User needs to have an automated system for monitoring the status of the devices and of the network connections to avoid missing alarms and to ensure the functionality of the services.

For the evolution and adoption of the IoT, technologies need to facilitate the use to regular users by making the processes transparent for them. With the innovations described in this chapter the RERUM system becomes more adaptable, secure and easy to use.

In all use cases defined in RERUM, the secure installation of a new device in the RERUM network, the auto detection of any problem and the automatic configuration of the installed device are common requirements that will be covered here.

5.2 Fast and Secure Network Bootstrapping

The RERUM architecture contains several security and privacy mechanisms to protect the RERUM system from security attacks and loss of personal data. Examples earlier described in this document, are Device-to-Device authentication, Secure Communication, Authorization or privacy mechanisms that will be described in the future RERUM deliverable D3.2. These security mechanisms typically rely on security credentials and the strength of the security architecture depends on the strength of the security credentials. If the credentials are weak or compromised, the security mechanisms cannot fulfil the intended security properties such as provisioning of confidentiality, authenticity or integrity.

In that sense, the setup of a network of RERUM Devices is a sensitive process and innovation 1 'Secure Credential Bootstrapping' of the RERUM deliverable 2.1 is about provisioning and bootstrapping of security credentials in a secure manner. It is relevant for each of the four use cases of RERUM. Fresh RERUM Devices to be installed as part of a Smart City use case typically do not have security credentials available in the first step, so that some bootstrapping mechanisms need to be in place. Bootstrapping of credentials is closely related with the setup of the network and the setup of the first communication links between RERUM Devices or other entities. A possible attacker may be already in place while a network of RERUM Devices is installed or an attacker may later try to connect or insert a malicious device into an existing network. The goal of fast and secure network bootstrapping is to minimize security attacks at the initial configuration of a network of RDs. When the network is initialized, the objects are quite vulnerable, because they do not have any knowledge regarding the environment and their neighbour nodes and no knowledge about security credentials to protect communication. Also it is the goal of fast and secure network bootstrapping to avoid the installation of malicious devices into an existing network.

In most of the envisioned use cases a network of constrained RERUM Devices rely on the following network layers according to the OSI layer model (the following bootstrapping procedure does not concern the case of arbitrary participatory sensing devices, such as citizen smartphones):

- Layer 2: IEEE802.15.4[49] [IEEE802.15.4]
- Layer 3: 6LoWPAN[50] [RFC4944] and RPL[51] [RFC6550]
- Layer 4: UDP and ICMP[52]

[49] IEEE Standard for Local and metropolitan area networks--Part 15.4: Low-Rate Wireless Personal Area Networks (LR-WPANs)

[50] Transmission of IPv6 Packets over IEEE 802.15.4 Networks

[51] IPv6 Routing Protocol for Low-Power and Lossy Networks

The secure bootstrapping procedures start from establishing the lower level layer 2 links, getting a security base for further instantiation of the layers above and bootstrapping of operational credentials required within the RERUM network and Smart City use case.

Figure 42 depicts an example of a network of RERUM Devices from a layer 2 and 3 network point of view. It consists of one or more PAN coordinators, each responsible for its PAN cluster, and constrained RERUM Devices being connected via IEEE802.15.4 hops towards the PAN coordinator. In RERUM the role of a PAN coordinator is applied by the RERUM gateway which provides also the link towards the Internet. The bootstrapping of credentials is controlled by the Security Center which is located in the Internet. Depending on the concrete application use case, the Security Center may be also located on one of the RERUM gateways. The Security Center is responsible for authentication and authorization of new RERUM Devices and granting access to the RERUM network by provisioning of operational security credentials to the RERUM Devices in a secure manner.

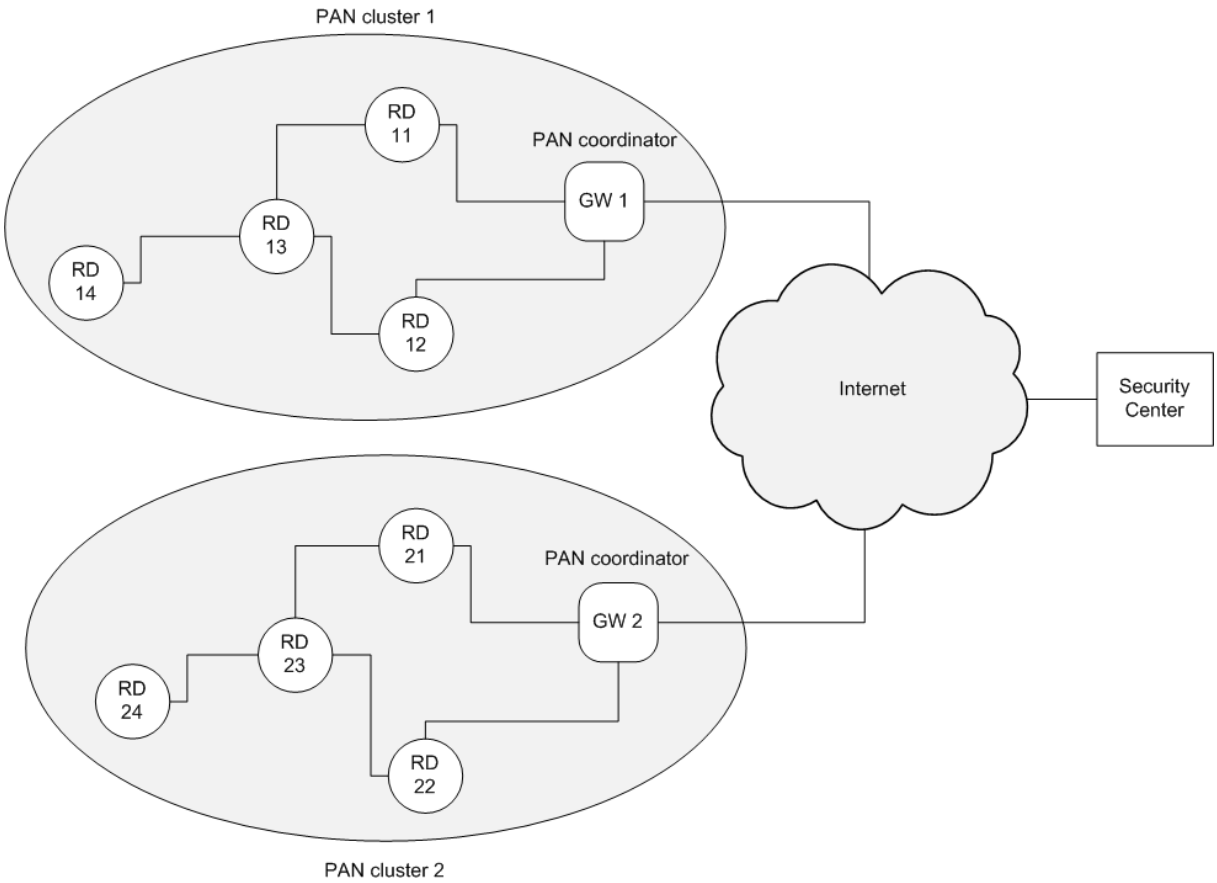


Figure 42: Network of RERUM Devices

In the following description we will explain the bootstrapping procedures with focus on bootstrapping security credentials in a secure way, first introducing a RERUM gateway into the system and then installing and connecting RERUM Devices to a PAN cluster controlled by the RERUM gateway. The steps consist of:

- Initialization of the Security Center: Here the first steps are described to set up the Security Center for bootstrapping of devices;
- Initialization and bootstrapping of the RERUM gateway: In a second step the first RERUM gateway is initialized to connect to the Security Center;

- Distribution of the Join Key: The Join Key with which a RERUM Device protects the first communication to the Security Center is distributed and
- Initialization and bootstrapping of RERUM Devices: A new RERUM Device is connected to the RERUM network and the credentials are bootstrapped.

5.2.1 Related Technology

The RERUM network relies on different network layers, each providing its own security mechanisms. However, often standards are explicitly excluding the procedures how keys are distributed towards different entities before the specified security mechanisms can be applied.

IEEE802.15.4

This standard specifies different security mechanisms to protect authenticity, integrity or confidentiality of layer 2 frames. The procedures to bootstrap the required keys towards the network entities are excluded in the standard. There has been much research work in the past on defining key management procedures for wireless sensor networks. All have in common that a manual pre-installation phase is required before devices can be bootstrapped. The key management approaches can be categorized into the following groups:

- Key management based on network key (e.g., [LKV02])
- Full pair wise keys (e.g., [CP05])
- Key management based on group keys (e.g., [WP++13])
- Trusted third party based key management (e.g., [PS++02])
- Matrix based key distribution (e.g., [RSH10])
- Polynomial based key distribution schemes (e.g., [LN03])
- Hash based key distribution schemes (e.g., [BCB13])
- Location based key distribution (e.g., [RLZ08])
- Probabilistic key pre-distribution (e.g., [CPS03])

There is no optimal key management schema available. Either

- protocols have a limited security scope, because they rely on few keys which are known to multiple entities or
- key distribution schemas require much storage capacity and do not scale, e.g. to store all possible pair-wise keys or
- protocols require much computation resources to calculate pair wise keys with neighbour hops.

6LoWPAN

Research is on-going to support IPSec communication security over 6LoWAPN networks too and first few IPSec implementations exist in that context. However, the challenge to bootstrap credentials onto constrained devices that are applied for establishment of authenticated communication channels is not solved. These credentials need to be configured manually.

RPL

RPL provides security mechanisms to protect routing messages. The key management aspect is also excluded from that standard and future research. The application of security mechanism for RPL routing messages can be also omitted if the layer below provides the required security properties.

DTLS

Similar as IPSec, DTLS is a protocol to establish a secure channel and to negotiate temporary session keys. Authentication of the DTLS handshake protocol requires some existing credentials on the constrained devices. The standard does not specify the procedures to bootstrap these kinds of keys.

EAP

EAP is a generic framework for authentication of devices, e.g. used in the context of 802.1X authentication in WLANs or fixed LANs. EAP authentication relies again on some pre-installed credentials to authenticate the requesting and responding entity, which are not available for newly introduced devices. In addition, as EAP is a generic framework to fulfil a broader range of scenarios, it does not fit well in the context of constrained devices with a limited ROM.

PANA

PANA adapts the EAP framework on the IP layer. In that sense the same limitations exist as described for EAP.

Conclusion:

Smart City applications and communication of RERUM Devices rely on different network layers. The available security approaches per layer cannot be considered separately, but they need to be considered in conjunction with each other to find the right balance. It is not reasonable using all security mechanisms of each layer in parallel, but to focus on the required ones. The security mechanisms described above typically consider only one layer. Bootstrapping procedures to distribute necessary credentials are not described by the standards at all or are designed in the context of a specific layer only.

The RERUM credential bootstrapping procedures described in the next sections are balancing the use of security mechanism across network layers as necessary to secure the distribution of different type of credentials needed by the RERUM framework. They start from establishing the lower level layer 2 links, getting a security base for further instantiation of the layers above and bootstrapping of operational credentials required within the RERUM network and Smart City application.

5.2.2 Initialization of the Security Center

The Security Center (SC) is a trusted RERUM server in the Internet providing security and privacy functionalities to RERUM Devices requiring a central reachable server component. For the bootstrapping process the Security Center is responsible for authentication and authorization of new RERUM Devices and granting access to the RERUM network by provisioning of operational security credentials to the RERUM Devices in a secure manner.

The Security Center is located in the Internet and may be installed as separate entity or on a RERUM gateway depending on the HW resources of the gateway. For indoor use cases the Security Center may be also located within the local network / Intranet of the indoor application.

The Security Center is being controlled by a security administrator who is authorized to configure security credentials (see Figure 43). For proper operation of the bootstrapping procedures described later on, the Security Center acts as key management center and PKI certification authority. As

described in section 2.5 a flat hierarchy of certificates is intended for RERUM Smart City applications and the trust anchor of the PKI is the certificate of the Security Center.

During the bootstrapping procedures secure communication channels will be established towards the SC, either from the RERUM gateway or RERUM Devices. These channels will be realized by TLS in case of the RERUM gateway and DTLS in case of the constrained RERUM Devices using the appropriate credentials. The TLS and DTLS cipher suites shall support mutual authentication.

5.2.3 Initialization and Bootstrapping of the RERUM Gateway

The first step to set up a network of RERUM Devices is the installation of a RERUM gateway. The gateway acts as a IEEE802.15.4 PAN coordinator to whom RERUM Devices can connect to. The gateway provides also the transition from/to the Internet. For indoor use cases the RERUM gateway may not need to provide this transition depending on the concrete application scenario.

The RERUM gateway hardware platform differs from the hardware platform of the constrained RERUM Devices, as it has more resources for operation as network transition point and provisioning of interfaces to the RERUM middleware, if the middleware itself is not running on the gateway.

The gateway is configured by a RERUM administrator responsible to setup the network. Besides network configuration, the RERUM administrator also needs to configure the PKI credentials or certificate issued by the Security Center to allow establishment of mutual authenticated TLS connections. This process follows state-of-the-art certificate enrolment protocols applied in the Internet and Enterprise networks already today and is out of scope of this document. Examples are PKCS#10 requests, Simple Certificate Enrolment Protocol or Certificate Management over CMS. The RERUM administrator needs also to configure the trust anchor of the Security Center.

Figure 43 depicts the bootstrapping procedures of the RERUM gateway:

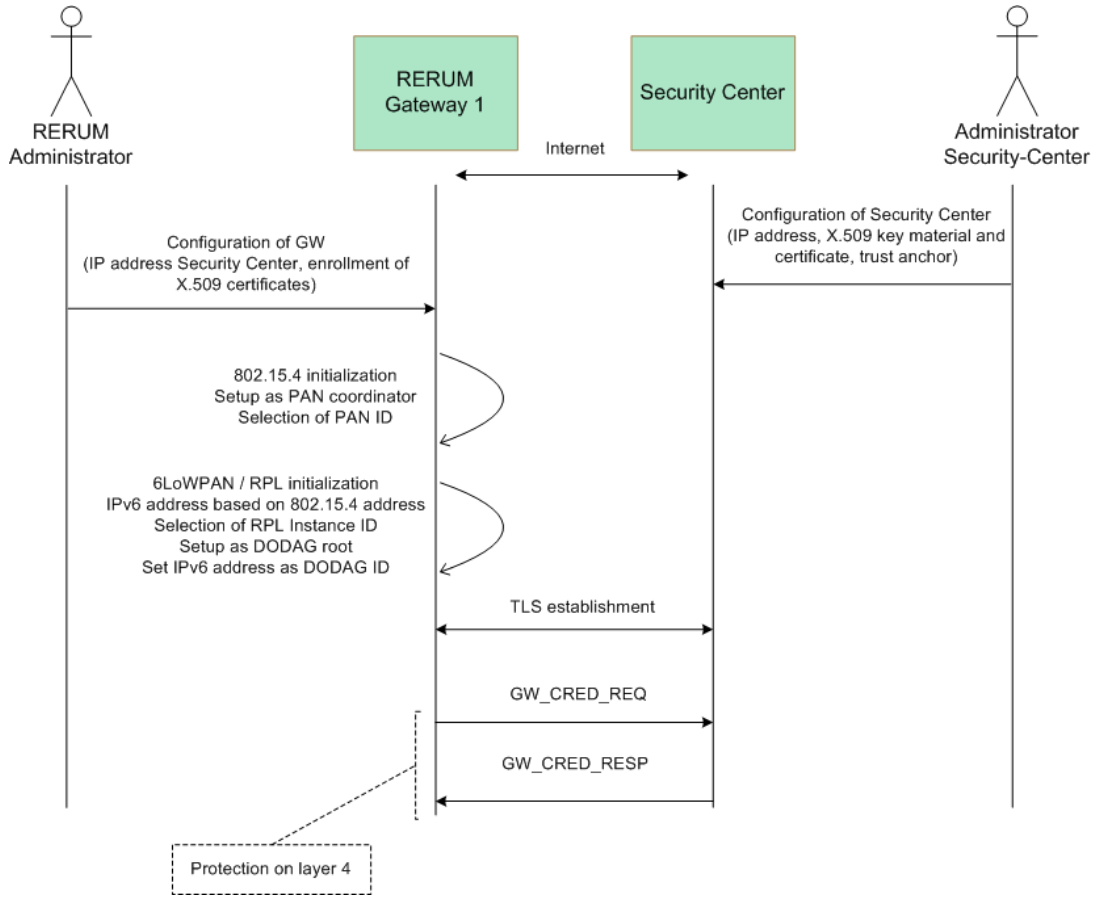


Figure 43: Initialization of RERUM gateway

While the RERUM gateway boots up, the IEEE802.15.4 layer and IP layer are initialized.

- The gateway acts as PAN coordinator and selects an unused PAN ID.
- The gateway initializes the 6LoWPAN IP stack and RPL routing protocol.
- The IPv6 address is based on the 802.15.4 MAC address according to the derivation function in [IEEE802.15.4].
- The gateway selects an unused RPL instance ID, sets up as DODAG root for routing of IP packets and the DODAG ID is set to the IPv6 address of the gateway.

After basic network initialization the RERUM gateway establishes a mutual TLS connection towards the Security Center based on the enrolled PKI credentials from the Security Center. All further messages between the gateway and the SC are protected on layer 4 (confidentiality, authenticity and integrity).

After successful establishment of the TLS channel the 802.15.4 network key is bootstrapped.

- `GW_CRED_REQ (GW_ADDR, PAN_ID)`
  The gateway requests a 802.15.4 network key from the Security Center.
- `GW_CRED_RESP (PAN_ID, NK_PAN)`
  The SC associates the ID of the gateway with the PAN ID, generates the requested network key sends it back to the gateway.

The network key `NK_PAN` will later be used for security on layer 2 (802.15.4) between RERUM Devices and the RERUM Gateway.

5.2.4 Distribution of the Join Key

The join key is a symmetric key and is the initial trust anchor between the RERUM Device and the Security Center. A join key has to be available in the RERUM device that wants to join a RERUM network. The security of the initial bootstrapping process is based on this join key. This key has to be configured/established/transported into the Security Center prior to the join procedure of a RD. In the following description different procedures are provided to configure this key in a secure and comfortable manner. These procedures are applied before the RERUM Device is connected to the 802.15.4 network.

5.2.4.1 Join Key Pre-Installed on the Device

The “pre-installation” of the join key on the RERUM Device can be done in different ways:

- The key is configured into a non volatile storage of the device during manufacturing as one step of the manufacturing procedure or
- The device provides a mechanism to generate a fresh key during the first start up of the device or on request of a user.

Independent of the way how the join key is generated in the RD the following actions have to take place:

- The join key of the RERUM Device and the corresponding device ID, e. g. the MAC address, has to be provided to the person who is setting up the RERUM network. For the following section

this person is called RERUM technician. Depending on the interfaces available at RERUM Devices there are different ways for doing this:

- Barcode: Additional to the key stored inside the device the key and the ID of the device are written as barcode on a label attached to or sent with the device. The RERUM technician has a barcode reader and stores the key and the corresponding device ID on his installation device like a smartphone, tablet, laptop.
- Physical limited hardware interface that allow to retrieve the join key, e. g. serial interface, memory card, USB, NFC, 802.15.4 with reduced range, simple display, LEDs, etc. Depending on the interface the retrieval of the join key is more or less comfortable. The key and the corresponding device ID on his installation device like a smartphone, tablet, laptop.

- The RERUM technician delivers the join keys/IDs of the RERUM Devices to be installed to the Security Center over a secure channel. This secure channel could be
  - a TLS connection to the SC from the RERUM technician's installation device,
  - e-mail transport of an encrypted ID and associated join key file to the SC over an unsecure channel or
  - storing the join key and RD ID on a storage medium like a USB stick or CD which is imported into the Security Center later on.

Figure 44 shows the transport of the join key and RD ID via a secure TLS channel from the installation device of the RERUM technician towards the Security Center. The end-to-end security association is established between the installation device of the RERUM technician and the Security Center. Regarding secure communication the RERUM Gateway in the following diagram is transparent but it has to do the routing from the RERUM network to the Security Center that may be accessible over the internet.

This installation method allows storage of the join key into the Security Center while installing the RERUM Device into the RERUM network.

- `NEW_RD_JK_REQ(RD_ID, JK_RD1_SC)`
  Authorization that the RERUM Device with RD_ID is allowed to join the RERUM network by using the join key JK_RD1_SC
- `NEW_RD_JK_RESP`
  Acknowledgement by the Security Center

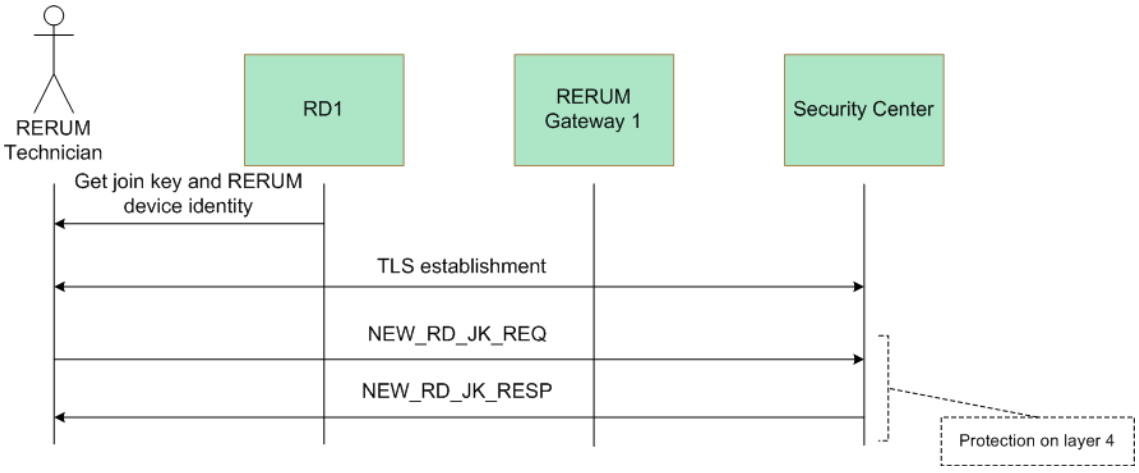


Figure 44: Pre-installed join key

This procedure grants access to a RD to join the RERUM network while the RERUM technician is installing the RD. The time that a join key can be used and may be misused is limited to the time the RERUM technician has authorized a RD to join and the time the RD has joined. On the other hand, each device installation requires a manual interaction with the device and requires a connection towards the Security Center at the premises.

In contrast to that and for Smart City applications in which an automated installation procedure has higher priorities, another installation method allows to collect the join keys offline, e.g. by a system integrator, and to store the pairs of device ID and join key on a storage medium. Afterwards all join keys are imported into the Security Center and the devices are authorized to join the RERUM network in a later step fully automated.

5.2.4.2 Join Key is Installed in the RERUM Device During Installation

In the following scenario the RDs are distributed without credentials stored inside the device during manufacturing and without the ability to generate the join key inside the device itself. In this case a join key has to be pushed into the device during installation of the RD before it can participate in a RERUM network. As in the pull method described in the previous sub section, the interface to push a join key is a physical limited hardware interface, e. g. serial interface, memory card, USB, NFC, 802.15.4 with reduced range.

The join keys can be generated in two different ways. Either

- join keys are generated offline, e.g. by a system integrator, and imported into the Security Center all at once or
- join keys are generated by the Security Center on request of the RERUM technician while installing a RERUM Device.

A RERUM technician who wants to install a RD needs to retrieve the device ID of the RD. Afterwards he requests a join key for that device and in the last step pushes the join key retrieved from the Security Center into the RERUM Device. As in the scenario above the end-to-end security association is established between the installation device of the RERUM technician and the Security Center. The RERUM Gateway has to do the routing from the RERUM network to the Security Center.

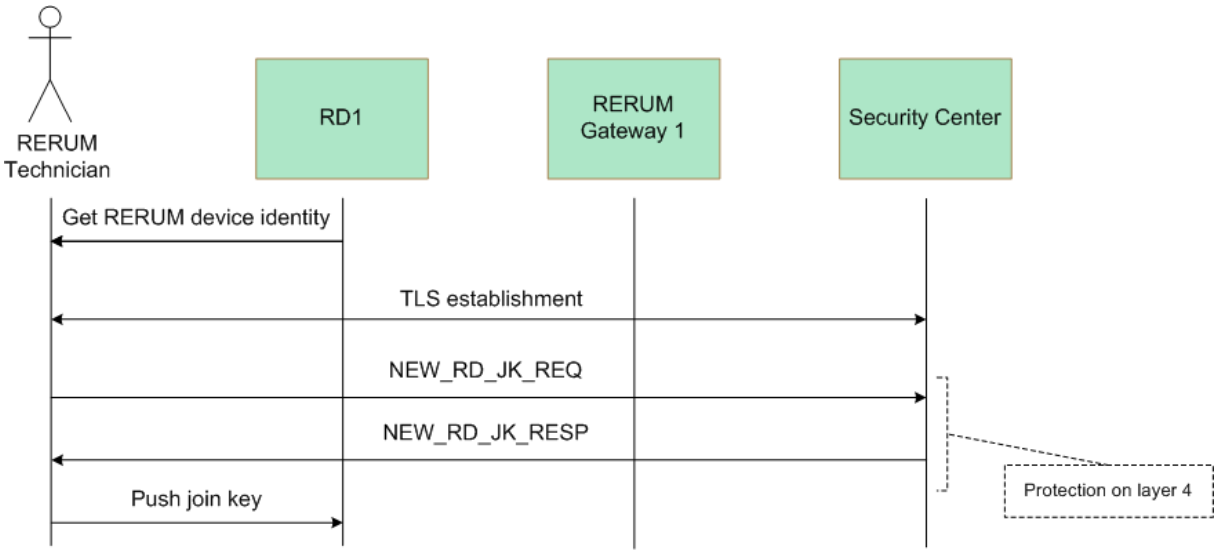


Figure 45: No pre-installed join key

- NEW_RD_JK_REQ(RD_ID)
  Authorization that the RERUM Device with RD_ID is allowed to join the RERUM network and requesting a join key
- NEW_RD_JK_RESP(RD_ID, JK_RD1_SC)
  Transmission of a join key to the RERUM technician. The join key is either generated freshly or offline generated keys are used.

5.2.5 Initialization and Bootstrapping of RERUM Devices

In this section the initialization and bootstrapping of RERUM Devices during installation of the RD into an RERUM network is described.

The assumptions here are that the initializations as described in the previous sections have been carried out:

- The RD knows its join key. The join key is established in the RD with one of the procedures described above.
- The gateway is up and running. Especially the GW knows the network key NK_PAN of the relevant 802.15.4 network (see above).
- The Security Center is up and running. Especially the SC knows the IDs of the RDs authorized for installation in the considered RERUM network and the corresponding join keys (see above).
- Already installed RDs are up and running. Especially the RDs know the network key NK_PAN for securing communication on 802.15.4. How this network key is distributed to the RDs is part of the bootstrapping described below.

At a high level the following steps are performed for initializing and bootstrapping a RD. These steps are performed automatically without manual interaction of a RERUM technician:

- Set up of the 802.15.4 network.
- Establishment of the network key NK_PAN for securing communication on 802.15.4 as well as symmetric key for secure communication between RD and SC (K_RDx_SC). K_RDx_SC is an individual key of each RDx.
- Set up of the IP layer (6LoWPAN). The messages on layer 2 (802.15.4) to set up 6LoWPAN and to establish routing information are secured by the 802.15.4 network key NK_PAN.
- If the RD needs additional credentials these credentials are requested/transported between RD and SC above layer 3. The communication is secured by K_RDx_SC.

In the following sections these steps are explained in more detail. Figure 46 shows the phase starting with initializing the 802.15.4 network. The result is the secure establishment of the 802.15.4 network key (NK_PAN) and the key for communicating to the SC (K_RDx_SC).

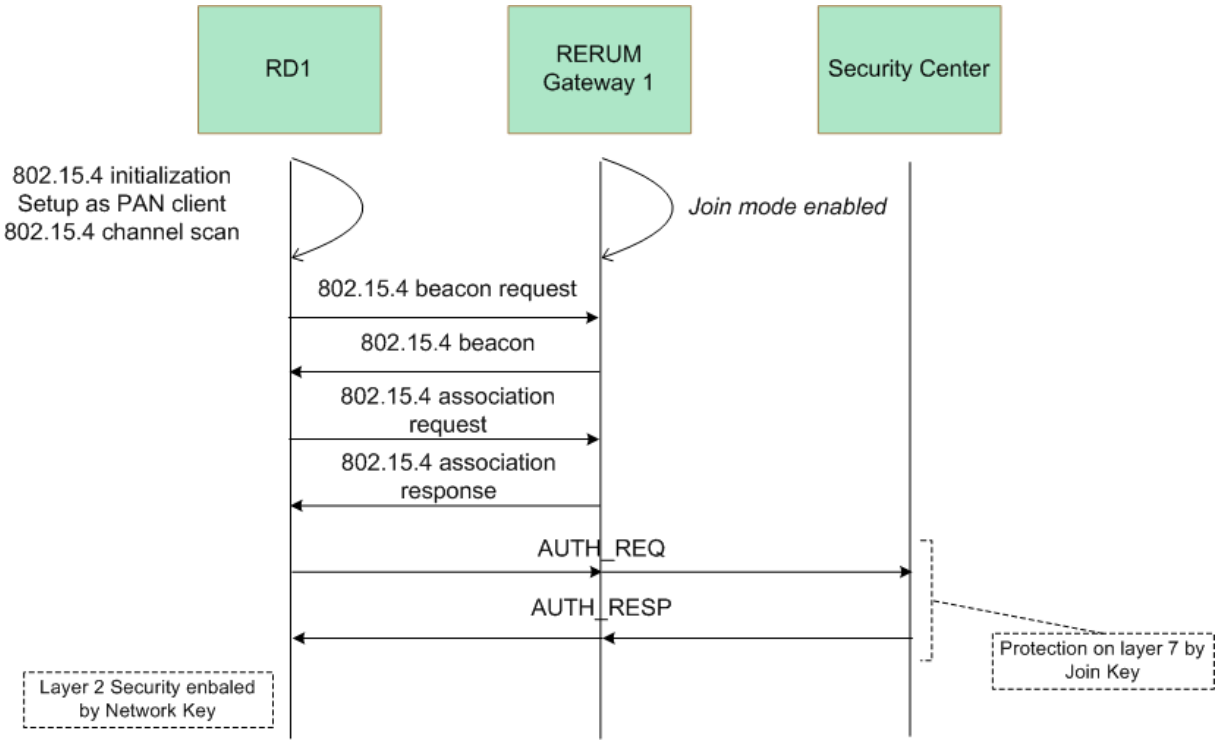


Figure 46: Bootstrapping 802.15.4 network key and the keys to communicate to GW and SC

The initial steps to connect to the 802.15.4 network are not protected:

- The RD tries to associate with 802.15.4 networks in range by sending a beacon request.
- The GW configured as PAN coordinator (or a coordinator) sends out the beacons.
- The RD sends an association request.
- The GW (or a coordinator) sends an association response.

At this point the RD is associated with the 802.15.4 network and can send layer 2 messages to the GW. The next step is to bootstrap keys for secure layer 2 communication and for secure end-to-end communication between RD and GW and between RD and SC.

- The RD sends a layer 2 message: `AUTH_REQ (RD1_ID, RD1jcount)`
    - The message contains the identifier of the RD and a join counter to prevent replay attacks.
    - This message is not protected on layer 2, but the message is end-to-end integrity protected between RD and SC by the join key.
    - The message is forwarded by the GW to the SC additionally secured on layer 4.
- The SC sends back: `AUTH_RESP (RD1_ID, GW1_ID, RD1jcount+1, NK_PAN, K_RD1_SC)`
    - The message contains the network key for the 802.15.4 network (`NK_PAN`), the RD-SC key (`K_RDx_SC`) and the incremented join counter.
    - The message is end-to-end integrity protected and, besides the identifier of RD1, encrypted between RD and SC by the join key.
    - Between SC and the GW the message is additionally secured on layer 4.

    - The GW forwards the message to the RD on layer 2 (the message is not protected on layer 2, but secured end-to-end).
- The RD decrypts the message and checks integrity. If successful the RD has now the keys:
    - NK_PAN for secure communication on layer 2
    - K_RDx_SC for end-to-end security between RD and SC

As just described, the messages AUTH_REQ and AUTH_RESP are not protected on layer 2. On the other hand it is reasonable to configure secure communication on layer 2 to protect all communication above layer 2, e. g. routing messages with RPL. If secure layer 2 communication is configured this would result in security errors on the RDs that are already installed, if an unsecured AUTH_REQ or AUTH_RESP messages has to be forwarded. Therefore the 802.15.4 stack has to look into the 802.15.4 payload if it is a unsecured 802.15.4 data message and has to decode the first byte of the message. If the first byte decodes to an AUTH_REQ or AUTH_RESP message, it has to be forwarded unsecured without generating a security error.

The subsequent steps to fully set up a RD are secured on layer 2 by the 802.15.4 network key (NK_PAN). The steps are (see Figure 47):

- Setup of the IP communication (6LoWPAN)
- Setup/update of routing information in the already installed RDs and the new RD

After these steps have been performed the RD is able to communicate on layer 3 (IP layer). The remaining configuration and bootstrapping is done based on IP communication. Especially the bootstrapping of additional security credentials is done by establishing a DTLS secure channel based on K_RD1_SC between the RD and the Security Center and using two commands:

- RD_CRED_REQ
    - The RD sends this command towards the SC to get all credentials that are configured for the RD.
- RD_CRED_REQ (RD1_cred)
    - The message contains all the credentials that are configured for the corresponding RD.

Either the RERUM Device is requesting a concrete type of credentials or it is requesting all credentials relevant for that device. At first the RERUM Device gets based on the RD_CRED_REQ command the necessary credentials to establish a secure DTLS channel towards the GW.

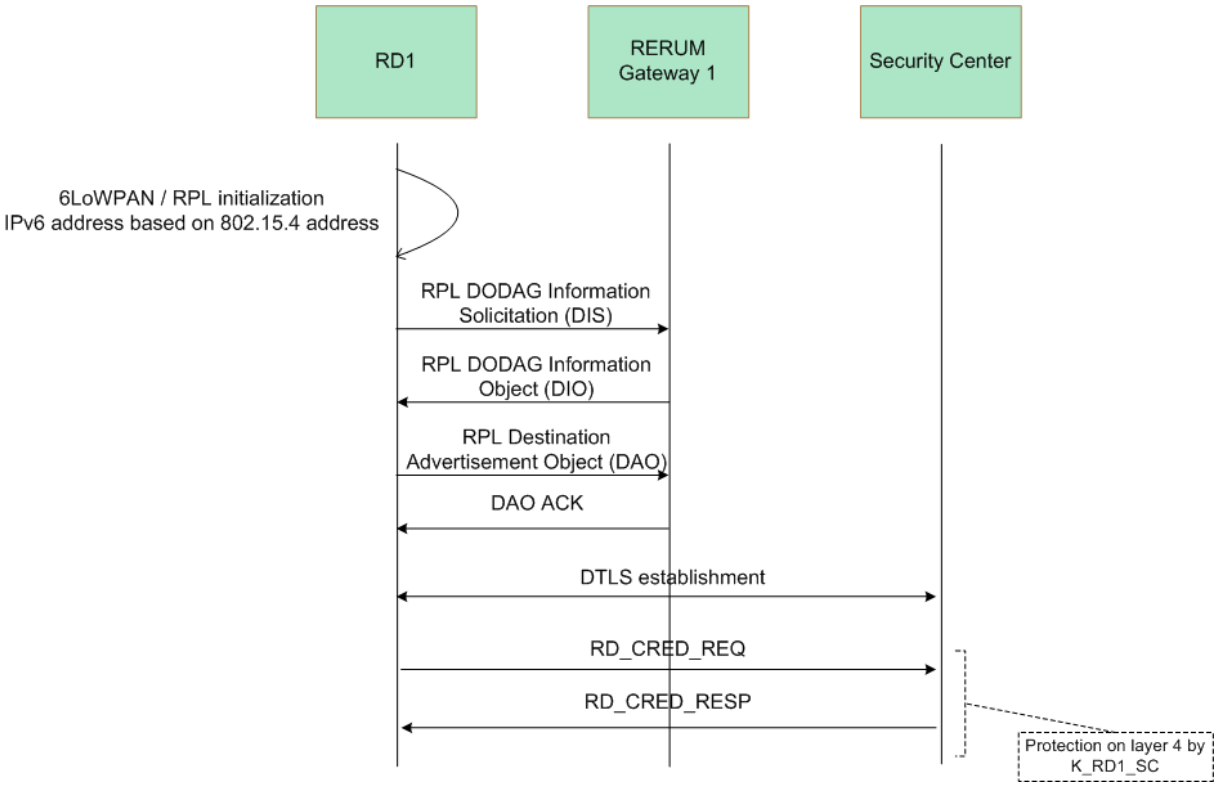


Figure 47: Setup of IP communication and bootstrapping further credentials

5.2.6 Credential Storage on RERUM Devices

Security credentials are sensitive data on which the RERUM security architecture relies on. If credentials are stored in clear-text in the non-volatile memory of RERUM Devices, these are vulnerable to be compromised. An attacker may dump the content of the non-volatile memory and tries to find potential credentials being used by a RD. Therefore, it is recommended to protect the confidentiality and integrity of stored security credentials.

It is out of scope of RERUM to define new credential storage types. The standard PKCS#12 [PKCS#12] is a commonly used key file container to store asymmetric and symmetric. A PKCS#12 key container is organized in SafeBags allowing encryption and integrity protection of sensitive credentials by another symmetric key. This symmetric key is generated out of a password by a password based key derivation function. During the first boot procedure a RERUM Device may generate a nonce to be used as password to protect the key store.

5.2.7 Relation to RERUM Architecture

The subsections of 5.2 detail the procedures and identify the collaborating components of the RERUM architecture for fast and secure network bootstrapping. Especially subsection 5.2.5 elaborates conceptually the Credential Bootstrapping Client and the Credential Bootstrapping Authority as introduced in the deliverable 2.3. Figure 48 from deliverable 2.3 depicts the location of these functional components and their relation to additional functional security components the Credential Bootstrapping Client/Authority builds on. For RERUM “Device Authentication client”, Integrity Generator/Verifier” and “Data Encrypter/Decrypter for data at transit” are provided by the secure communication functionality described in the “Profile DTLS” and the “Profile 802.15.4 Security” of Chapter 3. Credentials are securely stored by the “Secure Storage” component that provides “Data Encrypter/Decrypter for data at rest” and the “Integrity Generator/Verifier” (see Section 5.2.6 on Credential Storage on RERUM Devices)

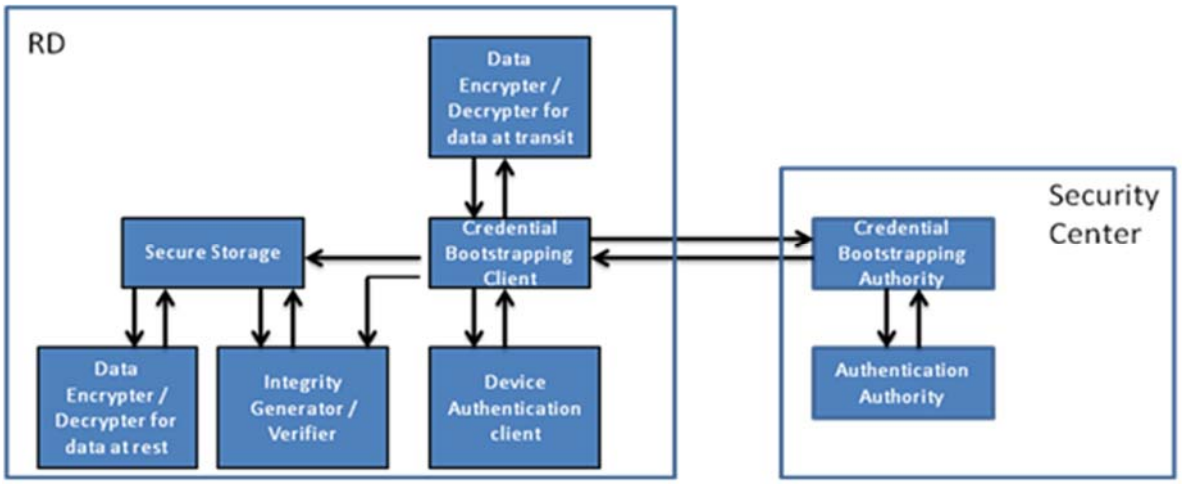


Figure 48: Credential Bootstrapping Client / Authority and related security components

5.3 Secure and Context Aware Dynamic Auto Configuration

The content of this section is related with the "SW Components Manager" component defined in the RERUM Architecture as shown in Figure 49:

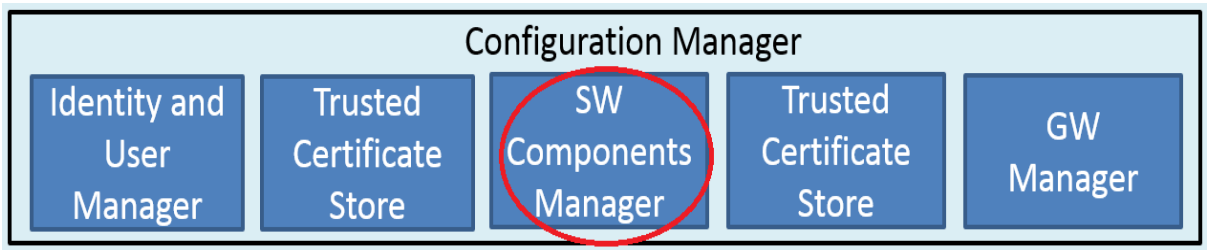


Figure 49: Configuration manager components from D2.3

5.3.1 RERUM Device Initial Configuration

The process to follow for check and get initial configuration for a new VRD is:

1. A RD is installed and connected to the Network.
2. The new RD is registered in the RERUM network (as described previous Section 0)
3. The GVO Manager (Register) adds this new device in the registry and adds context information about this device.
4. The Gateway Instance puts a message in the Message Bus noting that a new RD has been added to the network.
5. A SIEM Agent, which is listening the Message Bus, reads the new RD message and generates an event in SIEM format with the information available about the RD added.
6. The SIEM receives and stores the event from agent, detects that an action is required and reacts sending an alert to PRRS.
7. The PRRS registers the alert and executes the associated rule (previously defined by admin) of search appropriate firmware configuration for this Device and Context (additional context information will be requested to GVO Manager if needed). Finally sends information to OAP module to update the software installed in the RD if needed.

8. The OAP module receives the base firmware and the additional software components needed for that device to build them together and send the final built firmware to the RD Deployer.
9. The RD receives the firmware and installs it checking that is signed by a trusted authority and coming from a reputable supplier.
10. The RD sends a signal with the new firmware version installed ok to the MW (to the OAP Module?).
11. The OAP Module updates context information for the target RD in the GVO Register.

For installing new software components to the RERUM Devices, the OAP Module is the one in charge of building a firmware that fits with the characteristics of the target RERUM Device together with the software components previously selected by the PRRS.
Once the new firmware is built, it is sent to the target device following the specifications given by the communication module of the RERUM Device and, finally, the new firmware is deployed by the RERUM Deployer inside the RERUM Device.
The Firmware to deploy must be signed by a trusted authority and must come from a valid RERUM OAP Module.

5.3.2 Integration with a PRRS

The Internet of Things, per definition, allows devices to interact and work together. Moreover, marketing is permanently looking for a way to make different products work better together. The problem of this great goal is that you don't know what products are coming next, so you don't know how the future products might interact. For this reason is needed a way to add flexibility to the system, allowing the addition, reconfiguration and upgrade of devices that are part of the network.

The Platform for Runtime Reconfiguration of Security (PRRS) aims to cover this requirement of flexibility providing a way to maintain the devices up to date.

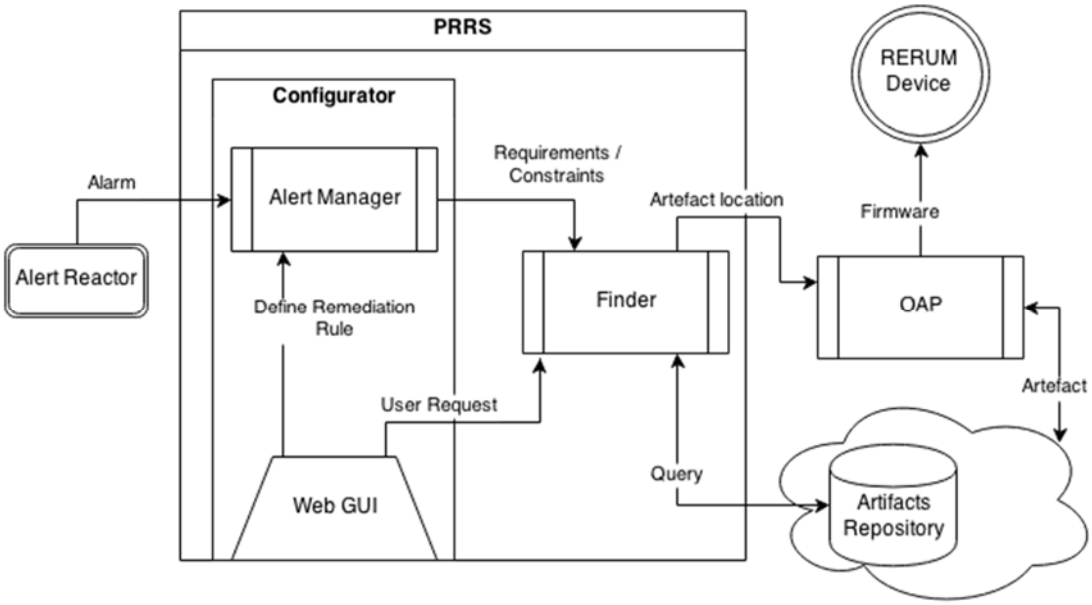


Figure 50: PRRS components and workflow

The components of PRRS are shown in Figure 50. The PRRS receive alerts in a predefined format and process them for react and take appropriate actions depending on the nature of the alert, the

remediation rules defined for that kind of alert and the context variables associated with the RERUM Device that throws the alert.

The artefacts to deploy can also be found manually using a PRRS interface for searching solutions in the repository of artefacts, introducing constraints and properties that we want to look for.

5.3.2.1 Security SW Component Selection

The PRRS bases its decisions on the context information gathered from incoming alerts and external monitors or knowledge bases, in this case from the GVO Registry (D2.3, Section 6.2.2) and the integrated SIEM (5.4.3)

The context variables will be defined in a configuration file indicating the sources for retrieve its values and the regular expression that matches with the given value. As an example:

```
{"contextVars": [
      {"name" : "temperature",
      "source" : "http://localhost:8080/PRRS-services/resources/requests",
      "format" : "/^\d+(,\d+)?$/"
      }
      {"name": "deviceId",
      "source" : "http://localhost:8080/PRRS-services/resources/requests/",
      "format" : "^\d+$"
      }
      {"name" : "OperatingSystem",
      "source" : "http://rerumserver/gvoregistry/resources/rerumdevices/",
      "format" : "/^Contiki|Linux|Windows/$"
      }
]}
```

Those context variables previously declared can be used to define rules in the PRRS Cofiguration Web Interface that is depicted in Figure 51.

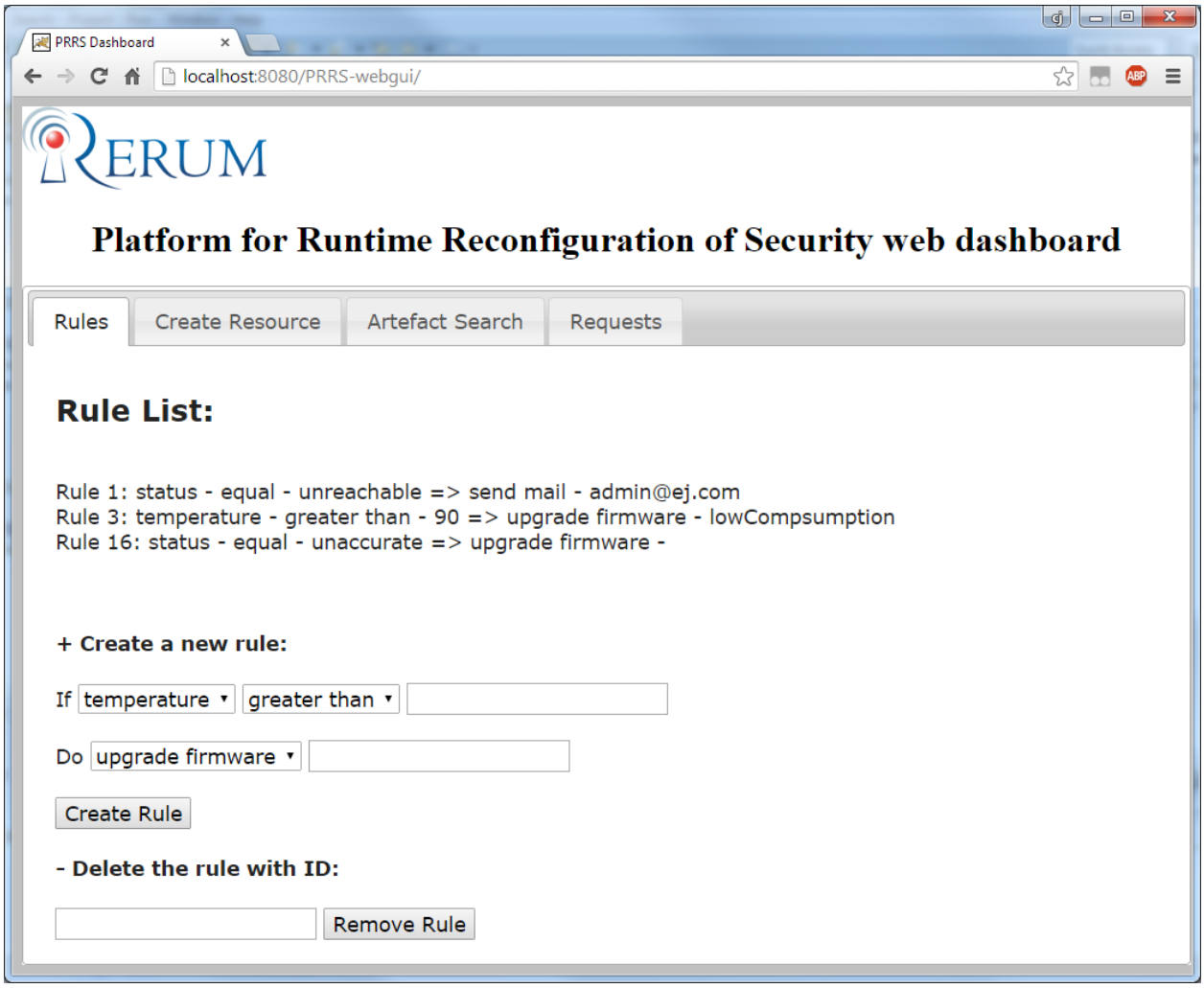


Figure 51: PRRS rule web editor snapshot

The syntax of the rules is being improved, by building a more powerful rule language to allow incorporating logic dynamically to the component instead of building an ad-hoc internal logic. It prevents re-writing code if the system context or monitors change and allows re-use of this component in future projects.

Once the rules are defined, they are checked every time the PRRS is alerted by any event. If any rule is met, taking into account the actual values of the context variables, the associated action will be launched.

5.3.2.2 SW Component Allocation

The software components available for enrich the RERUM Device's firmware with new functionalities or for solving concrete issues, will be stored in a secure and reliable repository. Each software component must have several meta-data labels associated, for example:

- `<ComponentName>`
- `<Developer>`
- `<Functionality>`
- `<Version>`

- <SecurityGoal>
- <SecurityMechanism>
- <OperatingSystemSupported>

Those labels could be referenced while the rules are being created for filtering which software component is suitable for the current use case that is being described in the rule.

When a rule is evaluated, the PRRS finder seeks in the software components repository, by filtering for the given context variables, and the labels that defines the components. When the search gives a result, it returns the URL of the software component. This URL is sent to the OAP component for building the new firmware with the new software component and taking into account the given context.

5.3.3 Over the Air Programming

OAP is a service based on reliable broadcast communication for challenging the issues of maintenance of a more or less complicated network. It is necessary to fix bugs, update codes and manage application requirement changes to build an adaptive system. The typical functionalities provided by an implementation of OAP technology that allows managing remotely the whole IoT network are:

- Discovering new devices.
- Recovering any device which sets stuck.
- Upgrade firmware versions of an entire network without physical access.

The OTA mechanism requires that the existing software and hardware of the target device support the feature, namely the receipt and installation of new software received from the provider. In RERUM this piece in charge of applying new firmware in the device is called RD Deployer.

5.4 Self Management and Self Monitoring Mechanisms

The components of the architecture defined in D3.2 related with this section are the Alert Processor and the Resource Monitor as shown in Figure 52:

Figure 52: Monitoring manager components from D2.3

Specifically the sub-section 5.4.3 is describing an implementation of an example of Alert Processor with a SIEM and the sections 5.4.1 and 5.4.2 are components that provide information about hardware resources of the RERUM Devices as part of a Resource Monitor as described in Section 6.8.2.1 of the architecture document D3.2.

5.4.1 Network Monitoring Mechanisms

5.4.1.1 Introduction

Self-management and self-monitoring techniques, from the network perspective, usually fall in the broader category of network management. For RERUM, network management is of high importance, as it can ensure network high availability mainly through energy saving and QoS support. As described in RERUM Deliverable D2.3, one of the RERUM's main components is the Network Manager that will support a number of functionalities such as network monitoring, spectrum management, clustering, mobility management, neighbour discovery, etc. Among those functionalities, here, we focus on the *network monitoring* one, which will make feasible a network-wide inspection mechanism.

RERUM envisions a reliable and resilient IoT architecture for smart cities. The fundamental blocks of such architectures are wireless sensor networks (WSNs) that can consist of hundreds or even thousands of miniature sensors. These devices are used in numerous applications like agriculture [XPHZ13], military [WS++13], health care [KLS++10], traffic management [PN++S12], etc., as they are capable of collecting a diverse type of data like humidity, ambient temperature and light, acceleration, location, etc. However, these devices are severe resource constrained in terms of memory, CPU, and storage. A typical sensor device might not contain more than 8KB of memory. Furthermore, in many scenarios, the sensors operate in unattended or even in harsh environments where human intervention (e.g. battery replacement) is not possible. Moreover, WSNs are susceptible to harmful interference as they mainly operate in the ISM band that is heavily overcrowded. In such scenarios, network life-time prolonging is of paramount importance. For these reasons, a network monitoring scheme is of key importance as it can guarantee, if properly designed and deployed, decent network operation and it can extend network's life-time as much as possible.

5.4.1.2 Network Monitoring System Requirements

A network monitoring system (NMS) designed for the resource constrained WSNs should have several characteristics. First of all, *lightweight operation* is of key importance for this system to be deployed in the sensors; therefore, it should not require too much CPU and memory resources for proper operation.

Another important characteristic a NMS should have is *fault tolerance* against failures either at the network or the sensor level. At the network level, significant packet loss can occur in the network due to various reasons such as harmful interference or protocol inefficiencies. Moreover, at the sensor level, a number of sensors may become inoperable due to, for example, battery depletion. NMS should be robust against these failures and being able to detect the potential failures even if significant information is missing (e.g. when an out-of-band mechanism is used).

In general, a WSN is highly dynamic, in the network context, as a number of sensors may join/leave the network or become inoperable due to energy problems. NMS should be *flexible* and adapt accordingly to the specific network states.

As mentioned previously, a WSN for IoT purposes can utilize hundreds or thousands of sensors. An efficient NMS should be *scalable* in order to operate efficiently as the network size increases.

Finally, although the proliferation of the sensor systems has been remarkably the last few years, medium-cost sensors that could be massively deployed in the IoT domain, still lack adequate memory size. A NMS should have *minimal storage requirements* for use in those sensors.

5.4.1.3 Network Monitoring Schemes

Several monitoring schemes have been proposed in the literature. They can be classified as centralized, distributed of hierarchical that perform proactive or reactive monitoring [LDO06]. The centralized schemes (e.g. [SKLS05], [TC05], [RKE05], [TURON05]) assume there is a single controller (sink or server) that periodically collects information from the sensors and infers regarding network problems. Information collection can be performed using different modes like: (i) event-driven, when information flows to the controller after an event has occurred, (ii) polling, when the controller requests information from a sensor or from a group of sensors, and (iii) hybrid, which is a combination of the event-driven and polling schemes. A major advantage of the centralized schemes is that the sensors do not have to perform any intensive tasks. They exclusively collect and transmit the monitoring information to the controller that performs the most intensive tasks of network management. However, in these schemes, excessive traffic is generated within the network, and valuable network and sensor resources (e.g. bandwidth, battery) are consumed. Also, interference increases leading to significant packet loss. Furthermore, the controller is a single-point-of-failure for the network management system.

On the other hand, in the distributed network monitoring schemes (e.g. [RAYA05], [BS03]), major monitoring tasks are assigned to the sensors, so no excessive traffic is generated in the network. Usually, the sensors monitor themselves, as well as their one-hop neighbours, inferring about potential problems. This means that the sensors have to assign resources (CPU, memory, storage) for running these tasks. Major disadvantages regarding the distributed monitoring schemes are that the scarce sensor resources may not be adequate to run these tasks, creating a two-fold problem: (i) sensors become inoperable to perform any tasks, and (ii) the NMS may be negatively affected when a numbers of sensors do not functionally operate. Another disadvantage of those schemes is that they are complex and very difficult to manage.

In order to address the limitations of the centralized and distributed schemes, several others have considered the deployment of local controllers (e.g. cluster heads in a clustered WSN) that collect monitoring information from a limited part of the network (cluster). These are known as hierarchical network management systems (e.g. [DBN04], [KH03]). Here, mainly by using an appropriate cluster algorithm, the WSN is partitioned into clusters, and then, for each cluster, a sensor with adequate resources is elected as the cluster head (CH). Network management tasks can execute on the sensors, as well as on each CH. Furthermore, CH is responsible to convey management information from its cluster to a global server (and vice-versa).

Another distinction between the network management systems can be performed according to the approach followed for monitoring and control [LDO06]. *Passive monitoring* refers to schemes where information about the network states is collected. Later, offline analysis may follow for detecting issues related to network stability and sensor failures. *Fault detection monitoring* [HL06] refers to information collection for detecting if failures within the network have occurred. *Reactive monitoring* [KH03], [FRL05] involves the process where data is collected from the network and based on their processing; decisions are taken for adaptively re-configuring network's parameters. On the other hand, *proactive monitoring* [SKLS05], [RKE05] refers to information collection in order to detect past events and to predict future ones.

5.4.1.4 RERUM Network Monitoring

As mentioned previously, the network monitoring tasks are assigned to the Network Manager entity of the RERUM architecture given in RERUM Deliverable D2.3. We plan to adopt the hierarchical type of network monitoring as it has several advantages over the centralized and distributed approaches. For this reason, we will consider a clustered-based architecture where the network monitoring and management tasks will be assigned to both the sensors and the CHs (Figure 53).

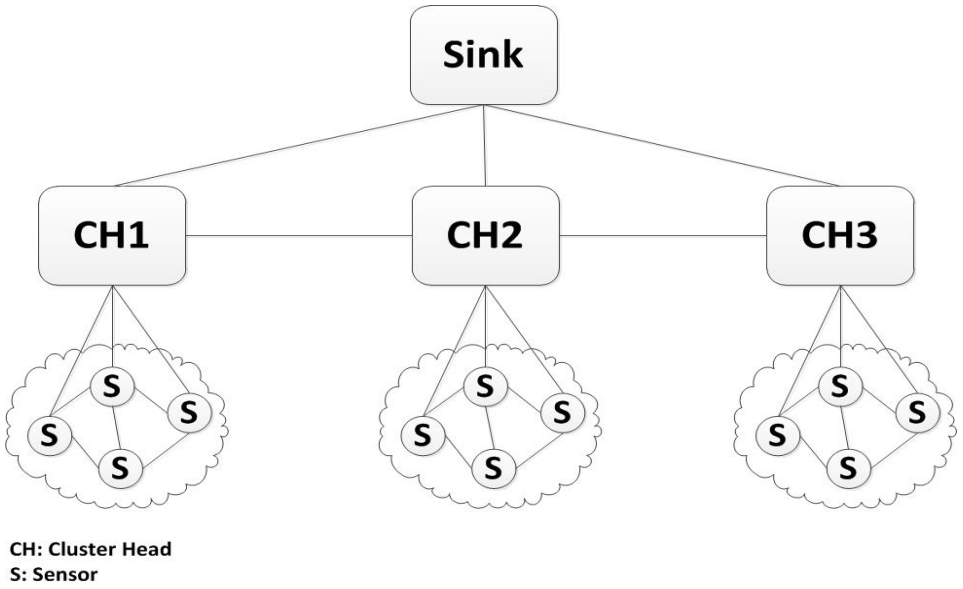


Figure 53: Hierarchical network management and monitoring

For the network monitoring, we will follow a two level approach. At the sensor level, several information regarding various parameters (at different layers) will be collected. We aim to explore the feasibility for collecting the following information:

- **Receive-Signal-Strength Indicator** (RSSI) available at the physical layer. RSSI gives an estimation of the signal (RF) power received at a receiver when this signal is transmitted by a specific transmitter. RSSI depends on the distance between the receiver and the transmitter, the transmission power of the transmitter, the antennas of the communicating parties, and the multipath effects in that area.

- **Link-Quality-Indicator** (LQI) that is available for the IEEE 802.15.4 devices. LQI is superior to RSSI as it has a strong correlation with the packet loss, so it can more reliably express the wireless link's quality. However, RSSI and LQI when used together can enhance the performance of the several algorithms (e.g. [HK12]).

- **Packets with Cyclic-Redundancy-Check (CRC) errors.** This information can be available at the MAC layer and can be used to detect harmful interference caused by transmitters that emit energy within the same channel ([F++A12]).

- **IP-layer related statistics** like the number of received packets, the number of transmitted packets, the number of the forwarded packets, the number of the dropped packets, packets dropped to various reasons (wrong IP version, wrong IP length, checksum errors, etc.).

- **Transport-layer related statistics** like the number of the received, sent, and forwarded packets; packets with errors (bad checksum, etc.).

The following table summarizes the network related information that can be collected by an RD:

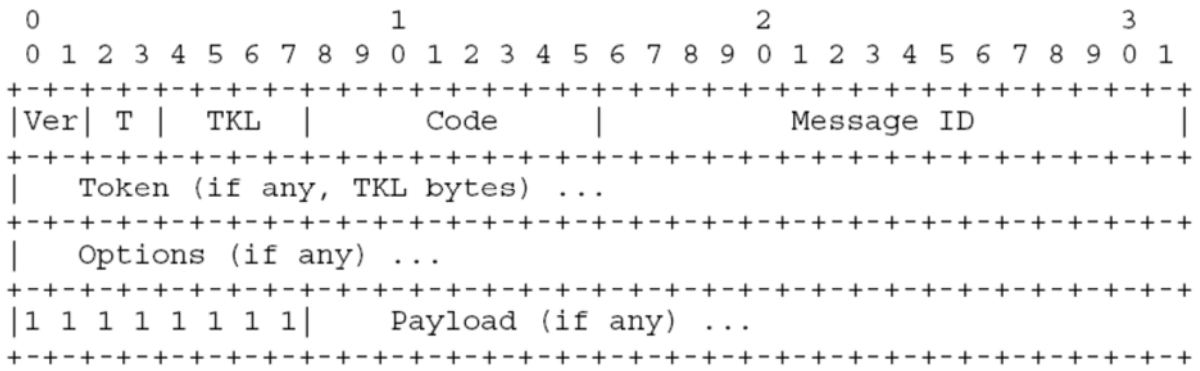
Table 2 Network Monitoring Information

Network Layer	Information
PHY	RSSI, LQI
MAC	CRC errors
IP	Dropped, received, and transmitted packets; packets with errors; type of errors
Transport	Dropped, received, and transmitted packets; packets with errors; type of errors

5.4.1.5 Protocol Definition

In Section 5.4.1.4 we described the network statistics that can be collected in each RD, for use within the context of the RERUM network monitoring scheme. The collected data will be sent to the Alert Processor (described in RERUM Deliverable D2.3); however not directly, but through the Security Information and Event Management (SIEM) component, described later in Section 5.4.3. For this reason, an appropriate communication protocol has to be defined.

Here we propose the use of the Constrained Application Protocol (COAP), as it has a number of attractive characteristics. COAP is software protocol intended to be used by resource constrained devices. This is an application layer protocol that can run over UDP, and can optionally support DTLS. COAP's message format is shown below [COAP14]:



On top of COAP we will define a suitable protocol (e.g. using JSON) to efficiently represent the network monitoring statistics.

5.4.2 On-Device Resource Monitoring

To complement network monitoring mechanisms described in Section 5.4.1 Network Monitoring Mechanisms, RERUM will also provide the ability to monitor on-device resources that can provide indications about the correct operation of the hardware, as well as to assist with the generation of alerts in cases of hardware faults.

Monitoring of on-device resources will use the same protocols and messages as those described in Section 5.4.1.5. Thus, on-device resources will be reported inside CoAP messages. In order to be able

to monitor their health and correct operation, RDs will expose the following resources: To monitor the health of each RD, Each device will expose two resources:

1. On-chip temperature. An alert will be generated if the value of this resource exceeds a configurable threshold.
2. Input voltage. For the CC2538 System-on-Chip [TI13], the absolute maximum input voltage is 3.9V, with a recommended maximum of 3.6V [TI14]. If input voltage exceeds these operating recommendations, an alert will be generated.

Both those metrics are available on all devices powered by the CC2538 System-on-Chip and will thus be available on Zolertia's Re-Mote platform. To expose those metrics as CoAP resources, we will adopt the path template and Resource Type recommendations of the IP for Smart Objects (IPSO) alliance. They are documented in the "IPSO Application Framework" [ZC12] and presented in Table 3.

Table 3 On-Device Resources, their Paths and Resource Types

Resource	Path	Resource Type
On-chip Temperature	"sen/temp"	"C"
Input Voltage	"sen/vdd3"	"mV"

Depending on the device type and capability, it may be possible to add more resources for device health monitoring. For example, if a device is equipped with an ambient temperature sensor, an alert can be generated if the temperature exceeds the device's recommended operating conditions. For example, the recommended operating ambient temperature range for the CC2538 is between -40 and 125 ^{o}C [TI14], and this information can be used to generate events accordingly.

5.4.3 Integrating a System Information and Event Management

The analysis of the information gathered from Devices and Network, allows the RERUM system to control and manage what is happening in the RERUM environment. For this reason we include in the system a Security Information and Event Management (SIEM) component that can provide a real-time analysis of the generated events, visualisation of events evolution through a console and also storage for a later forensic analysis and reporting of log data as shown in Figure 54.

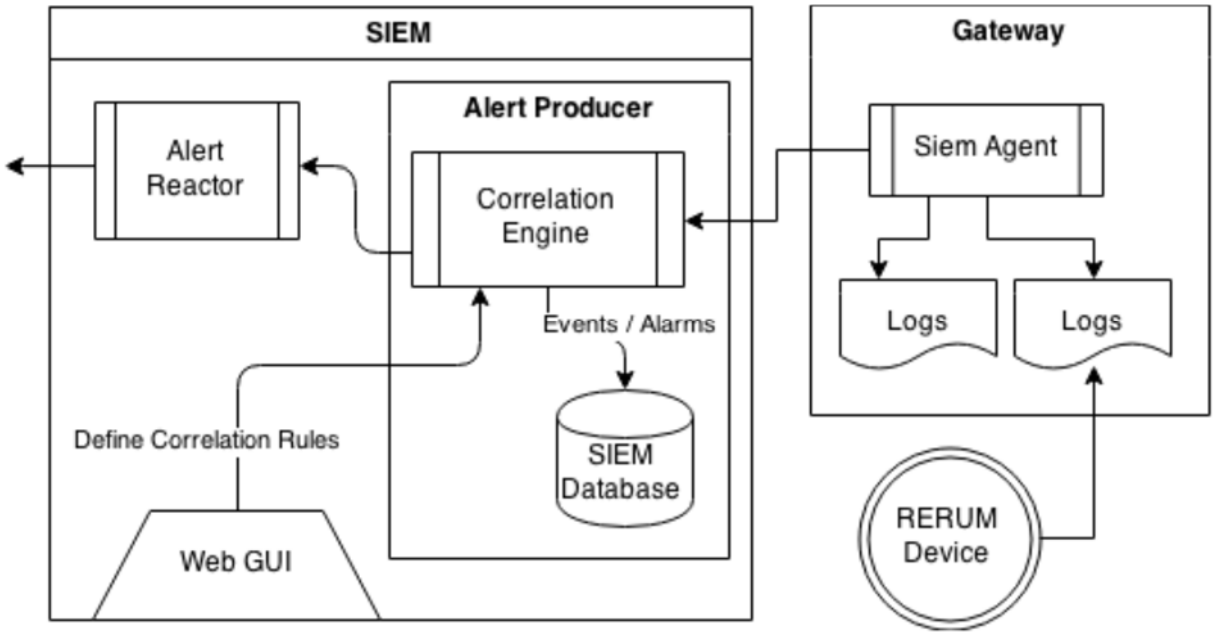


Figure 54: SIEM components and workflow

The SIEM agents installed in the context of RD, let's say a RERUM Gateway, collect information by parsing logs containing the information generated by Network Inspection (5.4.1) and Resource Inspection (5.4.2). Once the events are formatted by context agents, they are sent to the SIEM server where they are stored and analysed to detect anomalous behaviour as shown in Figure 55.

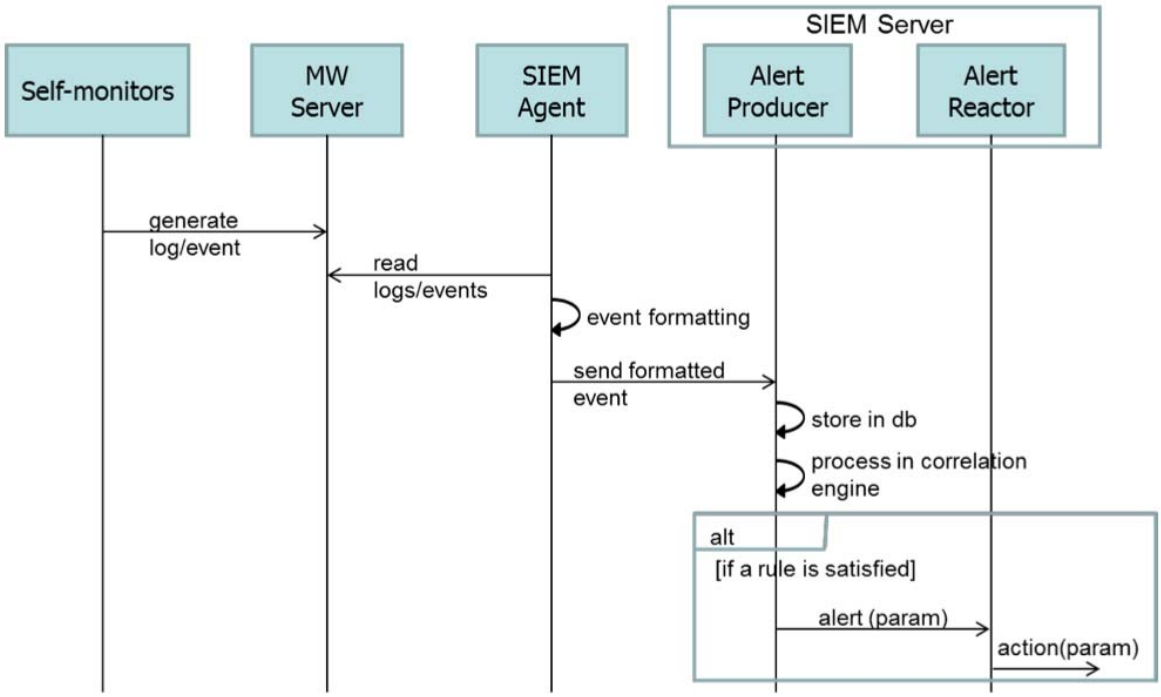


Figure 55: Events sequence diagram

5.4.3.1 Triggering Security Alarms

The SIEM Agents are installed in the RERUM servers where the activity of the RERUM Devices is being logged. They are in charge of collecting the events generated by RERUM monitors and forward them to the SIEM server. In the next sections is described how the generated events are manipulated and analysed for generating Security Alerts. The SIEM used for RERUM is based in OSSIM (Open Source Security Information Management), which is one of the most widely used Open Source SIEM. Therefore most formats, protocols and configuration have been adopted from this OSSIM implementation.

5.4.3.1.1 Event Collection

The event collection is one of the first things required in a SIEM. Data collection is firstly done as a result of the Agents installed in Sensors; the concept of 'Sensor', in the SIEM context, refers to a server that registers the activity that we want to analyse. Each server can receive data from different sources, but data collection is only possible from:

- Agents: These are the main sources for incoming events.
- Other Server: This is only possible in multi-level architecture.

In the RERUM context, the SIEM core will receive events coming from different sources like the MW Message / Event Bus or the Network Monitor. Consequently, our main goal here is to describe the specification of the SIEM agents that collect the incoming events to be processed by the SIEM.

Agents:

The SIEM agents are the components responsible for collecting all the data sent by the various devices existing on the network, in order to subsequently send them to the SIEM server in a standardized way.

The agents are installed on the sensor machines (for example a RERUM GW Instance or another server in the RERUM Network), normally one per machine although it is possible to install more than one if necessary. This will normally occur only in multi-level environments, where one machine with several agents can be sending information to various different servers, each from different devices.

The way in which the agent receives the data (which will then be converted into events for the SIEM) that it is going to send to the server is by means of reading a log file in most cases. Specifically in RERUM will be read one log generated by the Resource Monitor that receives the information from the final device and another log written by a RabbitMQ client that is listening service related data from the MW Event Bus. The port for communication between the Agent and the Server can be configured in the file `/etc/ossim/agent/config.cfg`.

Each of the events received by the SIEM server has always been processed beforehand by an agent in order to standardize them. The point of standardizing events prior to sending them to the server is so that the latter can deal with these events equally and so that storage and processing is simpler and more coherent.

For any device from which one wishes to collect data a **plugin** has to have been created beforehand so that SIEM is capable of processing it. This is achieved thanks to the creation of a plugin which basically consists of a series of regular expressions matching with a list of fields that allows the event type being produced to be unambiguously identified.

Plugins:

Plugins are configuration files defined in the Agent to analyse and standardize the information from a data source. Plugins are used to improve the collection capabilities of the Sensors and to indicate to

the system how to understand and to collect events generated by each application and device (data sources) that we are interested in. Once this has been standardized it is passed to the remaining functionalities of the Agent in order to be sent to the SIEM server in the form of an event.

In OSSIM there are two types of plugins:

- **Detectors**: Their job is to read from the logs that store the devices and to standardize them so that the Agent can send them to the OSSIM server. Detector plugins passively read a file or socket and send events upon pattern-matching lines.
- **Monitors**: These plugins will receive a question from the OSSIM server and send it to the corresponding tool; then as they obtain the reply, let the server know whether it agrees or not with what it has asked, for example if a specific `host:port` is `open`/`filtered`/`closed`. In RERUM is not planned to use this kind of plugins.

In general, each of the plugins can read and send data from a specific data source identified by its *plugin_id* and each event type belonging to that plugin is identified as its *plugin_sid*.

The plugins consist of two basic files, one with its configuration, and another with the information that the OSSIM server needs in order to correlate the events subsequently. In order to create a new plugin, it will only be necessary to create these two files as specified in the OSSIM documentation. One of the most important parts is to create a regular expression which must match with each line in the log file of the device for which we are creating the plugin. Both the server and the plugin have to agree on what each *plugin_id* and *plugin_sid* of each event means; both files are therefore inseparable and it is essential to have both in order for the plugin to work.

In the RERUM scope news plugins are required to read events coming from the Network Monitor or from the MW Bus.

Collection Methods:

There are several ways of collecting information in OSSIM and it is important to know which ones will be used in order to configure the agents and the plugins required to process the incoming data. The most common ways are:

- **Syslog**

  The device from which the logs wish to be extracted can inject information directly into the syslog of an OSSIM sensor. An agent will be active in this sensor to read from this syslog, and will standardize the events so that they can be sent to the server on which it depends.

- **SNMP**[53]

  An agent can receive events in SNMP format. Anyway, in order to receive them from any device, it will be necessary to install in the sensor, which is going to receive the data, additional software to establish the connection and make the sensor be able to understand this protocol. This software is available on SNMP Sourceforge web site.

- **Log Files**

  In the same way as with Syslog and SNMP, an agent can be configured in order to read from any log file once a dedicated plugin has been configured for this purpose.

- **Osiris: Unix HIDS**

[53] Simple Network Management Protocol Internet protocol for managing devices on IP networks

Osiris is a Host Integrity Monitoring System that periodically monitors one or more UNIX hosts for change. It maintains detailed logs of changes to the file system, user and group lists, resident kernel modules, etc.

It is possible to define both an agent and a plugin to extract information from UNIX machines by accessing Osiris stored information.

- **Snare: Collecting from Windows**

  Snare is the method OSSIM uses to extract information from a windows box. Each Windows host with snare agent installed must be able to send UDP port 514 data towards an OSSIM sensor. Then Windows events are normalized into the OSSIM nomenclature and sent to the ossim-server.

- **FW1LogGrabber: Collecting from Checkpoint FW-1**

  It is also possible, by installing in the sensor machine some additional software (available as part of Checkpoints OPSEC API) to download the logs from the Checkpoint FW-1. These logs, once downloaded and stored in the sensor hard drive, will be read from a plugin, exactly equal as the other plugins in the Agent.

5.4.3.1.2 OSSIM Event Description

OSSIM defines four types of events that are recognized by the server:

- Normalized event
- Mac event
- OS event
- Service event

The events received will be treated in a different way depending on the type of data. Plugins should parse events from different sources to these standardized ones, typically to the first of them as the other three are dedicated for special situations. Consequently, any developer who wants to implement a new plugin in compliance with the SIEM provided by the RERUM platform should take into consideration this event description.

In the RERUM SIEM, only OSSIM normalized event are considered for its parsing and later correlation to generate alarms. The fields of which the standardized event consists are detailed in the table below:

Field name	Description
Type	Type of event: detector or monitor
Date	Date on which the event is received from the device
Sensor	IP address of the sensor generating the event
Interface	Deprecated
Plugin_id	Identifier of the type of event generated
Plugin_sid	Class of event within the type specified in plugin_id
Priority	Deprecated
Protocol	Three types of protocol are permitted: TCP, UDP, ICMP
Src_ip	IP which the device generating the original event identifies as the source of this event
Src_port	Source port
Dst_ip	Ip which the device generating the original event identifies as the destination of this event
Dst_port	Destination port
Log	Event data that the specific plugin considers as part of the log and which is not accommodated in the other fields. Due to the Userdata fields, it is used increasingly less
Data	Normally stores the event payload, although the plugin may use this field for anything else
Username	User who has generated the event or user with whom it is identifying mainly used in HIDS events
Password	Password used in an event (HIDS events)
Filename	File used in an event, mainly used in HIDS

Userdata 1 to 9	These fields can be defined by the user from the plugin. They can contain any alphanumeric information, and on choosing one or another, the type of display they have in the event viewer will change. Up to 9 fields can be defined for each plugin

This event structure is followed for compatibility reasons, nevertheless all those fields could be null, so they may be used as needed. Typically the USERDATA fields are used as ad-hoc data that don't fit into the rest of fields.

5.4.3.1.3 Event Correlation

Event correlation is a way to create complex rules that imply more than one event, adding certain conditions. For instance, a naive solution for a failed login event would be just adding an OSSIM policy that detect this failed login event, but this means that an administrator will have to deal with a lot of false positives from genuine users that just forgot/mistyped their password. Also, in a big network it is not unusual that the number of these events can reach hundreds or thousands in an hour or a day.

To illustrate here the creation of these correlation directives in the SIEM we are going to describe a *misuse* case where there will be a big number of failed login attempts with a particular username and coming from the same machine.

From this we can create a tree of events for the misuse case that will be the structure of the correlation directive:

- **User Bob fails to authenticate** (root of the tree and start of the correlation)
    - **User Bob authenticates correctly** (user just forgot/mistyped the password, but he logged in the second attempt)

      This will be a leaf of the tree and will terminate the correlation rule.
    - **User Bob fails to authenticate more than 5 times** (user keeps entering the password incorrectly)

      This will be the other leaf of the tree but also will trigger the alarm because there is a big chance that an intruder is trying to steal Bob's credentials.

Once a correlation rule is met and it triggers an alarm, this alarm is stored in the SIEM database and its structure is basically an AlarmID and the list of Events that caused the alarm with all events' fields described before.

5.4.3.2 Reacting to Security Events

Policies and Actions

We can define actions to automatically respond to events that match the policy in order to tackle the misuse case presented. In OSSIM there are three types of actions:

- **Open a ticket**

  You can create and assign a new ticket to a given user in order to delegate the function to investigate a certain problem to someone with expertise in that matter. When that user logs

into the Service Level SIEM interface, he/she will see a new ticket in the list and can decide what actions to take;

- **Send an email**

  You can send an email to a given user. The email is fully customizable and you can include parts of the event with the use of keywords (source/destination IP, port, date, type, priority, etc.) and

- **Execute a command**

  This is a rather powerful option because it allows the user to execute a BASH script with the event parameters as input and do virtually anything that can be done with a console, both locally and from a remote computer.

Below you can see how to define an action in the OSSIM web interface. Note that all the fields defined in the event are available for use as variables in the action for its execution in runtime as shown in Figure 56.

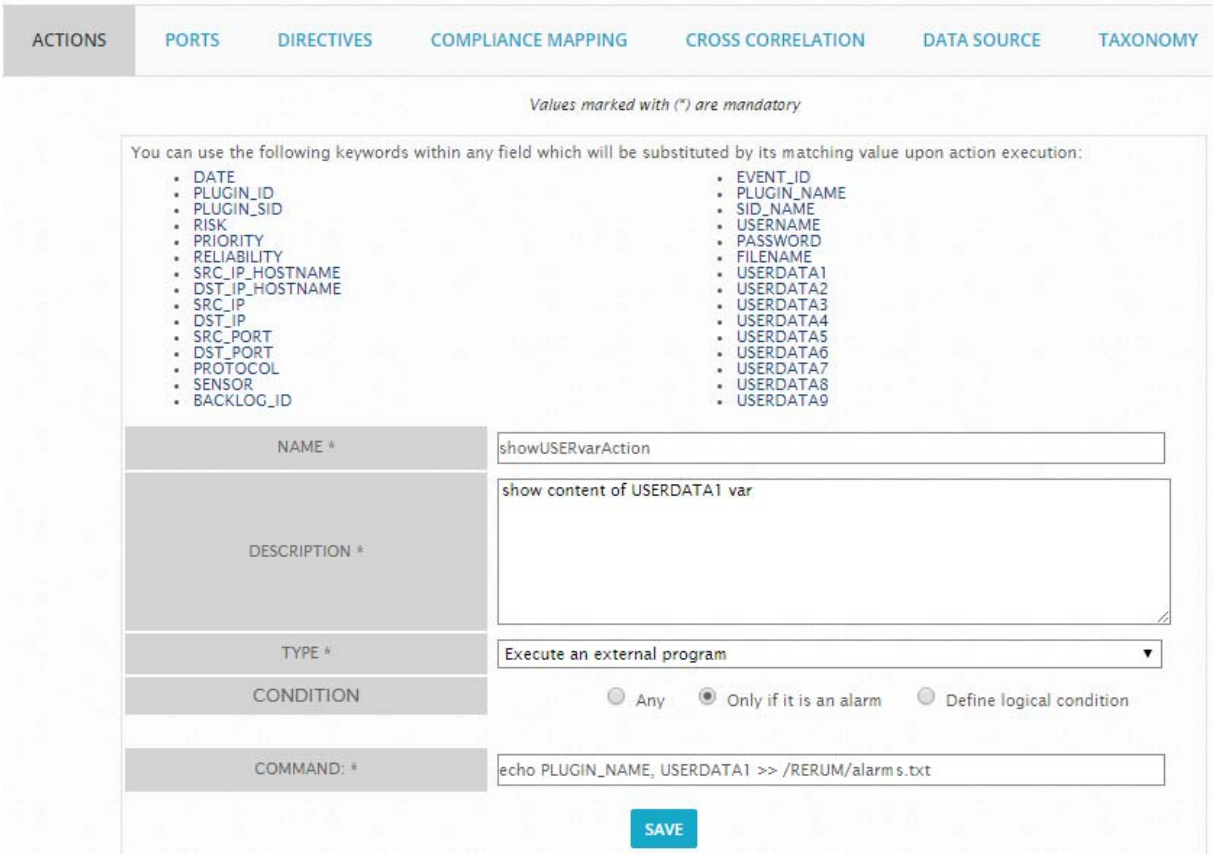


Figure 56: SIEM Action designer

Finally we can use the defined Action inside a Policy that will be the start point of the correlation process, filtering by a concrete Event Type, which is called 'RERUM_Alarm' in the example shown in Figure 57.

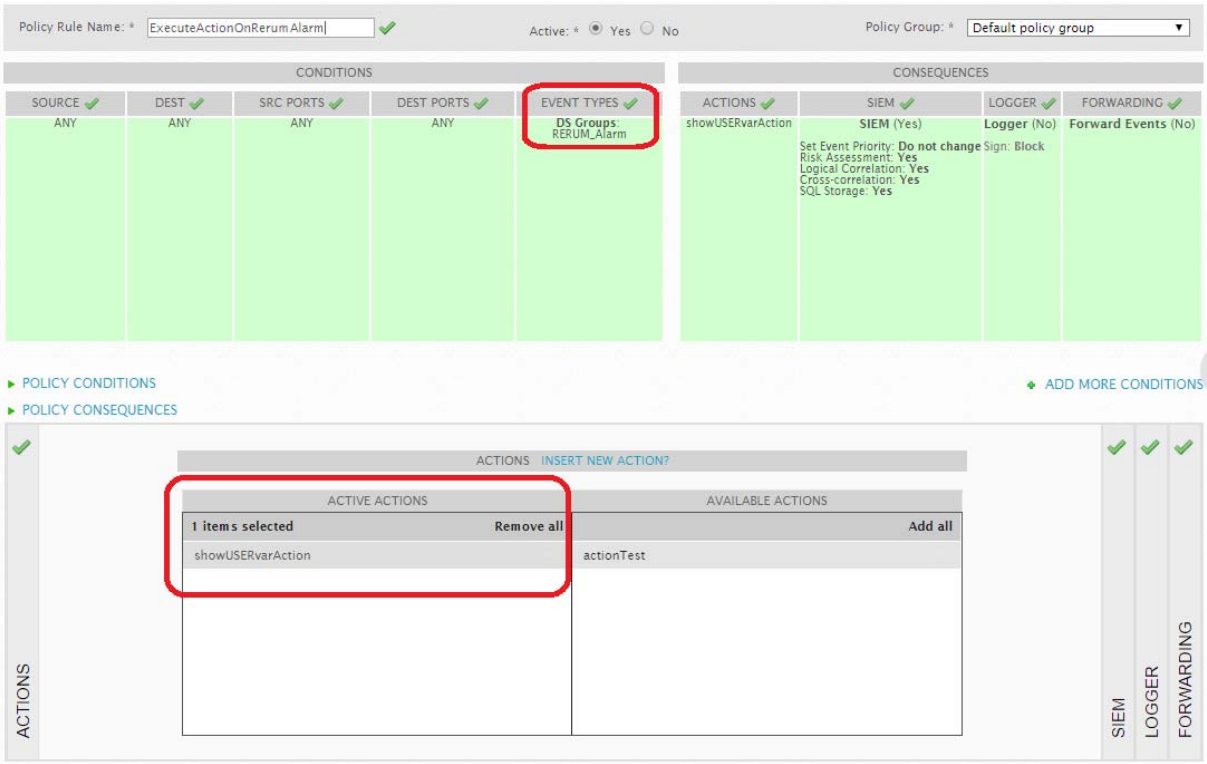


Figure 57: OSSIM Policy designer

5.4.3.3 Security Reactions for VRD Reconfiguration

The OSSIM Actions potential is limited by what we can do in a shell command line. The final objective is to react to the generated alerts by reconfiguring the RERUM Devices taking into account those alerts and the context of each device. For that reason we are delegating some of those reactions to a tool that take into account the system context to make a decision based in predefined rules, that is the Platform for Runtime Reconfigurability of Security, describes previously in Section 5.3.2.

6 Summary / Conclusion

This deliverable provided an overview of the security mechanisms that are developed within RERUM. This is provided at two logically different levels: device communication and service level. Both of these are supported by cryptographic primitives. The goal was to provide a detailed description of the components of the RERUM Security architecture. Basically, this document is the first of a series of three deliverables (of WP3) that jointly will provide a holistic Security, Privacy and Trust architecture for IoT deployments. This document set the foundations for this architecture, by providing the detailed components of the Security part.

Cryptography is the foundation of the security mechanisms within RERUM, thus in this document we first provide the notations and background on cryptographic primitives. This gives project partners, the IoT- and the academic-community the necessary background to understand the cryptographic primitives that are the basis for the security mechanisms described in this Deliverable. Note, group signatures or malleable signatures, for which the harmonized notation was already accepted as a scientific publication, are not yet used in any of the mechanisms described in this Deliverable. They are described in this deliverable already to ensure that all developments designed so far are flexible enough to be compatible also with cryptographic primitives that are updated or enhanced in future. The harmonised cryptographic notations stem from a very good and on-going collaboration among the security experts of RERUM and the discussion of their results with other partners.

One of the major IoT security issues concerns the security of the communication between the devices. Due to the fact that most IoT devices are resource constrained and cannot run advanced security mechanisms, up until now, most software developers didn't embed any type of security on the devices. Thus, secure communication is an important part of the RERUM architecture and within this document RERUM has described four secure communication mechanisms called "profiles". RERUM already prototypically implemented the profiles on RERUM devices to ensure the confidentiality and integrity of the communication, and allow origin authentication for messages exchanged securely within the RERUM world.

Briefly, the four profiles are:

- Profile DTLS describes how DTLS was brought on the Re-Mote, developing a research prototype for Contiki and the Re-Mote platform, by using the tinyDTLS library as our starting point (https://ict-rerum.eu/dtls-prototype/). With DTLS becoming available, RERUM is able to establish integrity and confidentiality (either end-to-end or hop-by-hop) at the transport layer level, including origin authentication. This transport layer security is achieved between RERUM Devices, but additionally between a RERUM Device and any other entity in, or even outside, the RERUM domain. With DTLS implementations being also available for non-constrained environments, RERUM greatly extends the coverage of confidentiality-protected communication.
- Profile On-Device-Signatures achieves seamless end-to-end integrity protection by implementing a cryptographically strong digital signature on a constrained device in a format that can be preserved as long as possible. This format, devised by RERUM, is called JSON Sensor Signature (JSS). We implemented elliptic curve cryptography (ECDSA) using the ecc-light library by NIST as a starting point. JSS uses the JSON Web Signature (JWS) as a starting point. But JSS keeps the JSON object's payload unchanged, so signed JSON stays compatible and the signed data can be processed as before by any processing step that correctly handles JSON. This includes entities like servers, outside of the constrained domain, and outside of RERUM's focus. Additionally, the format is open to be extended to different signature algorithms, allowing RERUM to later use it together with more privacy preserving cryptographic primitives that are researched in RERUM (RERUM Deliverable D3.2). JSS are also capable of using ECC based signatures based on Bernstein's Ed25519. From the research prototype (https://ict-rerum.eu/jss-ecdsa-prototype/) we can see that in only about two seconds even a constrained device like the Zolertia Z1 (MSP 430@16MHz)

can seamlessly lay the foundation for cryptographically strong end-to-end integrity protection of communications with RERUM Devices.

- Profile 802.15.4 Security presents the IEEE 802.15.4 security mechanisms that RERUM can use when securing low-level device communication. Again, this allows RERUM to secure the communication on constrained devices. These mechanisms achieve integrity and confidentiality in a hop-by-hop fashion between RERUM devices.
- Profile Lightweight and Secure Encryption Using Channel Measurements describes a secret key generation algorithm based on the compressive sensing (CS) theory is presented. It was accepted as an academic paper. This is used for lightweight symmetric encryption/decryption, as well as for compression/decompression, all in a single step. Secret key extraction is mainly based on channel measurements by recording the RSSI, fully available in commodity hardware. RSSI values are initially quantized and then hashed for achieving uniformity, a significant property for high CS performance. Secure sketches are then used for information reconciliation and final key generation. The evaluation results show that if an adversary is at a distance of more than a half of the wavelength of the carrier frequency, he experiences a reconstruction error of more than 60%, thus becoming unable to steal sensitive information. On the other hand, legitimate transceivers experience an error of less than 5%, depending on the quantization level.

Apart from the secure communication of the devices, an important part for protecting the devices is to ensure their secure configuration, which ensures the proper configuration of each RERUM Device connected to the RERUM network. Mechanisms for Secure Device Configuration have been provided in the deliverable, together with mechanisms to detect and react to malfunctions or misconfigurations on those devices.

In detail the procedures described within this document are the following:

- Fast and Secure Network Bootstrapping minimizes security attacks at the initial bootstrapping of RERUM Devices added to the network. Fresh RERUM Devices to be installed as part of a Smart City Use Case typically do not have security credentials available in the first step. The defined procedures balance the use of security mechanism across network layers as necessary to secure the distribution of different type of credentials needed by the RERUM framework. They start from establishing the lower level layer 2 links, getting a security base for further instantiation of the layers above and bootstrapping of operational credentials required within the RERUM network and Smart City application.
- Secure and Context Aware Dynamic Auto Configuration adapts the PRRS tool for take into account the context information in order to maintain the RERUM Devices software updated and configured.
- Self Management and Self Monitoring Mechanisms defines how the RERUM Devices are monitored to know its status and how this information can be used by an Alert Processor, in this case implemented with a SIEM, to trigger alerts in case of something strange happen in the RERUM Network. With this RERUM provides self-monitoring support.

At the Service level, RERUM has provided details on mechanisms for taking very controlled decisions based on the authorization of the requesting party.

Authorization in RERUM describes the access control mechanisms of the RERUM system. As authorization depends on proper authentication this is described for Service level authentication in RERUM. The goal was to support almost all available Identity Providers, both commercial and open source ones. In details, these authentication conditions state that all access to RERUM devices from the Internet must be done providing a security token issued by a trusted authentication provider, and user attributes must be included in that token as well. By an Analysis of authorization options the main options have been judged and the option elected is the policy based access proxy. RERUM suggests an authorization engine based on XACML policies. In this Deliverable RERUM has developed the authorization components defined in Deliverable D2.3. These components support authorization

decisions based on the attributes of the user, but also based on business specific logic contained in any text field of the request. The Design of Authorization Components has also been provided in this document.

The following table shows the relationship of the distinct security requirements of RERUM and the section of the document that fulfils them:

Table 4: Requirements addressed in this document

Requirement	Section
Secure bootstrapping of operational cryptographic credentials	5.2
Availability of initial credentials	5.2
Support of different operational credentials types	3.1 and 3.2 and 5.2
Reduction of manual interactions during credential bootstrapping	5.2
Update of operational credentials	5.2
Energy-efficient cryptographic primitives	2.3 and 3.1 and 3.2 and 3.3 and 3.4
Confidentiality protection of SL-C data in transit	3.1 and 3.3 and 3.4
Device authentication	3.1 and 3.2 and 3.3 and 3.4
Integrity protection of SL-I data in transit	3.1 and 3.2
Integrity protection of SL-I data at rest	2.5.2 and 3.2
Authorised modification of integrity protected data	4.4 and additionally this requirements is planned to be addressed by special MSS (planned to be described in Deliverable D3.2)
User authentication	4.2 (and technically also in 3.2)
Attribute-based access control	4.4
Low energy consumption	3.1 and 3.2 and 3.3 and 3.4
Microcontroller performance	3.1 and 3.2 and 3.3 and 3.4
Centralised management of constrained networks	5.3 and 5.4
Self-* mechanisms	5.3 and 5.4

Over-the-Air Programming	5.3
Find deployable software to RERUM devices	5.3
Monitoring and traceability by the middleware	5.3
Object configuration isolated per application	to be supported by application design
Secure design and implementation of RERUM components	all described mechanisms of this document

Referencess

[AB++11] J. H. Ahn, D. Boneh, J. Camenisch, S. Hohenberger, A. Shelat, and B. Waters. Computing on authenticated data. Cryptology ePrint Archive, Report 2011/096, 2011. http://eprint.iacr.org.

[ACMT05] G. Ateniese, D. H. Chou, B. de Medeiros, and G. Tsudik. Sanitizable signatures. In ESORICS, pages 159–177. Springer, 2005.

[AF12] Hazem, Ahmed, and Hossam AH Fahmy. "LCAP-a lightweight can authentication protocol for securing in-vehicle networks." 10th escar Embedded Security in Cars Conference, Berlin, Germany. 2012.

[AHP13] J. Aumassony, L. Henzenz, W. Plasencia. «Quark: a lightweight hash.» Journal of Cryptology, 2013: 313–339.

[AL11] imperialviolet.org Adam Langley. No! Don't enable revocation checking. Last accessed on 2014-06-09. 2011. url: https://www.imperialviolet. org/2014/04/19/revchecking.html.

[ALP12] N. Attrapadung, B. Libert, and T. Peters. Computing on authenticated data: New privacy definitions and constructions. In ASIACRYPT, pages 367–385, 2012.

[ALP13] N. Attrapadung, B. Libert, and T. Peters. Efficient completely context-hiding quotable and linearly homomorphic signatures. In PKC, pages 386–404, 2013.

[ASS14] Maryam Rajabzadeh Asaar and Mahmoud Salmasizadeh and Willy Susilo, Security Pitfalls of a Provable Secure Identity-based Multi-Proxy Signature Scheme by http://eprint.iacr.org/2014/496, 2014

[B06] D.J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Public Key Cryptography, pages 207-228, 2006.

[BB++10] C. Brzuska, H. Busch, O. Dagdelen, M. Fischlin, M. Franz, S. Katzenbeisser, M. Man- ulis, C. Onete, A. Peter, B. Poettering, and D. Schröder. Redactable Signatures for Tree-Structured Data: Definitions and Constructions. In Proceedings of the 8th International Conference on Applied Cryptography and Network Security, ACNS'10, pages 87–104. Springer, 2010. ISBN 3-642-13707-5, 978-3-642-13707-5.

[BB++10] C. Brzuska, H. Busch, O. Dagdelen, M. Fischlin, M. Franz, S. Katzenbeisser, M. Manulis, C. Onete, A. Peter, B. Poettering, and D. Schröder. Redactable Signatures for Tree-Structured Data: Definitions and Constructions. In ACNS, ACNS'10, pages 87–104. Springer, 2010.

[BC++14] D.J. Bernstein, T. Chou, C. Chuengsatiansup, A. Hulsing, T. Lange, R. Niederhagen, and C. van Vredendaal. How to manipulate curve standards: a white paper for the black hat. Technical report, Cryptology ePrint Archive, Report 2014/571.

[BCB13] W. Bechkit, Y. Challal and A. Bouabdallah, A new class of Hash-Chain based key pre-distribution schemes for WSN, Computer Communications, 2013, p. 243–255.

[BCLN14] J.W. Bos, C. Costello, P. Longa, and M. Naehrig. Selecting elliptic curves for cryptography: An efficiency and security analysis. IACR Cryptology ePrint Archive, 2014:130, 2014.

[BD++12] D.J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang. High-speed high-security signatures. Journal of Cryptographic Engineering, 2(2):77-89, 2012.

[BF++09] C. Brzuska, M. Fischlin, T. Freudenreich, A. Lehmann, M. Page, J. Schelbert, D. Schröder, and F. Volk. Security of Sanitizable Signatures Revisited. In PKC 2009, pages 317–336. Springer, 2009.

[BF++09b] C. Brzuska, M. Fischlin, A. Lehmann, and D. Schröder. Sanitizable signatures: How to partially delegate control for authenticated data. In Proc. of BIOSIG, volume 155 of LNI, pages 117–128. GI, 2009.

[BF11] D. Boneh and D. M. Freeman. Homomorphic signatures for polynomial functions. In EUROCRYPT, pages 149–168, 2011

[BF11b] D. Boneh and D. M. Freeman. Linearly homomorphic signatures over binary fields and new tools for lattice-based signatures. In Public Key Cryptography, pages 1–16, 2011.

[BFAS10] C. Brzuska, M. Fischlin, A. Lehmann, and D. Schröder. Unlinkability of Sanitizable Signatures. In PKC, pages 444–461, 2010.

[BFSH12] C. Bormann B. Frank Z. Shelby, K. Hartke. Constrained application protocol (CoAP), draft-ietf-core-coap-13. Orlando: The Internet Engineering Task Force - IETF, Dec, 2012.

[BGV01] An Efficient Identity-Based Key Management Scheme for Wireless Sensor Networks Using the Bloom Filter by Zhongyuan Qin, Xinshuai Zhang, Kerong Feng, Qunfang Zhang and Jie Huang, In Sensors 2014, 14(10), 17937-17951; doi:10.3390/s141017937

[BPS12] C. Brzuska, H. C. Pöhls, and K. Samelin. Non-Interactive Public Accountability for Sanitizable Signatures. In EuroPKI, volume 7868 of LNCS, pages 178–193. Springer, 2012.

[BPS14] C. Brzuska, H. C. Pöhls, and K. Samelin. Efficient and Perfectly Unlinkable San- itizable Signatures without Group Signatures. In EuroPKI 2013, volume 8341 of LNCS, pages 12–30. Springer, 2014.

[BS03] A. Boulis and M.B. Srivastava, “Node-level Energy Management for Sensor Networks in the Presence of Multiple Applications,” in Proc. IEEE PerCom Conf., Mar. 2003.

[BS12] D.J. Bernstein and P. Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, CHES, volume 7428 of Lecture Notes in Computer Science, pages 320-339. Springer, 2012.

[C08] D. Cooper. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. 2008.

[Certicom] Certicom.com, Online: http://www.certicom.com/index.php?action=res,cc&issue=2-2&&article=3 Explaining Implicit Certificates, Code and Cipher Vol. 2, no. 2

[CJ10] S. Canard and A. Jambert. On extended sanitizable signature schemes. In CT- RSA, pages 179–194, 2010.

[CJL12] S. Canard, A. Jambert, and R. Lescuyer. Sanitizable signatures with several signers and sanitizers. In AFRICACRYPT, pages 35–52, 2012.

[CLM08] S. Canard, F. Laguillaumie, and M. Milhau. Trapdoor sanitizable signatures and their application to content protection. In ACNS, pages 258–276, 2008.

[CLX09] E.-C. Chang, C. L. Lim, and J. Xu. Short Redactable Signatures Using Random Trees. In CT-RSA, CT-RSA ’09, pages 133–147, Berlin, Heidelberg, 2009. Springer.

[COAP14] “The Constrained Application Protocol (COAP) ”, RFC 7252, pp. 1-112, June 2014.

[CP05] H. Chan and A. Perrig, PIKE: peer intermediaries for key establishment in sensor networks, In proceedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 2005, p. 524–535.

[CPS03] H. Chan, A. Perrig and D. Song, Random key predistribution schemes for sensor networks, In Proceedings of 2003 IEEE Symposium on Security and Privacy, pp. 197-213, 2003.

[CPS10] A. Cavoukian, J. Polonetsky, and C. Wolf. Smartprivacy for the smart grid: embed- ding privacy into the design of electricity conservation. Identity in the Information Society, 3(2):275–294, 2010.

[CS97] CAMENISCH, Jan; STADLER, Markus. Efficient group signature schemes for large groups. In: Advances in Cryptology—CRYPTO'97. Springer Berlin Heidelberg, 1997. S. 410-424.

[CV91] Chaum, D., & Van Heyst, E. (1991, January). Group signatures. In Advances in Cryptology—EUROCRYPT'91 (pp. 257-265). Springer Berlin Heidelberg.

[CW08] E. Candes and M. Wakin, "An introduction to compressive sampling," IEEE Signal Processing Magazine, vol. 25, no. 2, pp. 21–30, 2008.

[D04] M.J. Dworkin. SP 800-38C. Recommendation for block cipher modes of operation: the CCM mode for authentication and confidentiality. 2004.

[DBN04] B. Deb, S. Bhatnagar, and B. Nath, "STREAM: Sensor Topology Retrieval at Multiple Resolutions," Kluwer Journal of Telecommunications Systems, vol. 26, no. 2, pp. 285–320, 2004.

[DH76] W. Diffie and M. E. Hellman. New directions in cryptography. IEEE Transactions on Information Theory, November 1976.

[DORS08] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy extractors: How to generate strong keys from biometrics and other noisy data," SIAM Journal on Computing, vol. 38, no. 1, pp. 97–139, 2008.

[DR08] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. 2008.

[DT13] R. Dautov, G. Tsouri. «Establishing secure measurement matrix for compressed sensing using wireless physical layer security.» ICNC. 2013. 354–358.

[DTLS15] Internet Engineering Task Force IETF. DTLS in constrained environments (DICE), 2015. https://datatracker.ietf.org/wg/dice/charter/.

[ELKS08] M. Elkstein (February 2008). "What is REST?".rest.elkstein.org

[ENIS13] European Union Agency for Network and Information Security. Algorithms, key sizes and parameters report 2013 recommendations, October 2013. Available at https://www.enisa.europa.eu.

[ET05] P. Eronen and H. Tschofenig. Pre-shared key cipher suites for transport layer security (TLS). Technical report, RFC 4279, December, 2005.

[F++A12] A. Fragkiadakis, E. Tragos, T. Tryfonas, I. Askoxylakis, "Design and performance evaluation of a lightweight wireless early warning intrusion detection prototype", EURASIP Journal on Wireless Communications and Networking, vol. 2012, pp. 1-18, 2012.

[F12] D. M. Freeman. Improved security for linearly homomorphic signatures: A generic framework. In PKC, pages 697–714, 2012.

[FFS88] Uriel Feige, Amos Fiat, and Adi Shamir. "Zero-knowledge proofs of identity." Journal of cryptology 1.2 (1988): 77-94.

[FIPS01] National Institute of Standards and Technology (NIST). Advanced Encryption Standard (AES). FIPS Publication 197,2001.

[FIPS11] PUB FIPS. 186-4. Digital signature standard (DSS). National Institute of Standards and Technology (NIST), 2011.

[FNT12] A. Fragkiadakis, S. Nikitaki, P. Tsakalides. «Physical-layer intrusion detection for wireless networks using compressed sensing.» WiMob. 2012. 845–852.

[FRAAT13] A. Fragkiadakis, I. Askoxylakis, E. Tragos. «Joint compressed sensing and matrix-completion for efficient data collection in wsns.» Camad. 2013. 84–88.

[FRL05] C. Fok, G. Roman, and C. Lu, "Mobile Agent Middleware for Sensor Networks: An Application Case Study", in Proc. IEEE ICDCS Conf., June 2005.

[FTT14] Fragkiadakis, A.; Tragos, E.; Traganitis, A., "Lightweight and secure encryption using channel measurements," Wireless Communications, Vehicular Technology, Information Theory and Aerospace & Electronic Systems (VITAE), 2014 4th International Conference on , vol., no., pp.1,5, 11-14 May 2014

[GMR88] S. Goldwasser, S. Micali, and R. L. Rivest. A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. SIAM Journal on Computing, 17:281–308, 1988.

[GQZ11] J. Gong, H. Qian, and Y. Zhou. Fully-secure and practical sanitizable signatures. In InsCrypt, volume 6584 of LNCS, pages 300–317. Springer, 2011.

[GYHP13] A. Gurtov M. Ylianttila E. Harjula P. Porambage, P. Kumar. Certificate based keying scheme for DTLS secure IoT, 2013.

[HH++08] S. Haber, Y. Hatano, Y. Honda, W. G. Horne, K. Miyazaki, T. Sander, S. Tezoku, and D. Yao. Efficient signature schemes supporting redaction, pseudonymization, and data deidentification. In ASIACCS, pages 353–362, 2008.

[HK12] S. Halder, and W. Kim, “A fusion approach of RSSI and LQI for indoor localization system using adaptive smoothers”, Journal of Computer Networks and Communications, vol. 2012, pp. 1-11, 2012.

[HL06] C. Hsin and M. Liu, “Self-monitoring of wireless sensor networks”, Computer Communications, Elsevier, 2006.

[HP14] Internet of Things Research Study, HP IoT report 2014. http://www8.hp.com/h20195/v2/GetDocument.aspx?docname=4AA5-4759ENW

[HS13] C. Hanser and D. Slamanig. Blank digital signatures. In AsiaCCS, pages 95 – 106. ACM, 2013.

[HS13] M. Hutter and P. Schwabe. NaCl on 8-bit AVR microcontrollers. In AFRICACRYPT, pages 156-172. Springer, 2013.

[IEEE06] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs), IEEE Standard, 802.15.4-2006”, 2006.

[IEEE802.15.4] “Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (WPANs), IEEE Standard, 802.15.4-2006”, 2006.

[IEEE802.15.4] IEEE802.15.4, Low-Rate Wireless Personal Area Networks (LR-WPANs), 2011

[IIKO11] T. Izu, M. Izumi, N. Kunihiro, and K. Ohta. Yet another sanitizable and deletable signatures. In AINA, pages 574–579, 2011.

[IKPS09] T. Izu, N. Kunihiro, K. Ohta, M. Sano, and M. Takenaka. Sanitizable and deletable signature. In ISA, volume 5379 of LNCS, pages 130–144. Springer, 2009.

[IOT13] Panagiotis Ilia, George Oikonomou, Theo Tryfonas, "Cryptographic Key Exchange in IPv6-Based Low Power, Lossy Networks", in Proc. Workshop in Information Theory and Practice (WISTP 2013), ser. Lecture Notes in Computer Science, 7886, pp. 34-49, 2013

[ISO10181-6] EF Michiels. ISO/IEC 10181-6: 1996 Information technology — Open Systems Interconnection — Security frameworks for open systems: Integrity framework. ISO Geneve, Switzerland, 1996.

[JAW74] Jakes, W. « Microwave Mobile Communications.» Wiley, 1974.

[JMSW02] R. Johnson, D. Molnar, D. Song, and D. Wagner. Homomorphic signature schemes. In Proceedings of the RSA Security Conference - Cryptographers Track, pages 244–262. Springer, Feb. 2002.

[JMSW02] R. Johnson, D. Molnar, D. Song, and D.Wagner. Homomorphic signature schemes. In CT-RSA, pages 244–262. Springer, Feb. 2002.

[Jø13] Audun Jøsang. PKI Trust Models. 2013.

[JP04] Audun Jøsang and Sťephane Lo Presti. “Analysing the Relationship be- tween Risk and Trust”. English. In: Trust Management. Ed. by Christian Jensen, Stefan Poslad, and Theo Dimitrakos. Vol. 2995. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 135–145. isbn: 978-3-540-21312-3. doi: 10.1007/978-3-540-24747-0_11. url: http: //dx.doi.org/10.1007/978-3-540-24747-0_11.

[JSON14] http://tools.ietf.org/html/rfc7159

[K87] N. Koblitz. Elliptic curve cryptosystems. Mathematics of computation, 48(177):203-209, 1987.

[KB08] A. Kundu and E. Bertino. Structural Signatures for Tree Data Structures. In Proc. of PVLDB 2008, New Zealand, 2008. ACM.

[KB10] A. Kundu and E. Bertino. How to authenticate graphs without leaking. In EDBT, pages 609–620, 2010.

[KB13] A. Kundu and E. Bertino. Privacy-preserving authentication of trees and graphs. Intl. J. of Inf. Sec., pages 1–28, 2013.

[KH03] T.H. Kim and S. Hong, “Sensor Network Management Protocol for State-Driven Execution Environment,” in Proc. ICUC Conf., Oct. 2003.

[KL06] M. Klonowski and A. Lauks. Extended Sanitizable Signatures. In ICISC, pages 343–355, 2006.

[KLS++10] J. Ko, C. Lu, M. B. Srivastava, J. A. Stankovic, A. Terzis, and M. Welsh, “Wireless sensor networks for healthcare,” Proceedings of the IEEE, vol. 98, no. 11, pp. 1947–1960, 2010.

[KRM13] Konrad-Felix Krentz, Hosnieh Rafiee, and Christoph Meinel. 2013. 6LoWPAN security: adding compromise resilience to the 802.15.4 security sublayer. In Proceedings of the International Workshop on Adaptive Security (ASPI '13). ACM, New York, NY, USA

[KS99] P. Karn and W.A. Simpson. ICMP security failures messages. 1999.

[LDO06] W. L. Lee, A. Datta, and R. Cardell-Oliver, "Network Management in Wireless Sensor Networks," in Handbook of Mobile Ad Hoc and Pervasive Communications: American Scientific Publishers, 2006.

[LKV02] B. Lai, S. Kim and I. Verbauwhede, Scalable Session Key Construction Protocols for Wireless Sensor Networks, In Proceedings of the IEEE Workshop on Large Scale RealTime and Embedded Systems, pp. 1-6, 2002.

[LLP12] S. Lim, E. Lee, and C.-M. Park. A short redactable signature scheme using pairing. Sec. and Comm. Netw., 5(5):523–534, 2012.

[LN03] D. Liu and P. Ning, Establishing Pairwise Keys in Distributed Sensor Networks, In proceedings of the 10th ACM conference on Computer and communications security, pp. 52-61, 2003.

[LN08] A. Liu and P. Ning. TinyECC: A configurable library for elliptic curve cryptography in wireless sensor networks. In Information Processing in Sensor Networks, 2008. IPSN'08. International Conference on, pages 245-256. IEEE, 2008.

[LXMT06] Z. Li, W. Xu, R. Miller, W. Trappe. «Securing wireless systems via lower layer enforcements.» WiSe. 2006. 33–42.

[M06] CITE: J.S. Milne, Elliptic Curves, 2006, BookSurge Publishers.

[M86] V. Miller. Use of elliptic curves in cryptography. In Advances in Cryptology - CRYPTO85 Proceedings, pages 417-426. Springer, 1986.

[MA12] Abdullah Al-Mahmud and Rumana Akhtar, Secure Sensor Node Authentication in Wireless Sensor Networks , http://research.ijcaonline. org/volume46/number4/pxc3879238.pdf, 2012

[MC96] D. Harrison McKnight and Norman L. Chervany. The Meanings of Trust. Working Paper. 1996. url: http://www.misrc.umn.edu/workingpapers/ fullPapers/1996/9604_040100.pdf.

[MF12] Manulis, M., Fleischhacker N. (2012). Group Signatures: Authentication with Privacy. Technical University Darmstadt. https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/GruPA/GruPA.pdf?__blob=publicationFile

[MHI06] K. Miyazaki, G. Hanaoka, and H. Imai. Digitally signed document sanitizing scheme based on bilinear maps. ASIACCS '06, pages 343–354, New York, NY, USA, 2006. ACM.

[MI++05] K. Miyazaki, M. Iwamura, T. Matsumoto, R. Sasaki, H. Yoshiura, S. Tezuka, and H. Imai. Digitally Signed Document Sanitizing Scheme with Disclosure Condition Control. IEICE Transactions, 88-A(1):239–246, 2005.

[MOV01] A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. Handbook of Applied Cryptography. CRC Press, 5th printing, 2001 edition, 1996. Available at www.cacr.math.uwaterloo.ca/hac.

[MPPS13] H. de Meer, H. C. Pöhls, J. Posegga, and K. Samelin. Scope of security properties of sanitizable signatures revisited. In ARES, pages 188–197, 2013.

[MPPS14] H. de Meer, H. C. Pöhls, J. Posegga and K. Samelin. On the Relation between Redactable and Sanitizable Signature Schemes. In Proc. of the 6th International Symposium on Engineering Secure Software and Systems (ESSoS 2014), pages 113-130, Springer International Publishing, 2014.

[MR04] N. Modadugu and E. Rescorla. The design and implementation of datagram TLS. In NDSS, 2004.

[MS++03] K. Miyazaki, S. Susaki, M. Iwamura, T. Matsumoto, R. Sasaki, and H. Yoshiura. Digital documents sanitizing problem. Technical report, IEICE, 2003.

[MVTM11] S. Mathur, A. Varshavsky, W. Trappe, and N. Mandayam. «Proximate: Proximity-based secure pairing using ambient wireless signals.» Mobisys. 2011. 211-224.

[NS+T13] S. Nikitaki, P. Scholl, K. Laerhoven, P. Tsakalides. «Localization in wireless networks via laser scanning and bayesian compressed sensing.» SPAWC. 2013. 739–743.

[NUMS14] Internet Engineering Task Force IETF. Nothing up my sleeve (NUMS) curves for ephemeral key exchange in transport layer security (TLS) (draft), 2014. https://tools.ietf.org/html/draft-black-tls-numscurves-00.

[OAS08] A. Orsdemir, H. Altun, M. Sharma. «On the security and robustness of encryption via compressed sensing.» MILCOM. 2008. 1-7.

[OASIS] https://www.oasis-open.org/

[OpId12] http://openid.net/ and https://tools.ietf.org/html/rfc6749

[PJC++K13] S. Premnath, S. Jana, J. Croft, P. Gowda, M. Clark, S. Kasera. «Secret key extraction from wireless signal strength in real environments.» IEEE Transactions on Mobile Computing, 2013: 917-930.

[PKCS#12] RSA Laboratories, PKCS #12: Personal Information Exchange Syntax, PKCS Version 1.1, December 2012.

[PN++S12] A. Pascale, M. Nicoli, F. Deflorio, B. Dalla Chiara, and U. Spagnolini, “Wireless sensor networks for traffic management and road safety,” IET Intelligent Transport Systems, vol. 6, no. 1, pp. 67–77, 2012.

[PP++13] H. C. Pöhls, S. Peters, K. Samelin, J. Posegga, and H. de Meer. Malleable signatures for resource constrained platforms. In WISTP, pages 18–33, 2013.

[PS++02] A. Perrig, R. Szewczyk, V. Wen, D. Culler and J. D. Tygar, SPINS: Security Protocols for Sensor Networks, Wireless Networks, 2002, p. 521-534.

[PSP11] H. C. Pöhls, K. Samelin, and J. Posegga. Sanitizable Signatures in XML Signature - Performance, Mixing Properties, and Revisiting the Property of Transparency. In ACNS, volume 6715 of LNCS, pages 166–182. Springer, 2011.

[PV00] Leon Pintsov and Scott Vanstone, Postal Revenue Collection in the Digital Age, Financial Cryptography 2000, Lecture Notes in Computer Science 1962, pp. 105–120, Springer, February 2000

[PV11] Roberto Di Pietro , Alexandre Viejo, Location privacy and resilience in wireless sensor networks querying, Computer Communications, v.34 n.3, p.515-523, March, 2011

[QZ++14] An Efficient Identity-Based Key Management Scheme for Wireless Sensor Networks Using the Bloom Filter by Zhongyuan Qin, Xinshuai Zhang, Kerong Feng, Qunfang Zhang and Jie Huang, In Sensors 2014, 14(10), 17937-17951; doi:10.3390/s141017937

[RAYA05] N. Ramanathan and M. Yarvis, “A Stream-oriented Power Management Protocol for Low Duty Cycle Sensor Network Applications,” in Proc. IEEE EmNetS-II Workshop, May 2005

[RB08] Y. Rachlin, D. Baron. «The secrecy of compressed sensing measurements.» Allerton. 2008. 813–817

[RFC4944] RFC4944, Transmission of IPv6 Packets over IEEE 802.15.4 Networks, 2007

[RFC6550] RFC6550, RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, 2012

[RKE05] N. Ramanathan, E. Kohler, and D. Estrin, “Towards a Debugging System for Sensor Networks,” International Journal for Network Management, vol. 15, no. 4, pp. 223–234, 2005.

[RLZ08] K. Ren, W. Lou and Y. Zhang, LEDS: Providing Location-Aware End-to-End Data Security in Wireless Sensor Networks, IEEE Transactions On Mobile Computing, 2008, p. 585-598.

[RSA78] R. L. Rivest, A. Shamir, and L. M. Adleman. A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM, 21(2):120-126, 1978.

[RSH10] M. Rahman, S. Sampalli and S. Hussain, A Robust Pair-wise and Group Key Management Protocol for Wireless Sensor Network, In 2010 IEEE GLOBECOM Workshops, 2010, pp. 1528-1532.

[SB02] R. Steinfeld and L. Bull. Content extraction signatures. In ICISC. Springer Berlin / Heidelberg, 2002.

[SBZ02] Ron Steinfeld, Laurence Bull, and Y. Zheng. Content extraction signatures. In 4th International Conference on Information Security and Cryptology (ICISC 2001), volume 2288, pages 163–205. Springer, 2002.

[SENSO] “Sensorscope: Sensor networks for environmental monitoring, http://lcav.epfl.ch/sensorscope-en.”

[SKLS05] H. Song, D. Kim, K. Lee, and J. Sung, “Upnp-Based Sensor Network Management Architecture,” in Proc. ICMU Conf., Apr. 2005.

[SM95] Gustavus J. Simmons and Catherine Meadows. “The Role of Trust in In- formation Integrity Protocols”. In: Journal of Computer Security (1995), pp. 71–84.

[SO++08] P. Szczechowiak, L.B. Oliveira, M. Scott, M. Collier, and R. Dahab. NanoECC: Testing the limits of elliptic curve cryptography in sensor networks. In Wireless sensor networks, pages 305-320. Springer, 2008.

[SOAP07] http://www.w3.org/TR/soap/

[SP++12] K. Samelin, H. C. Pöhls, A. Bilzhause, J. Posegga, and H. de Meer. On Structural Signatures for Tree Data Structures. In ACNS, volume 7341 of LNCS, pages 171– 187. Springer, 2012.

[SP++12b] K. Samelin, H. C. Pöhls, A. Bilzhause, J. Posegga, and H. de Meer. Redactable signatures for independent removal of structure and content. In ISPEC, volume 7232 of LNCS, pages 17–33. Springer, 2012.

[SR10] Daniel Slamanig and Stefan Rass. Generalizations and extensions of redactable signatures with applications to electronic healthcare. In Communications and Multi- media Security, pages 201–213. Springer, 2010.

[SR10] D. Slamanig and S. Rass. Generalizations and extensions of redactable signatures with applications to electronic healthcare. In Communications and Multimedia Security, pages 201–213, 2010.

[TC05] G. Tolle and D. Culler, “Design of an Application- Cooperative Management System for Wireless Sensor Networks,” in Proc. EWSN, Feb. 2005.

[TG07] J. Tropp and A. Gilbert, “Signal recovery from random measurements via orthogonal matching pursuit” IEEE Transactions on Information Theory, vol. 53, pp. 4655–4666, 2007.

[TI13] Texas Instruments, “CC2538 System-on-Chip Solution for 2.4-GHz IEEE 802.15.4 and ZigBee®/ZigBee IP® Applications”, Version C, May 2013

[TI14] Texas Instruments, “CC2538 Powerful System-On-Chip for 2.4-GHz IEEE 802.15.4, 6LoWPAN and ZigBee® Applications”, Sep. 2014

[TURON05] M. Turon, “MOTE-VIEW: A Sensor Network Monitoring and Management Tool,” in Proc. IEEE EmNetS-II Workshop, May 2005.

[VMM12] Binod Vaidya, Dimitrios Makrakis, and Hussein Mouftah. "Secure remote access to Smart Energy Home area Networks." Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES. IEEE, 2012.

[WH++12] Zhen-Yu Wu, Chih-Wen Hsueh, Cheng-Yu Tsai, Feipei Lai, Hung-Chang Lee, and Yufang Chung. Redactable Signatures for Signed CDA Documents. Journal of Med- ical Systems, 36:1795–1808, 2012. ISSN 0148-5598. doi: 10.1007/s10916-010-9639-0.

[Woe06] Thomas Woelfl. “Formale Modellierung von Authentifizierungs- und Autorisierungsinfrastrukturen: Authentizitaet von deskriptiven Attributen und Privilegien auf der Basis digitaler Zertifikate.” http://d-nb.info/980317398. PhD thesis. University of Regensburg, 2006, pp. 1–139. isbn: 978-3-8350- 0498-6.

[WP++13] F. Wu, H. T. Pai, X. Zhu, P. Y. Hsueh and Y. H. Hu, An adaptable and scalable group access control scheme for managing wireless sensor networks, Telematics and Informatics, 2013, p. 144–157.

[WS++13] M. Winkler, M. Street, K. D. Tuchs, and K. Wrona, “Wireless sensor networks for military purposes,” in Autonomous Sensor Networks, vol. 13 of Springer Series on Chemical Sensors and Biosensors, pp. 365–394, 2013.

[WT++14] P. Wouters, H. Tschofenig, J. Gilmore, S. Weiler, and T. Kivinen. Using raw public keys in transport layer security (TLS) and datagram transport layer security (DTLS). Internet Requests for Comments, RFC Editor, RFC, 7250, 2014.

[XACML13] https://www.oasis-open.org/committees/xacml/

[XPHZ13] Y. Xiaoqing, W. Pute, W. Hana, and Z. Zhanga, "A survey on wireless sensor network infrastructure for agriculture," Computer Standards and Interfaces, vol. 35, no. 1, pp. 59–64, 2013.

[YPL10] D. H. Yum, J. W. Seo, and P. J. Lee. Trapdoor sanitizable signatures made easy. In ACNS, ACNS'10, pages 53–68, Berlin, Heidelberg, 2010. Springer.

[YRS06] C. Ye, A. Reznik, Y. Shah. «Extracting secrecy from jointly gaussian random variables.» ISIT. 2006. 2593–2597.

[YRW10] An Authentication Framework for Wireless Sensor Networks using Identity-Based Signatures by Rehana Yasmin, Eike Ritter, Guilin Wang, Rehana Yasmin, Eike Ritter, Guilin Wang http://www.uow.edu.au/~guilin/papers/TSP10-WSNs.pdf,

[YSLM08] T. H. Yuen, W. Susilo, J. K. Liu, and Y. Mu. Sanitizable signatures revisited. In CANS, pages 80–97, 2008

[ZC14] Z. Shelby, C. Chauvenet, "The IPSO Application Framework", Aug. 2014.

[ZOL1] "Zolertia Z1 Platform", http://www.zolertia.com/products/z1.

