Deliverable D4.2: open-source toolkit for security in CASCADAS.

Status and Version:	Version 1.0, Final	
Date of issue:	02.07.2007	
Distribution:	Public	
Author(s):	Name	Partner
	Pietro Michiardi	Institut Eurecom
	Roberto Cascella	UNITN
	Ricardo Lent	ICL
	Christos Xenakis	NKUA
	Sanjay Rawat	UNITN
	Mauro Brunato	UNITN
	Ioannis Stavrakakis	NKUA
	Roberto Battiti	UNITN
Checked by:	Bruno Crispo	UNITN

Table of Contents

1 Introduction

Traditionally, security has often been neglected during the early stages of the design of a system, mechanism, protocol or algorithm. It is in general well known that the hard task of coming up with a working prototype of a new (maybe revolutionary) system is a challenge per-se, while its secure operation is often left aside relegating the choice of pertinent attacker models and countermeasures as add-on mechanisms that can be plugged in after several trials of the un-secure system.

In CASCADAS, the partners involved in WP4, in a joint effort with the whole project consortium, came up with a first deliverable (D4.1) [28] in which a wide range of security problems related to the very nature of autonomic systems has been dissected. Not only numerous attacker models have been considered, ranging from *malicious* entities, targeting at disrupting the correct functioning the CASCADAS system as a whole or aiming at thwarting in particular the most sensible parts of it, to *selfish* entities, whose target is to maximize their utility in participating to the system or minimize their costs. But also numerous research directions have been investigated, tackling problems that are specific to the very nature of an autonomic system. These initially widespread research directions have been narrowed down after the first year of the project, yet they remain intellectually important and technically challenging problems that need to be carefully addressed during the whole project: they constitute the added-value, from a research stand point, of the WP4 activities in the security domain for the CASCADAS project.

However, in order to protect the system from traditional attacks that mine the system by exploiting communication vulnerabilities or by improper use of messages and resources, in the following of this deliverable (which follows up some basic considerations that were made already in deliverable D4.1) we focus on basic security problems that are common to any communicating systems and that range from **information security** and **communication security** services. Furthermore, we present a relevant case study that focus on IPSec, a security framework for end-to-end communications that is suitable for an evolved version of the CASCADAS architecture.

In the following sections, we introduce, explain and provide practical examples of those basic security services that are needed by the founding components of the CASCADAS architecture (namely the ACEs), to securely communicate, to securely store data, and to securely grant access to resources.

1.1 Purpose and Scope

This document is intended as a practical guide for all the CASCADAS partners to help in understanding what are the most common security services that are needed for a (autonomic) communicating system. The approach that we take in this Deliverable is to use current technology. The reader of this Deliverable should not look for innovative methods to achieve well-known security goals that affect today and future communication systems; rather, we provide an engineering guide whose aim is to come as a solid ground for a joint implementation effort, towards the integration of the many components that constitute the CASCADAS architecture.

In our effort to come up with hands-on examples of typical security problems of a communicating system, we selected those cryptographic libraries that are compatible with the development environment chosen for the project. For these libraries, we explain how the basic constructs and interfaces can be used by other partners to integrate and complement their software components to achieve the basic security levels required by the application scenarios defined as prominent examples of the CASCADAS system. Nonetheless, the integration effort needs to be carried out by all parties involved in the implementation of the CASCADAS demonstrator.

Lastly, we also focus on practical considerations that need to be pondered when deciding which security service or cryptographic function needs to be selected to achieve a specific goal, both taking as a reference the Goal Achievable / Goal Needed philosophy, and more practical considerations that are concerned with application performance, storage and computational requirements and so on. Also note the case study section wherein we focus on practical considerations and performance analysis of deploying an IPSec framework.

1.2 Reference Material

1.2.1 Reference Documents

[1] O. Elkeelany et. all, "Performance Analysis of IPSec Protocol: Encryption and Authentication," IEEE Communications Conference (ICC 2002), pp. 1164-1168, 2002.
[2] S. Kent, R. Atkinson, "Security Architecture for the Internet Protocol," RFC 2401, Nov. 1998.
[3] E. Danielyan, "Goodbye DES, Welcome AES," Cisco The Internet Protocol Journal, vol. 4, no. 2, June. 2001, pp 15-21.
[4] S. Frankel, R. Glenn, S. Kelly "The AES-CBC Cipher Algorithm and Its Use with IPsec," RFC 3602, Sept. 2003.
[5] R. Phan, "Impossible differential cryptanalysis of 7-round Advanced Encryption Standard (AES)," Information Processing Letters, Vol 91 Issue 1, July 2004, pp 33-38.
[6] D. Bertsekas, R. Gallager, "Data Networks", Prentice Hall, 1992.
[7] US National Bureau of Standards, "Data Encryption Standard," Federal Information Processing Standard (FIPS) publication 46-2, Dec. 1993, http://www.itl.nist.gov/fipspubs/fip46-2.htm
[8] R. Rivest, "The MD5 Message-Digest Algorithm," RFC1321, Apr 1992.
[9] D. Eastlake, P. Jones, "US Secure Hash Algorithm 1 (SHA1)," RFC 3174, Sept. 2001.
[10] C. Madson, R. Glenn, "The Use of HMAC-MD5-96 within ESP and AH," RFC 2403, Nov. 1998
[11] C. Madson, R. Glenn, "The Use of HMAC-SHA-1-96 within ESP and AH," RFC 2404, Nov. 1998
[12] National Institute of Standards and Technology (NIST), "Advanced Encryption Standard (AES)," Federal Information Processing Standard (FIPS) publication 197, Nov. 2001, http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf
[13] F. Granelli, G. Boato, "A novel methodology for analysis of the computational complexity of block ciphers: Rijndael, Camellia and Shacal-2 compared," 3rd Conference on Security and Network Architectures (SAR'04), June 2004, http://eprints.biblio.unitn.it/archive/00000514/01/DIT-04-004.pdf
[14] J. Daemen, V. Rijmen, "The Design of Rijndael", Springer, 2002.
[15] C. Lu, S. Tseng, " Integrated Design of AES (Advanced Encryption Standard) Encrypter and Decrypter, " Proc. of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors (ASAP'02), 2002.
[16] ETSI,Universal Mobile Telecommunication System (UMTS);Selection Procedures

for the Choice of Radio Transmission Technologies of the UMTS, Technical Report TR 101 112 v3.2.0,1998.

[17] ARM microprocessor solutions from ARM Ltd, http://www.arm.com/products/CPUs

[18] National Institute of Standard, (NIST), Cryptographic Toolkit, Random Number Generation, http://csrc.nist.gov/CryptoToolkit/tkrng.html

[19] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for Key Management - PArt 1: General. NIST Special Publication 800-57, May 2006. see http://csrc.nist.gov/publications/nistpubs/800-57/SP800-57-Part1.pdf.

[20] Dan Boneh. Java cryptography extension 1.2 -api specification & reference. On-line Tutorial. see http://crypto.stanford.edu/˜dabo/courses/cs255_winter00/JCE-1.2.htm.

[21] W. Diffie and M. E. Hellman. New directions in cryptography. IEEE Trans. Inform. Theory, IT-22: 644– 654, November 1976.

[22] Ueli M. Maurer and Stefan Wolf. The Diffie-Hellman protocol. Designs, Codes and Cryptography, 19:147–171, 2000.

[23] DocJar Services. Crypto java docs. On-line Tutorial. see http://www.docjar.com/docs/api/javax/crypto/overview-summary.html.

[24] The Java Tutorials. Lesson: Api and tools use for secure code and file exchanges. On-line Tutorial. see http://java.sun.com/docs/books/tutorial/security/sigcert/index.html.

[25] The Bouncy Castle Crypto java http://www.bouncycastle.org/java.html

[26] Java Cryptography Architecture. API Specification & Reference. http://java.sun.com/j2se/1.5.0/docs/guide/security/CryptoSpec.html

[27] CASCADAS Consortium. D1.1. Report on state-of-art, requirements and ACE model. January 2007.

[28] CASCADAS Consortium. D4.1. Security Architecture. January 2007

[29] Pretty good privacy. World Wide Web: http://www.pgpi.org.

[30] Alfarez Abdul-Rahman and Stephen Hailes. A distributed trust model. In NSPW '97: Proceedings of the 1997 workshop on New security paradigms, pages 48{60, New York, NY, USA, 1997. ACM Press.

[31] C. Adams and S. Farrell. Internet x.509 public key infrastructure: Certificate management protocols. RFC 2510. Technical report, 1999.

[32] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. IEEE Symposium on Security and Privacy, pages 164{173, 1996. 1081-6011 1996; Annual: 1 issue per year IEEE COMPUTER SOCIETY USA

[33] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The role of trust management in distributed systems security. pages 185{210, 1999.

[34] Carl Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian Thomas, and Tatu Ylonen. SPKI certificate theory. Technical report, IETF, Sept. 1999. RFC 2693

[35] W. Josephson, E. Sirer, and F. Schneider. Peer-to-peer authentication with a distributed single sign-on service. In International Workshop on Peer-to-Peer Systems, 2004

[36] B. C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. IEEE Communications Magazine, 32(9):33, 1994. 0163-6804

[37] T. A. Parker. Single sign-on systems-the technologies and the products. In Security and Detection, European Convention, pages 151{155, 1995.

[38] R. S. Sandhu and P. Samarati. Access control: Principles and practice. IEEE Communications Magazine, 32(9):40, 1994. 0163-6804

[39] Giorgos Zacharia and Pattie Maes. Trust management through reputation mechanisms. Applied Artificial Intelligence, (14):881{907, 2000.

[40] Christos Xenakis, Nikos Laoutaris, Lazaros Merakos, Ioannis Stavrakakis, "A Generic Characterization of the Overheads Imposed by IPsec and Associated Cryptographic Algorithms," Computer Networks, Elsevier Science, Vol. 50, No. 17, Dec 2006, pp. 3225-3241.

1.2.2 Acronyms

TMF	Telemanagement Forum
SKC	Symmetric Key Cryptography
PKC	Public Key Cryptography
PKI	Public Key Infrastructure
DH	Diffie-Hellman
DC	Digital Certificate

1.2.3 Definitions

TMF Telemanagement Forum , formerly NMF, Network Management Forum

1.3 Document History

Version	Date	Authors	Comment
0.1	11/06/2007	Pietro Michiardi	Draft Document
0.2	14/06/2007	Sanjay Rawat	Key Management Section
0.3	19/06/2007	Roberto Cascella	Draft Section 7 and 8
0.4	21/06/2007	Christos Xenakis	Update on section 6
0.5	20/06/2007	Pietro Michiardi	Update on section 2
0.6	21/06/2007	Roberto Cascella	Update on references
0.7	26/06/2007	Pietro Michiardi	Updated conclusion
0.8	02/07/2007	Roberto Cascella	Document checking
1.0	06/07/2007	Pietro Michiardi	Final editing

1.4 Document overview

Keeping in mind our original goal, this Deliverable is concise and technical, and its structure can be summarized as follows:

Section 2 describes the basic security services we provide for the CASCADAS architecture, its components, and its communication protocols.

Section 3 is dedicated to the description of the cryptographic functions that are required by the security services outlined in Section 2.

Section 4 reviews the fundamental problem of key management and distribution, the indispensable ingredient that is required for security services to function properly.

With the aim of reviewing the most common techniques in protecting resources available in the (distributed) components that constitute the CASCADAS framework, in Section 5 we delve into the problem of access control and present several practical techniques to achieve resource preservation and control their usage by remote and possibly un-secure parties.

Section 6 discusses on practical considerations that need to be made when deciding which kind of security service to implement in CASCADAS components and the impact of this choice on system performance and requirements. In this section we provide also a case study that is related to the use of IPSec as a secure, end-to-end, communication framework that is well suited for future development of the project, in which a business oriented approach can be of value.

Section 7 gives an overview of the ACE model and protocol of communication and describes how a security library is included in an ACE. Section 8 discusses how security is applied to the case scenario envisioned for CASCADAS.

Finally, Section 9 describes cryptographic functions and libraries that are open source in nature and that can be seamlessly integrated in the CASCADAS framework.

2 Basic security services

Traditionally, security has been reckoned an important issue for infrastructure networks (i.e. networks in which dedicated components, such as routers, provide the basic network operation), especially for those running security-sensitive applications.

Similarly, the security of an infrastructure-less network (such as the one addressed by the CASCADAS project) and the applications designed for such type of dedicated networks, is of paramount importance.

The basic security services that we target in the framework of CASCADAS focus on the protection of code, data and resources of a system. These services need to be determined using global attributes (such as privacy, confidentiality, anonymity, integrity, accountability and availability) in order to specify the appropriate level of security, in view of the different type of CASCADAS components. It should be noted that this Section is a follow up of what has been presented in the Deliverable D4.1 [28].

An autonomic communication system such as the one we target in the CASCADAS project may need to keep any data stored on a node, carried by an agent, or exchanged between system components. For this reason, system components must be able to ensure that their communications remain confidential if required.

Except for confidentiality, an autonomic system should be able to provide nodes with anonymity. It should keep a node's identity secret from other nodes, but it maintains a form of reversible anonymity where it can determine the node's identity, if necessary and legal. However, there are many situations in which the participants are unwilling to engage in transactions with anonymous counterparts. Purchasers of goods and services may want to protect their privacy by remaining anonymous, but credit agencies would not extend credit to anonymous consumers without being able to verify their credit history and credit worthiness.

A node should provide data integrity to protect it against unauthorized modification or tampering. In addition, the secure operation of autonomic systems depends on the integrity of local and remote nodes.

Not only agents are targeted by attacks that originate from a node, but also the opposite. For this reason, system access controls must be in place to protect the integrity of the node from unauthorized users and from network worms, trojan horses and computer viruses. The agents should not be allowed to violate the node's resources (e.g., files, network resources, etc) and they should have only restricted access to them.

In the following, we present a common understanding of the basic security services and provide a brief overview of them. It is also important to relate the following concepts with what will be presented in Section 6, wherein practical considerations (especially that are concerned with computational requirements) are discussed with respect to the basic security services addressed by this Document.

2.1 Integrity

Integrity guarantees that a message being transferred is never corrupted. A message could be corrupted because of benign failures or because of malicious attacks on the network.

Practically, integrity is achieved by appending to a message an un-forgeable digest of the original message that has the following property: the modification of a single bit in the original message would invalidate the message digest, and inform of a recipient of an integrity attack. Message digest takes the form of a hash function, which has the property of "translating" a message from its original form (the *domain* of the hash function) to a concise and unique summary of the message (the *co-domain* of the hash function). In Section 7 we describe how to use in practice hash functions using the java cryptographic library we suggest for the CASCADAS project.

2.2 Authentication

Authentication enables a node to ensure the identity of the peer node it is communicating with. Without authentication, an adversary could masquerade a node, thus gaining unauthorized access to resource and sensitive information and interfering with the correct operation of other nodes.

As a typical example a message can be considered authentic when it is digitally signed, using for example a RSA signature. In the following sections we provide practical examples on how to generate or obtain a digital certificate proving the identity of a peer or node of the system and how to use the cryptographic keying material to digitally sign a message.

Authentication can be also defined for data exchanged between parties that share a common secret. For instance, to ensure that data are coming from an entity that knows the shared secret, an authenticated digest of the message is created using for example a

HMAC. This specific case is presented in more details in Section 7 where we show how data authentication can be applied in the context of the ACE component.

2.3 Confidentiality

Confidentiality ensures that certain information is never disclosed to unauthorized entities. Network transmission of sensitive information, such as strategic or economically valuable information, requires confidentiality. Leakage of such information to an eavesdropper could have severe consequences.

In the following sections we discuss in detail how confidentiality can be achieved through message encryption: several encryption techniques are discussed (both symmetric and asymmetric, depending on the keying material used and on the performance required).

2.4 Non repudiation

Non-repudiation ensures that the origin of a message cannot deny having sent the message. There are other security goals (e.g., authorization, intrusion detection, etc...) that are of concern to certain applications that we will discuss later in this section.

Similarly to a digital signature, non-repudiation of the origin can be achieved using the cryptographic library proposed for the project. Non-repudiation of receipt can be a more difficult problem that requires the design of a dedicated protocol: depending on the application requirements, the non-repudiation protocol should be centralized or not.

3 Basic cryptographic functions

3.1 Ciphering algorithms

The Data Encryption Standard (DES) algorithm, [7], is a symmetric (shared secret key) block cipher with block and key size of 64 bits (8 of the 64 bits of the key are used for odd parity, reducing the effective key length). Although widely used, DES has been compromised on several occasions in the past; in fact there exists specialized hardware for breaking it in a few hours [9]. This has lead to the introduction of triple DES (3DES), which is no more than a triple repetition of the basic DES encryption: first the data block is DES-encrypted using an initial key, then the encrypted block is decrypted using a second (different) key and then the new block is re-encrypted using the initial key. This process is equivalent to using a larger effective key length of 112 bits. The obvious disadvantage of 3DES is that it runs three times slower than DES on a given platform.

The Rijndael algorithm, selected as the algorithm of choice for the new Advanced Encryption Standard (AES), [3,4,12], is one of the newest components of IPsec. Rijndael is a symmetric block cipher that supports different key and block sizes (128, 192, or 256 bits). The AES standardized version of Rijndael, however, is tied to a fixed block size of 128 bits. The initial block is passed through a round transformation function, which is repeated 10 times (respectively, 12 or 14) under a key length of 128 bits (respectively, 192 or 256). Rijndael combines an increased resistance against attacks with an implementation simplicity and, thus, high execution rate. It has proved to be quite durable against differential, truncated differential, linear, interpolation, and Square attacks, [3,5]. Rijndael is quite versatile as it may also serve as a Message Authentication Code (MAC) algorithm, as a hash function and as a pseudo random number generator.

3.2 Hash functions

The Message Digest (MD5) [8] and Secure Hash Algorithm 1 (SHA-1), [9], implement so called "one-way hash functions" and are usually used in conjunction with the above cryptographic algorithms for performing authentication. Both of them process input text blocks of 512 bits to generate 128-bit and 160-bit hash values, respectively, which verify the correct message transfer. Both apply padding to make the plaintext a multiple of 512 bits, but they cannot be directly used as MAC algorithms, as they do not include a secret key. For that reason, they are used in conjunction with keyed-Hashing for Message Authentication (HMAC), [10, 11]. HMAC is a secret key authentication algorithm that provides a framework for incorporating various hashing functions. The combined HMAC-MD5 and HMAC-SHA-1 mechanisms are in position to offer data origin authentication and integrity protection services.

3.3 Random Number Generators

A cryptographically secure pseudo-random number generator (CSPRNG) is a pseudo-random number generator (PRNG) with properties that make it suitable for use in cryptography.

Many aspects of cryptography require random numbers, for example:

- Key generation
- Nonces
- Salts in certain signature schemes.
- One-time pads

The "quality" of the randomness required for these applications varies. For example creating a nonce in some protocols needs only uniqueness. On the other hand, generation of a master key requires a higher quality, such as more entropy. And in the case of one-time pads, the information-theoretic guarantee of perfect secrecy only holds if the key material is obtained from a true random source with high entropy.

Ideally, the generation of random numbers in CSPRNGs uses entropy obtained from a high quality source, which might be a hardware random number generator or perhaps unpredictable system processes — though unexpected correlations have been found in several such ostensibly independent processes. From an information theoretic point of view, the amount of randomness, the entropy that can be generated is equal to the entropy provided by the system. But sometimes, in practical situations, more random numbers are needed than there is entropy available. Also the processes to extract randomness from a running system are slow in actual practice. In such instances, a CSPRNG can sometimes be used. A CSPRNG can "stretch" the available entropy over more bits.

When all the entropy we have is available before algorithm execution begins, we really have a stream cipher. However some crypto system designs allow for the addition of entropy during execution, in which case it is not a stream cipher equivalent and cannot be used as one. Stream cipher and CSPRNG design is thus closely related.

Several CSPRNGs have been standardized. For example [18],

- FIPS 186-2
- NIST SP 800-90: Hash_DRBG, HMAC_DRBG, CTR_DRBG and Dual_EC_DRBG.
- ANSI X9.17-1985 Appendix C
- ANSI X9.31-1998 Appendix A.2.4
- ANSI X9.62-1998 Annex A.4, obsoleted by ANSI X9.62-2005, Annex D (HMAC_DRBG)

4 An overview of Key management

Symmetric-key cryptography (SKC) has advantage over asymmetric-key cryptography (public key cryptography, PKC) mainly due to faster encryption and relatively smaller key size. The same key is used to encrypt and decrypt the message; this implies that before encrypting the message, the key should be distributed among the participants to have a secure communication. Therefore, SKC poses the problem of key distribution. In practice, there are other problems, like generating secure and strong keys, storing them in reliable manner, distribution etc, that need to be addressed well.

Key management is the title that covers all the problems stated above [1]. To achieve secure key distribution, both SKC and PKC are used in practice. As keys are the fundamental to many primitives, like, authentication, confidentiality, authorization, integrity etc, there are various types on keys, mentioned in the literature [1] and each of these requires a different level of security and management.

Following are the types of keys:

- **Private signature keys** are the private keys of asymmetric (public) key pairs that are used by public key algorithms to generate digital signatures with possible long-term implications. When properly handled, private signature keys can be used to provide authentication, integrity and non-repudiation.
- **Public signature verification key** is the public key of an asymmetric (public) key pair that is used by a public key algorithm to verify digital signatures, either to authenticate a user's identity, to determine the integrity of the data, for non-repudiation, or a combination thereof.
- **Symmetric authentication key**s are used with symmetric key algorithms to provide assurance of the integrity and source of messages, communication sessions, or stored data.
- **Private authentication key** is the private key of an asymmetric (public) key pair that is used with a public key algorithm to provide assurance as to the integrity of information, and the identity of the originating entity or the source of messages, communication sessions, or stored data.
- **Public authentication key** is the public key of an asymmetric (public) key pair that is used with a public key algorithm to determine the integrity of information and to authenticate the identity of entities, or the source of messages, communication sessions, or stored data.
- **Symmetric data encryption key**s are used with symmetric key algorithms to apply confidentiality protection to information.
- **Symmetric key wrapping key**s are used to encrypt other keys using symmetric key algorithms. Key wrapping keys are also known as key encrypting keys.

- **Symmetric and asymmetric random number generation keys** are keys used to generate random numbers.
- **Symmetric master key** is used to derive other symmetric keys (e.g., data encryption keys, key wrapping keys, or authentication keys) using symmetric cryptographic methods.
- **Private Key transport key**s are the private keys of asymmetric (public) key pairs that are used to decrypt keys that have been encrypted with the associated public key using a public key algorithm. Key transport keys are usually used to establish keys (e.g., key wrapping keys, data encryption keys or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors).
- **Public key transport key**s are the public keys of asymmetric (public) key pairs that are used to encrypt keys using a public key algorithm. These keys are used to establish keys (e.g., key wrapping keys, data encryption keys or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors).
- **Symmetric key agreement key**s are used to establish keys (e.g., key wrapping keys, data encryption keys, or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors) using a symmetric key agreement algorithm.
- **Private static key agreement key**s are the private keys of asymmetric (public) key pairs that are used to establish keys (e.g., key wrapping keys, data encryption keys, or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors).
- **Public static key agreement key**s are the public keys of asymmetric (public) key pairs that are used to establish keys (e.g., key wrapping keys, data encryption keys, or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors).
- **Private ephemeral key agreement key**s are the private keys of asymmetric (public) key pairs that are used only once10 to establish one or more keys (e.g., key wrapping keys, data encryption keys, or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors).
- **Public ephemeral key agreement key**s are the public keys of asymmetric key pairs that are used in a single key establishment transaction to establish one or more keys (e.g., key wrapping keys, data encryption keys, or MAC keys) and, optionally, other keying material (e.g., Initialization Vectors).
- **Symmetric authorization key**s are used to provide privileges to an entity using a symmetric cryptographic method. The authorization key is known by the entity responsible for monitoring and granting access privileges for authorized entities and by the entity seeking access to resources.
- **Private authorization key** is the private key of an asymmetric (public) key pair that is used to provide privileges to an entity.
- **Public authorization key** is the public key of an asymmetric (public) key pair that is used to verify privileges for an entity that knows the associated private authorization key.

The reason for defining so many types is the observation that same key should not be used for various different purposes. It weakens the security of the key. However, for most of our purpose, we care more about *Symmetric data encryption key*, *Public key transport key*, *Symmetric key wrapping key*, *Private signature key* and *Public signature verification key*. For the sake of simplicity and clarity, we divide the above keys into two sets of keys - symmetric key (Symmetric data encryption key, Symmetric key wrapping key) and public-key pair (Public key transport key, Private signature key and Public signature verification key).

In the following sections, we describe a widely used symmetric key exchange protocol, called Diffie-Hellman Key Exchange Protocol, followed the by Public Key Infrastructure, which details the various components required to used public-key cryptography in practice. We will also provide few pointers to Java implementation of PKC/PKI in Section 9.

4.1 Diffie-Hellman Key Exchange Protocol

In 1976, Diffie and Hellman presented their seminal paper to lay the foundations of public-key cryptography [21]. They proposed a method to publicly exchange the key, now popularly known as *DH Key Exchange Protocol*. The protocol allows two participants A and B to generate a *secret key* K over an insecure channel. The shared key K , then, can be used as symmetric key to be used in other cryptographic schemes. The protocol goes as follows [22]:

Let G be a cyclic group of order $|G|$ and generator g . In order to generate a secret key, A and B secretly choose numbers s_A and s_B respectively, randomly from the interval $[0,|G|]$. A computes $K_A = g^{s_A}$ and B computes $K_B = g^{s_B}$. They exchange the numbers, so generated, to calculate $K_{AB} = K_B^{s_A} = K_{BA} = K_A^{s_B} = K$. In this way, both share a secret over an insecure, public channel. There have been many proposals on the choice of different groups that can be used in DH protocol [22], for example, *multiplicative groups of large finite fields, multiplicative group of residues modulo a composite number, elliptic curves over finite fields, the Jacobian of a hyper-elliptic curve over a finite field, and the class group of imaginary quadratic fields*.

In order for an adversary to know the secret key K, the obvious (but not easiest) method is to calculate the *discrete log,* i.e. given g and a, calculate s such that $g^s = a$. There are few methods, like Pollard's *rho-methods, lambda-method* to calculate discrete log and, therefore, it is suggested to use a large exponents in the group Z_p^* with randomly chosen p [22]. For other practical issues related to, for example, life and size of key please refer to NIST recommendations in [1].

4.2 Public Key Infrastructure

Public-key infrastructure (PKI) refers to the functionalities and components that enable a set of users (more precisely, entities) to exchange information securely and in an authentic way over an insecure medium, like Internet. The need of PKI arises due to the difficulty of distributing secret key among parties to communicate securely, which involves confidentiality and integrity of data, and authentication of parties involved in the communication.

A PKI mainly consists of the following components:

1. A digital certificate that binds the identity of the user to its public key.

2. A certificate authority (CA) that issues and verifies digital certificate. A certificate includes the public key or information about the public key.
3. A registration authority (RA) that acts as the verifier for the certificate authority before a digital certificate is issued to a requester.
4. One or more directories where the certificates (with their public keys) are held.
5. A certificate management system.

A digital certificate (DC) typically consists of user's public key, credentials related to their rights and privileges, information about the guaranteeing authority and a time range of validity. It is digitally signed by a trusted third part (CA) and the industry standard for DC is *X.509*. One example of widely accepted CA is Verisign. DCs are published via directory service (LDAP) that can be queried by users to know the status of any DC. The same is useful to know if a certificate is still valid or has been revoked. PGP is an example of PKI, which is based on mutual trust among the users.

4.3 Manual configuration

In this section, we provide details on some procedures or guidelines to perform secure communication in the absence of an in-built infrastructure for providing security. If there is no security infrastructure available, the nodes have to manage security mechanisms manually. Under such conditions, we do not assume the presence of PKI related functioning and nodes try to establish secure communication with minimum but sufficient security. We can envisage the following scenarios:

- There is a node X that can be considered to be present all the time. The nodes share a symmetric key with this node. Whenever any node A wants to communicate with other node B, they can agree on a shared key K by using node X as an intermediate node. Any new node needs to contact node X to obtain the shared key.
- If the nodes form a group and there is a shared common key to communicate within the group, every new node has to contact one existing member of the group to get the common key.

The first point assumes the presence of a trusted node that is present all the time during the existence of the network. This may not be valid always. The second point avoids the presence of such a node, but introduces a problem to be solved. Whenever a node leaves the group, the common shared key has to be changed. This involves agreeing on a key and distributing that key to all the present members of the group. If the group exists for a short period of time and there is not much mobility, the second scheme provides a good option. Therefore, depending on the scenario, one may use any of the above mentioned procedures

5 Overview of Access Control

A system of ACEs is heterogeneous in nature, and therefore is susceptible to security threats common to all open distributed systems. The autonomy enables one to make independent choices to participate in creating or using a service; this does not preclude an ACE from unrestrained selfish behaviour. Access control pertains to efforts in minimising such undesirable outcomes in the system.

Authorised entities can safely collaborate to offer/access services, while entities with less authority may be restricted in their activities. The process of authorisation comprises two components, authentication and access control: the first is the process of verifying an entity's identity that it is truly who it claims to be; the second is the process of deciding what resources or services that it can access, after authentication.

Traditionally, in systems where all players are well-known in advance, these components have been treated as distinct and implemented separately. For example, Kerberos [36] is an authentication protocol that uses a third party trusted entity, an authentication server(AS), to establish a client's identity with an application server. The application server would thereafter determine what operations or services the client is allowed to access, typically defined using an access control list (ACL) [38].

However open distributed systems have more difficulties in defining and implementing authorisation rules than traditional systems. The heterogeneity and dynamics of such system introduces new players, and as a result, new opportunities for collaboration in creating services. The unknown entities may pose a higher threat of security; however if access control were extremely strict, there is a possibility of wasted opportunities.

In light of such situations, the idea of *trust management* [32][33] advocates integrating authentication and access control for increased scalability, better delegation, and improved expressibility. Safely verifying an entity's identity is on itself insufficient to determine the access rights that should be given to it, especially when the entity is unknown. The appropriate mechanism should require the entity to carry some credentials, in the form of tickets, certificates or digital signatures that introduces the entity and describes its access permissions to another party.

5.1 Public Key cryptography

The *X.509* [31] standard for Public Key Infrastructure (PKI) describes digital certificates as a form of credentials that an entity carries. The essence of the standard relies on the asymmetric property of the public key cryptography. A pair of keys, public and private, is used; messages encrypted with one key can only be decrypted using the counterpart, and knowledge of one key cannot be used to reverse-engineer the counterpart.

Therefore an entity that (digitally) signs a message, by encrypting with its private key, assures message integrity and non-repudiation.

An entity obtains a certificate from a certificate authority (CA), a third party trusted server, to authenticate itself with other entities. The certificate is a digitally signed message from the CA with its private key, such that the integrity is assured. Like Kerberos [36] earlier, this mechanism relies on implicitly trusting a third-party server where certifying activities are centralised, both of which leave much to be desired in open distributed systems. By binding an entity's identity to its public key in a X.509 certificate, the standard defines the authentication protocol; it does not dictate any access permission rules.

Pretty Good Privacy (PGP) [1] certificate systems relieve the need for a centralised server. Instead certificates are issued by entities themselves, and trust is developed when one produces a certificate signed by another trusted entity; thus promoting transitivity of trust.

5.2 Trust Management

The SPKI/SDSI [34] standard uses certificates to authenticate and authorise; it binds the entity's public key with its authorisation within the certificate. A verifier reading a certificate learns the permissions of the carrying entity, and determines its access rights accordingly. Along these lines, the PolicyMaker and KeyNote [32][33] implement a *trust-management engine* that offers more programmability to express privileges, restrictions and policies. An entity may delegate authority on its services or resources to several external certificate issuers. Any other entity that requests for access must produce these certificates to establish trust; the trust-management engine checks for the request's compliance with local policies which hold the highest overriding authority. These systems offer programmability that grants application developers expressibility in defining policies while maintaining application-independence in the implementation.

On a different level of abstraction, Abdul-Rahman *et. al.* proposed a decentralised trust model [30] that is applicable in an open, distributed systems, in which entities manage their own trusts. The model allows for recommendation-based build up of trusts, however argues against direct transitivity.

There are two types of trusts: a *direct trust* between A and B; or *recommendation trust* that describes how much A trusts B as a good recommender. Trust is transitive upon recommendation when some conditions are met: B sends A an explicit recommendation about C; A trusts B as a good recommender; and A makes its choice on how much to trust C. Trust is multi-category, and multi-valued within each category.

A closely related concept to trust management is that of reputation management [39]. An entity's reputation is its asset that can be used to obtain some future services or resources; therefore it strives to build a reputation by serving others in return. All entities report their observations or experiences of interactions with other entities to the system which dynamically rates the reputation. The system is a repository of information that helps entities assess and manage risks when interacting with others.

In autonomous distributed systems, these models of trust allow entities that may be complete strangers to obtain more information about each other beyond the basic binary trust (complete trust or mistrust), to evaluate the risks involved, to weigh the advantages of a possible collaboration, and to finally make a rational decision.

5.3 Single sign-on

When a client has to obtain services from a server, it has to authenticate itself for service, typically using a login and password combination. When the number of servers is big, this becomes a tedious repetitive process, in addition to the increased security risk of submitting password many times and user's password fatigue. Single sign-on (SSO) is an idea that a client authenticates itself once to the distributed system and the proof of authentication is automatically propagated to each server requesting verification.

The simplest non-conventional SSO system is one that uses scripts [37] at client's workstation, such that the process of submitting login/password pair is simulated by the script for each login. Although this requires least change to the underlying system mechanism, it poses serious security threat because the password has to be stored in the script and transmitted each time. The alternative is ticket-based SSO system, such as Kerberos [36]. When a user first logs in, the authentication server (AS) sends it a ticket-granting ticket. User then submits this ticket to a ticket granting server (TGS) to request for service tickets for each application server that it needs access to. All the tickets cannot be tampered with, nor replicated without being discovered; therefore are strong digital credentials. The user's password is only requested once, and the workstation does not store this information once the tickets are obtained.

Between the commercially available SSO systems, using a centralised authentication server is a common practice but the underlying protocol and how credentials are exchanged may differ.

The X.509 PKI's digital certificates are an alternative form of credentials to the Kerberos tickets. The Cornell single sign-on (CorSSO) [35] is an effort to distribute the identity checking process to several authentication servers that reside in separate domains.

6 Practical considerations: Processing overhead of the basic ciphering algorithms

6.1 Introduction

This section presents an assessment of the processing overhead of the basic ciphering algorithms and evaluates the feasibility of deploying them on mobile devices, which are characterized by limited processing resources. The most prominent ciphering algorithms such as Data Encryption Standard (DES), Advanced Encryption Standard (AES), Message Digest (MD5) and Secure Hash Algorithm 1 (SHA-1), are presented and analyzed. The analysis considers their processing requirements (referred to as processing overhead) so as to facilitate a comparative performance evaluation, independently of specific implementations and across different algorithms. This analysis is incorporated in a simulation study that attempts to assess the feasibility of deploying ciphering algorithms on mobile devices and networks. A simple analytic model of a mobile device that performs security functions is also derived yielding analytical results in line with the simulative ones and providing for an alternative approach for assessing the performance of security deployment on mobile devices and networks. In the following section, the processing overheads introduced by the above algorithms are examined, quantifying the impact of security on the underlying devices.

This section is of paramount importance for the successful development and integration of basic security services in the CASCADAS framework, in which we assume that heterogeneous devices, some of them with very limited resources (see also the application case studies we discuss later on the in the Deliverable), will need to implement and carry on securely their activities.

6.1.1 DES and 3DES

The DES cipher uses a key of 56 bits, and a block of 64 bits. Since DES is a Feistel cipher, it requires the same amount of processing for both encryption and decryption. 3DES results from a triple execution of DES and, thus, requires three times more processing. Let T_{DES} and T_{3DES} denote the number of operations required for encrypting one block of user data with DES and 3DES respectively. The analysis of the two ciphers that appears in [2] has shown that T_{DES}=2697 and T_{3DES}=8091. Let S_d denote the size of an unencrypted user data packet and let $U_{DES}(S_d)$ and $U_{3DES}(S_d)$ denote the corresponding numbers of operations required to encrypt it with DES and 3DES. Then clearly,

$$U_{DES}(S_d) = \left\lceil \frac{8 \times S_d}{64} \right\rceil \times T_{DES} \quad (1) \;, \qquad U_{3DES}(S_d) = \left\lceil \frac{8 \times S_d}{64} \right\rceil \times T_{3DES} \quad (2)$$

($\lceil\rceil$ denotes the ceil function). Consider now a processor that can perform C_P Millions Instruction Per Second (MIPS), and let t_{DES} (S_d, C_P) and t_{3DES} (S_d, C_P) denote the time required by this processor for encrypting one user packet of length S_d with DES and 3DES respectively. Then,

$$t_{DES}(S_d, C_P) = \left\lceil \frac{8 \times S_d}{64} \right\rceil \times \frac{T_{DES}}{C_P} \quad (3), \qquad t_{3DES}(S_d, C_P) = \left\lceil \frac{8 \times S_d}{64} \right\rceil \times \frac{T_{3DES}}{C_P} \quad (4)$$

6.1.2 AES

Rijndael is an iterated block cipher with a block of length $4N_b$ bytes (or N_b (32-bit) words) and a variable key of length $4N_k$ bytes. The encryption of each block of the data involves the following: (a) an initialization phase; (b) N_r-1 iterations of the basic encryption processing of the algorithm; (c) a finalization phase. The version of the Rijndael algorithm that was integrated as part of the AES encryption standard [18] uses a block of 128 bits (i.e., N_b=4) and a key of 128, 192, or 256 bits (i.e., N_k=4, 6, or 8). Depending on the selected key length, the AES standard defines the number of rounds for phase (b) as follows: N_r(128)=10, N_r(196)=12, N_r(256)=14. In [13] the authors have analyzed the Rijndael encryption and have derived simple expressions for T_{Rij}, the computational effort required for encrypting one block of data with this particular cipher. They have expressed this computational effort as a function of the block size, the key size, and the number of processing cycles required for performing basic operations such as a byte-wise AND (T_a), a byte-wise OR (T_o), and a byte-wise shift (T_s). The resulting general expression is:

$$\begin{aligned} &T_{Rij\text{-}ENC} = \\ &(46\ N_b\ N_r\ \text{-}30\ N_b)\ T_a + \\ &[31\ N_b\ N_r + 12\ (N_r\ \text{-}1) - 20\ N_b]\ T_o + \\ &[64\ N_b\ N_r + 96\ (N_r\ \text{-}1) - 61\ N_b)]\ T_s \end{aligned}$$

By assuming that each basic operation requires one processing cycle, i.e., T_a=T_o=T_s=1, we can derive the corresponding number of processing cycles required for encrypting one block of data with each one of the three standardized flavours of AES (for different key lengths):

$$T_{AES\text{-}ENC}(128)=6168$$

$$T_{AES\text{-}ENC}(192)=7512$$

$$T_{AES\text{-}ENC}(256)=8856$$

Using the same definitions and notation as with DES and 3DES, we can write:

$$U_{AES-ENC}(S_d)=\left\lceil \frac{8\times S_d}{128} \right\rceil \times T_{AES-ENC} \quad (5)$$

and,

$$t_{AES-ENC}(S_d, C_P)=\left\lceil \frac{8\times S_d}{128} \right\rceil \times \frac{T_{AES-ENC}}{C_P}, \quad (6)$$

where $T_{AES\text{-}ENC}$ can take either of the following values $T_{AES\text{-}ENC}$ (128), $T_{AES\text{-}ENC}$ (196), or $T_{AES\text{-}ENC}$ (256), depending on the selected key length.

An important difference of Rijndael as compared to other ciphers such as DES and 3DES, is that Rijndael has a non-Feistel structure, meaning that the decryption process makes use of partially different code, which allows for only partial re-use of the encoding circuitry that implements the cipher. The implementation differences are identified in phase (b) of the decryption code, and in particular in the InvMixColumns operation, which uses a different polynomial structure as compared to the corresponding MixColumns operation of the encryption code and, thus, leads to an increased complexity for the decryption. By using the analysis presented in [13], we can obtain an expression for the number of processing cycles for decrypting one block of data.

$$T_{Rij\text{-}DEC} = T_{Rij} + 96\, N_b\, T_a + (Nr-1)\times(72\, N_b\, T_o - 32N_b\, T_s) \quad (7)$$

The above expression (Eq. (7)) points to the fact that the Rijndael decryption is computationally more expensive than the encryption (the actual difference being 96 N_b T_a + 72 N_b T_o - 32N_b T_s operations for each of the Nr-1 rounds of phase (b)). Using this expression, we can obtain the corresponding number of processing cycles required for decrypting one block of data with each one of the three standardized flavours of AES (for different key lengths):

$$T_{AES\text{-}DEC}(128)=10992$$

$$T_{AES\text{-}DEC}(192)=13408$$

$$T_{AES\text{-}DEC}(256)=15824$$

From these values, we can obtain $U_{AES\text{-}DEC}$ (S_d) and $t_{AES\text{-}DEC}$ (S_d) similarly to the encryption case. Notice that the number of operations required for the decryption is significantly higher than for the encryption. Thus, there have been some efforts for reducing this asymmetry through faster implementations (see for example [13, 14, 15]).

6.1.3 HMAC-MD5

A common MAC algorithm is the combined HMAC-MD5. The first step in the MD5 algorithm is padding the original message for its size to become a multiple of 512 bits with the last 64 bits of the last block indicating the length of the message. Then, the algorithm produces a 128-bit hash value. The hash computation and the hash verification in MD5 are equivalent procedures and, thus, they consume the same amount of time. The total number of operations required for MD5 processing per block (512 bits), T_{MD5} , is 720 plus 24 operations for initialization and termination [1].

The combined HMAC-MD5 algorithm is formulated as follows:

$$MD5(K_o, MD5(K_i, Text))$$

where

$$K_i = Key \oplus ipad$$

$$K_o = Key \oplus opad$$

K_i and K_o are two extended forms (512-bit) of the input *Key,* which are generated by "exclusive oring" the *Key* with *ipad* (the inner padding (512 bits)) and *opad* (the outer padding (512 bits)) respectively. *Key* is an arbitrary size secret key shared by a sender and a receiver, and ⊕ denotes the XOR operation.

For a user packet of size S_d bytes, the number of input blocks for the inner MD5, n_k , is

$$n_k = \left(\frac{8 \times S_d + s_p + s_s + K}{512} \right) \quad (8)$$

where s_p is the size (in bits) of the padding field, s_s is the size (in bits) of the field that specifies the message length, and K is the size (in bits) of the extra appended inner form of the key.

In the outer MD5, the output of the inner MD5 (128-bit digest) is appended to K_o. According to MD5, this is padded to two 512-bit blocks. Thus, the total number of operations in applying the combined HMAC-MD5, $T_{HMAC\text{-}MD5}(n_k)$, and, $U_{HMAC\text{-}MD5}(S_d)$, as a function of the number of input blocks n_k, and the user packet size S_d , are

$$T_{HMAC\text{-}MD5}(n_k) = 32 + (2 + n_k) \times 744 \quad (9)$$

$$U_{HMAC-MD5}(S_d) = 2264 + 744 \times \left(\left\lceil \frac{(8 \times S_d) + 64}{512} \right\rceil \right) \qquad (10)$$

The factor 32 in Eq. (9) derives from the XOR operations performed to produce the inner and the outer keys, K_i and K_o . Specifically, it results from the division of the size of XOR operands (512 bits) by the word length supported by the processor (i.e. it is assumed to be 32 bits). The outcome of the division (i.e., 16) is multiplied by 2, as the XOR operation occurs twice (one for K_i and one for K_o). Finally, the required authentication and verification time for HMAC-MD5, $t_{HMAC-MD5}$ (n_k ,C_P), as a function of the number of input blocks and the processor speed , is

$$t_{HMAC-MD5}(n_k, C_P) = \frac{32 + (2 + n_k) \times 744}{C_P} \qquad (11)$$

6.1.4 HMAC-SHA-1

The functionality of SHA-1 is similar to that of MD5, and both algorithms use the same block size and padding procedure. However, SHA-1 employs five 4-byte intermediate registers instead of four that are used in MD5 and, thus, the produced message digest is 160-bits long. For each input block of size of 512 bits, the total number of operations required for SHA-1 processing, T_{SHA-1} , is 900 plus 210 operations for initialization and termination [1].

The combined HMAC-SHA-1 algorithm is formulated as follows:

$$SHA\text{-}1(K_o, SHA\text{-}1(K_i , Text)),$$

where the keys K_i and K_o are computed similar to those used in HMAC-MD5 algorithm.

For an input text of size S_d bytes, the number of input blocks for the inner SHA-1, n_k , is given by Eq. (8). In the outer SHA-1, the output of the inner SHA-1 (160-bit digest) is appended to K_o , and the outcome is padded to two 512-bit blocks. Thus, the total number of operations required for applying the combined HMAC-SHA-1 processing, $T_{HMAC-SHA-1}(n_k)$, and, $U_{HMAC-SHA-1}(S_d)$, as a function of the number of input blocks n_k and the user packet size S_d , are given similarly to the HMAC-MD5.

$$T_{HMAC-SHA-1}(n_k) = 32 + (2 + n_k) \times 1110 \qquad (12)$$

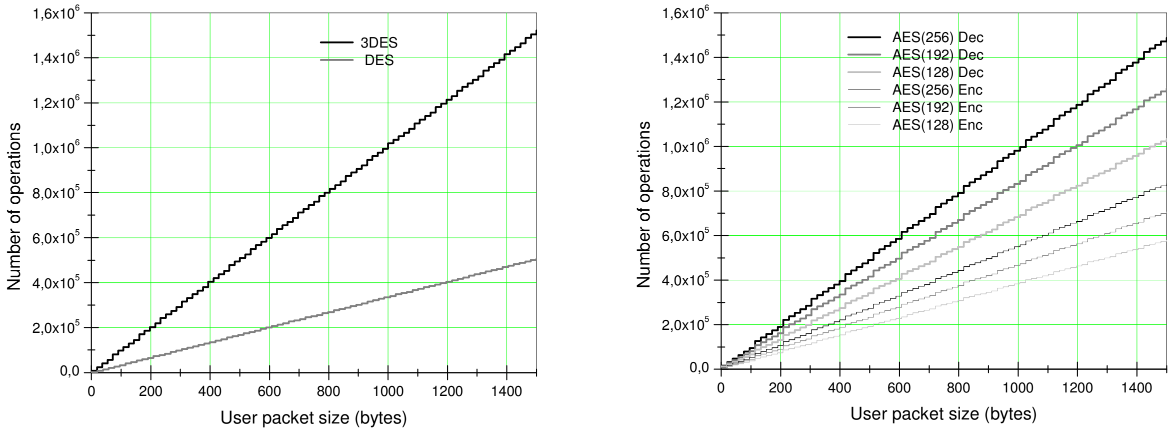
$$U_{HMAC-SHA-1}(S_d) = 3362 + 1110 \times \left(\left\lceil \frac{(8 \times S_d) + 64}{512} \right\rceil \right) \qquad (13)$$

Likewise, the required authentication and verification time for HMAC-SHA-1, $t_{HMAC-SHA-1}$ *(n_k ,C_P),* as a function of the number of input blocks and the processor speed, is

$$t_{HMAC-SHA-1}\left(n_k, C_P\right) = \frac{32 + \left(2 + n_k\right) \times 1110}{C_P} \quad (14)$$

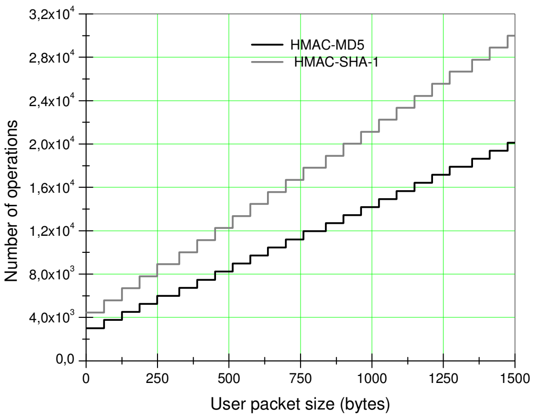
6.1.5 Comparison of the processing overhead

This section presents a comparison of the processing overhead of the previously analyzed algorithms. **Error! Reference source not found.** shows the total number of operations required by a processor to performs (a) DES and 3DES, (b) AES with variable key length for both encryption and decryption process, and (c) the combined HMAC-MD5 and HMAC-SHA-1 algorithms, as a function of user packet size; the plotted values have been obtained from Eqs (1), (2), (5), (10) and (13). The various padding operations that are employed by all algorithms produce the stepped behaviour that appears in the graphs. The height of step depends on the selected block size of the employed algorithm and, thus, the authentication algorithms exhibit bigger steps than the confidentiality algorithms (the authentication algorithms have a block size of 512 bits, whereas the confidentiality algorithms have block sizes of 64 or 128 bits). From the presented figures it becomes clear that the confidentiality services consume significantly more processing resources than the authentication. 3DES and AES decryption with 256-bit key impose the highest processing overhead and are followed by AES decryption with shorter key lengths (192, 128-bit), AES encryption with key lengths 256, 192, 128-bit, and the DES algorithm. Finally, regarding authentication services the combined HMAC-SHA-1 algorithm requires more processing resources compared to the HMAC-MD5.



(a)

(b)



(c)

Fig. 1: The number of operations required to perform (a) DES and 3DES, (b) AES with variable key length for both encryption and decryption process, and (c) HMAC-MD5 and HMAC-SHA-1 as a function of user packet size.

6.2 Simulation study

Fig. 2 depicts a block diagram of the mobile device that is considered in the following simulation study. The model consists of the following components: (i) a traffic generator for the creation of non-real time traffic according to the parameters that are defined in a next paragraph; (ii) a security processor queue where user data packets accumulate before entering the processor that applies the cryptographic algorithm; (iii) a transmitter queue where the encrypted packets accumulate before being transmitted over the wireless channel.

The conduct of simulation is useful because of a variety of parameters that influence the system performance except for the generated traffic load and the service rate of the queues. More specifically, the security schemes that require significant processing resources mainly delay data transmission in the security processor queue. On the other hand, in the "lighter" security schemes, data packets spend more time in the transmitter queue.

The employed simulated traffic represents non-real time user traffic according to the reference model defined by the 3GPP in [16]. It is assumed that there exists an active user that generates packet sessions. Each session involves bursty sequences of packets. The mean user data rate is denoted λ_{data} and ranges from 128 Kbit/s to 2 Mbit/s. Packet inter-arrival times between subsequent user packets in a packet call are modeled by an i.i.d. random variable that follows an exponential distribution with parameter μ_d. The sizes of user packets are modeled by an i.i.d. random variable S_d that follows the truncated Pareto distribution $f_{Sd}(x)$:

$$f_{S_d}(x) = \begin{cases} \dfrac{ak^a}{x^{a+1}}, k \leq x < m \\ \left(\dfrac{k}{m}\right)^a, x = m \end{cases} \qquad (15)$$

The parameters k and m define the minimum and the maximum user data packets respectively and the parameter *a* defines the skewness of the distribution (the default values are a=1.1, k=81.5 bytes and m= 66666 bytes [16]). The average packet size is μ_n=480 bytes, and the radio channel capacity is 2 Mbps. The mobile device is assumed to be equipped with an embedded processor with a processing rate C_p in the range of 100 to 500 Millions of Instructions Per Second (MIPS) [17]. Table 2 summarizes the values of the basic simulation parameters.

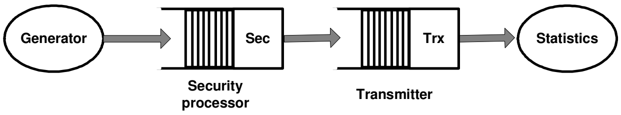


Fig. 2: Model of a security equipped processor.

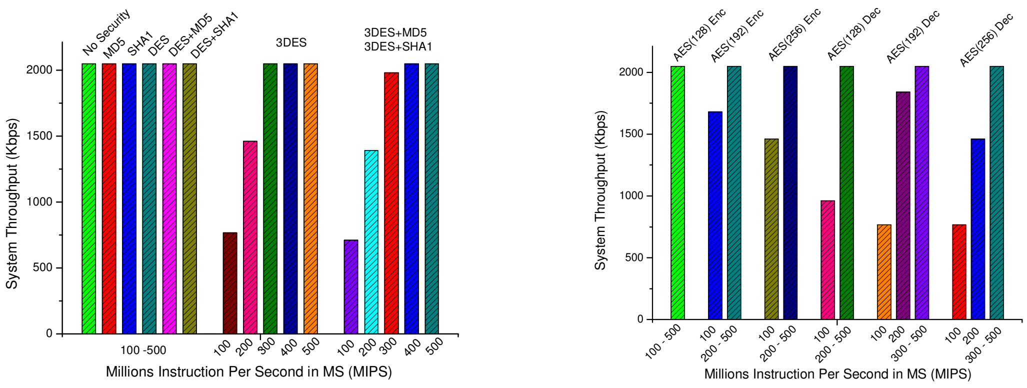
A total of twenty seven (27) different security scenarios are considered. They include several different cryptographic algorithms that provide different levels of security: (i) no security, (ii) pure confidentiality (DES, 3DES and AES with variable key length for the encryption and decryption process), pure authentication (MD5 and SHA1), and combined confidentiality and authentication (DES+MD5, 3DES+MD5, DES+SHA1, 3DES+SHA1, AES(128)Enc+MD5, AES(128)Enc+SHA1, AES(192)Enc+MD5, etc). The evaluation of the different scenarios is based on the following performance metrics: (i) the system throughput, and (ii) the packet latency. In the next paragraphs we summarize our observations from the simulation experiments.

Simulation parameters	Base values
Mean data rate λ_{data}	128 Kbit/s – 2 Mbps
MS processing speed C_{MS}	100 – 500 MIPS
Average size of datagram μ_n	480 bytes
Radio channel capacity	2 Mbps

Table 1: Simulation parameters setting

In the majority of the employed security scenarios, the encryption and decryption are symmetric processes and, thus, they consume the same amount of processing. For that reason we have selected to develop a simulation model that represents only the encryption, and use it as a basis for the accomplished performance evaluation and the comparison of the different security scenarios. However, as mentioned previously, the AES cipher has a non-Feistel structure and the decryption is computationally more expensive than encryption. Thus, both AES processes (encryption and decryption) with all the possible key lengths (i.e., 128, 192, 256) are applied to the developed simulation model. This fact facilitates the assessment of the computational difference between the encryption and the decryption process in AES, and the comparison of both AES processes with the rest of the employed ciphering algorithms.

Fig. 3 depicts the system throughput as a function of the processing speed of the mobile device for the above security scenarios. One may observe that the more "lightweight" security schemes like MD5, SHA1, DES, DES+MD5 and DES+SHA1 do not degrade the system throughput, as they add a rather limited amount of processing (see Fig. 3 (a)). This points to the fact that a processing rate of 100 MIPS and above should be enough for handling the added processing of these lightweight schemes. For the above combinations of security schemes and processing rates, the bottleneck in terms of throughput is dictated by the capacity of the radio channel. Stronger encryption schemes like 3DES, 3DES+MD5 and 3DES+SHA1 provide for an increased resistance against attacks but pose higher processing requirements and, thus, reduce the system throughput when the MS processing rate is below 300 MIPS (which appears to be the borderline minimum for employing these schemes).



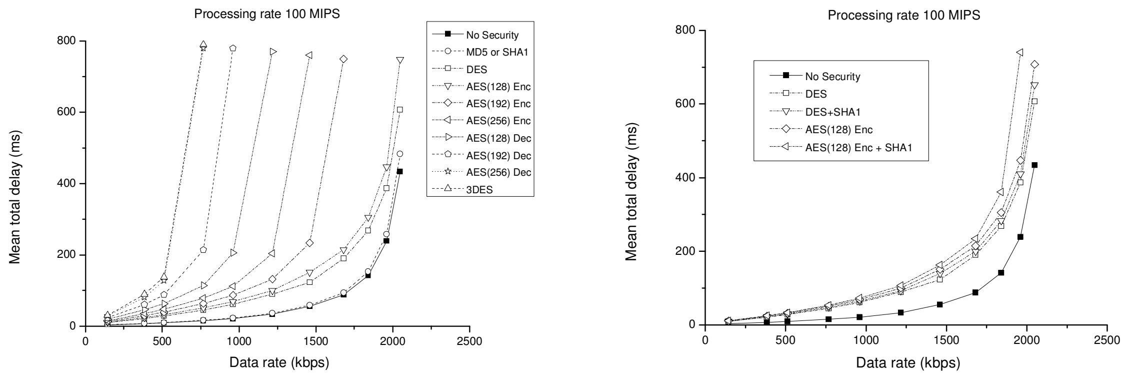
(a) (b)

Fig. 3: System throughput as a function of the processing speed of the mobile device for different security scenarios like (a) no security, MD5, SHA1, DES, DES+MD5, DES+SHA1, 3DES, 3DES+MD5 and 3DES+SHA1, (b) AES with different key size for both encryption and decryption process.

As with 3DES, the AES protection is resource consuming and, thus, can lower the system throughput when there isn't sufficient processing capability at the mobile device (see Fig. 3 (b)). The throughput, however, is generally higher under AES than under 3DES, despite the fact that AES provides for a much stronger encryption than 3DES [2,4]. The encryption process of AES presents higher throughput values compared to the decryption.

The lightest flavour of AES, i.e., the one that provides encryption with a key length of 128 bits, has almost no effect on the system's throughput. Increasing the key length, however, puts more strain on the processor and this can translate into reduced throughput. Combining confidentiality with authentication services by adding MD5 or SHA-1 to AES increases even more the strain on the processor. This extra strain is, however, relatively small as compared to the one imposed by the encryption scheme and thus is hardly visible on the figures.

Except for its impact on the system's throughput, a security scheme increases the total delay for transmitting a user packet. Fig. 4 shows the total delay as a function of the user data rate for the various security schemes and a processing rate of 100 MIPS. Sole application of authentication services, like MD5 and SHA1, hardly has an impact on the total delay (see Fig. 4 (a)). The same applies for DES and AES encryption with 128-bit key (labeled AES(128)Enc in the corresponding figure) both of which have a similar behavior, and add marginally to the total delay as compared to the no-security scenario. The AES encryption with larger key lengths (AES(192)Enc, AES(256)Enc), the AES decryption with variable key lengths (AES(128)Dec, AES(192)Dec, AES(256)Dec) and 3DES have stronger impact on the total delay. Moreover, these scenarios under sufficiently high user data rates lead to excessive delay values, which point to the fact that the user data rate has exceeded the maximum capacity of the MS.

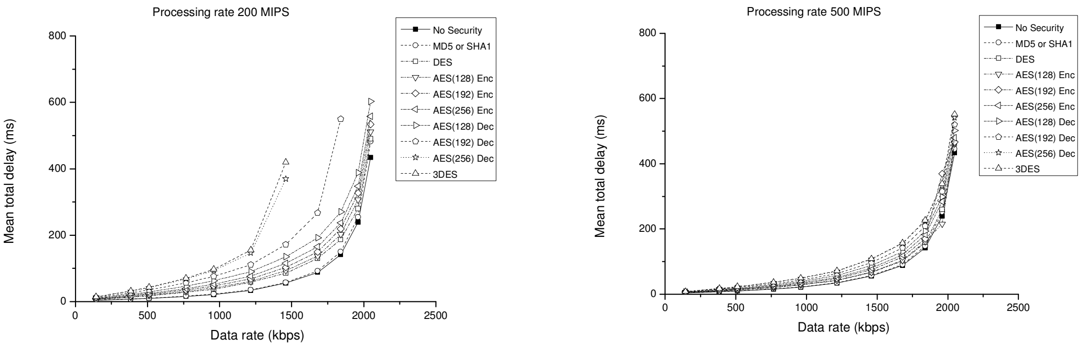


(a) (b)

Fig. 4: Mean total delay as a function of mean data rate for 100 MIPS processing rate at the MS and (a) MD5, SHA1, DES, AES and 3DES (b) DES, DES+SHA1, AES and AES+SHA1

Fig. 4 (b) presents the total delay of the combined confidentiality and authentication security services using DES+SHA1 and AES(128)Enc+SHA1 algorithms, as a function of the user data rate and for a processing rate of 100 MIPS. It compares the above total delay values to the total delay of the pure confidentiality security services using DES and AES(128)Enc algorithms, respectively. Observed that the addition of authentication security services hardly increases the total packet delay values, since authentication represents a relatively lightweight (from the processing overhead point of view) security service.

For a greater MS processing rate of 200 MIPS, there is a similar qualitative behavior as with the abovementioned 100 MIPS case. However, the absolute delay values become smaller, owing to the shorter time spent on the IPsec processor queue. In fact, with such a processing rate, some of the lightweight security schemes incur a total delay that approaches the one of the no security scenario (see Fig. 5 (a)). Increasing the MS processing rate further to 500 MIPS (Fig. 5 (b)), pushes the delay curves of the various security schemes very close to the no security curve, which means that in this case, IPsec has almost a negligible impact on the system's performance with respect to the delay.



(a) (b)

Fig. 5: Mean total delay as a function of mean data rate for MD5, SHA1, DES, AES and 3DES and (a) 200 (b) 500 MIPS processing rate at the MS.

6.3 Analytic model of an IPsec-equipped MS

The goal of this Section Is to provide a case study that is oriented to an enhanced version of the basic components of the CASCADAS framework: in most commercial applications, end-to-end security is required in order to obtain increased protection against external and internal attacks. IPSec is a de-facto standard (we do not present un-necessary details on IPSec in this Deliverable) that lacks any substantial validation when executed in a resource-constrained environment. For this reason, we develop a simple analytic model for the abstract mobile device that is depicted in Fig. 2. The analysis is carried out by modeling each one of the two queues of the tandem as an independent M/G/1 queue. The analysis aims at both verifying the simulation results and providing a faster alternative to them.

6.3.1 First queue (processor)

The first queue is an M/G/1 queue with the following characteristics: (i) a Poisson arrival process of rate λ for modeling the arrivals of data packets from the user; (ii) i.i.d. service times X_1 with expected value $E\{X_1\}$ and expected square value $E\{X_1^2\}$. Let μ_1 denote the constant service rate of the server of the first queue (to be defined in detail shortly). Then $E\{X_1\}$ and $E\{X_1^2\}$ can be written as functions of μ_1 and $f_{Sd}(x)$, the probability density function of user packets which are assumed to be following a truncated Pareto distribution with parameters k, m, a. Thus,

$$E\{X_1\} = \int_k^m \frac{x}{\mu_1} f_{S_d}(x)dx = \tfrac{a(mk^a - km^a)}{m^a \mu_1 (1-a)} + m(k/m)^a / \mu_1 \quad (16)$$

$$E\{X_1^2\} = \int_k^m \left(\frac{x}{\mu_1}\right)^2 f_{S_d}(x)dx = \tfrac{a(m^2k^a - k^2m^a)}{m^a \mu_1^2 (2-a)} + m^2(k/m)^a / \mu_1^2 \quad (17)$$

Let W_1 denote the mean total delay at the first queue (queuing and transmission components); W_1 is given by the well known Pollaczek-Khinchin (P-K) formula, i.e.,

$$W_1 = E\{X_1\} + \frac{\lambda E\{X_1^{\,2}\}}{2(1-\lambda E\{X_1\})} \quad (18)$$

The service rate μ_1 of the security processor queue depends on: (i) the speed of the processor (C_p) in instructions per second; (ii) the block-size N_X used by the employed cryptographic algorithm X for encrypting user data (iii) the number of instructions required by the cryptographic algorithm X for encrypting one block of user data of size N_X (in the previous section this quantity has been denoted T_X). The exact relationship giving μ_1 is μ_1= (N_X/8) (C_p/T_X). An important observation with regard to the expression of E{X_1} and E{X_1^2} is that a user packet of size *x* requires the processing of $\lceil x/N_b \rceil$ blocks of data under a block-cipher with block size N_b. In order to simplify the derivation, we have neglected the ceiling function and, thus, the analytic model may account for up to minus one blocks per user packet. As will be shown later, this approximation has a rather minor effect on the accuracy of the model and, thus, is worth performing it in order to simplify the analysis.

6.3.2 Second queue (transmitter)

The arrival process of the transmitter queue is given by the output process of the IPsec processor queue and, thus, is no longer a Poisson process. We will employ an *independence approximation* and assume it to be Poisson nevertheless. The basis for making this assumption is that under heavy load conditions and highly variable service times at the first queue, the independence approximation can produce usable results in terms of accuracy[1]. The aforementioned conditions hold to a large extent true for our application and, thus, as will be shown in the sequel, the numerical results from our approximate analytic model compare favourably to the simulations results of the actual system. This makes the approximate analytic model a useful tool for conducting a first qualitative analysis of a mobile device without having to resort to laborious simulations.

[1] Such approximations are known to be leading to a smaller average delay than the actual system of two queues in tandem and this is also the case in our results here **Error! Reference source not found.**.

More security schemes (possibly future ones) and different parameter sets can thus be evaluated quickly without needing a simulation study.

Under the above-mentioned approximation, the second queue becomes too an M/G/1 queue with the same Poisson arrival process and i.i.d. service times X_2 that correspond to the time that is required for transmitting the IPsec-protected user packets over the wireless link. To write $E\{X_2\}$ and $E\{X_2^2\}$ we will take into consideration the following facts: (i) the wireless channel has a constant transmission rate of μ_2 bytes per second and (ii) the user data packets have sizes that correspond to a truncated Pareto distribution. For the second queue, however, the truncated Pareto distribution will be a shifted version of the original one (the shifting being on the x-axis), because each transmitted packet has an additional space overhead of R bytes due to the encryption related information inserted by the employed security scheme. Thus,

$$E\{X_2\} = \int_k^m \frac{x+R}{\mu_2} f_{S_d}(x)dx = \frac{k^a(ma - R + aR) - m^a(ka + R - aR)}{m^a \mu_2 (1-a)} + \frac{(m+R)(k/m)^a}{\mu_2} \tag{19}$$

$$E\{X_2{}^2\} = \int_k^m \left(\frac{x+R}{\mu_2}\right)^2 f_{S_d}(x)dx = \frac{A+B+C}{m^a \mu_2{}^2 (1-a)(a-2)} + \frac{(m+R)^2 (k/m)^a}{\mu_2{}^2} \tag{20}$$

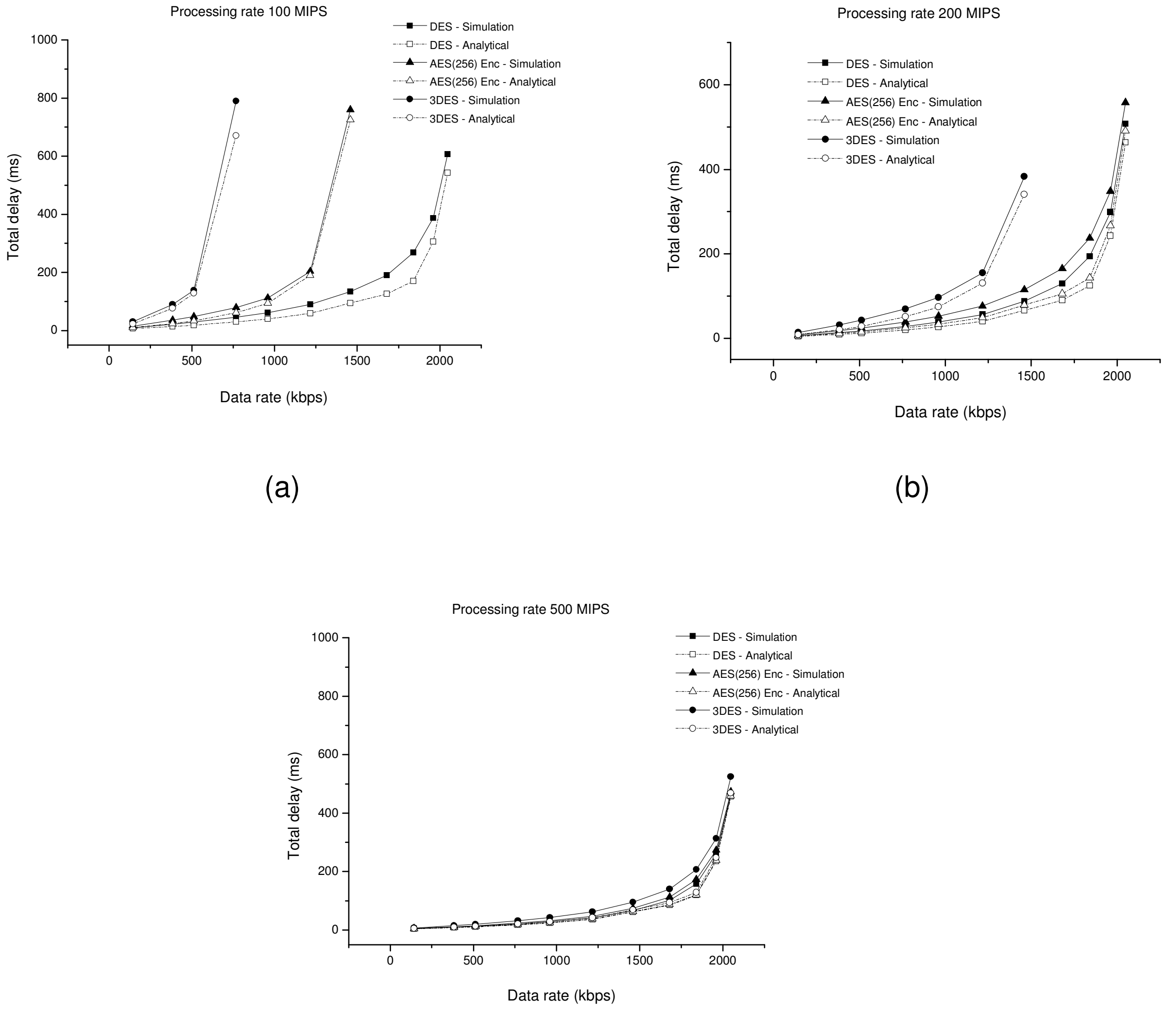
The parameters A, B and C used in Eq. (20) are as follows:

$$A = -3ak^a R^2 + k^a R^2 a^2 + 2k^a R^2 - m^2 k^a a + k^a a^2 m^2 + 2k^a R a^2 m$$
$$B = -4mk^a Ra + 3m^a R^2 a - m^a R^2 a^2 - 2m^a R^2 + m^a ak^2 - m^a a^2 k^2$$
$$C = -2m^a Ra^2 k + 4m^a Rak$$

To produce numerical results from the analytic model, we will use the analysis of the space overhead that appears in Sect. 3 in order to identify the appropriate values for R. To simplify our model, however, we will let R capture only the fixed part of the space overhead; this will have the effect of considering a slightly smaller space overhead than the actual..

The delay at the second queue, W_2, is then easily obtained by the P-K formula. The overall delay (encryption and transmission components) is then taken from $W=W_1+W_2$.

In Fig. 6 we plot the total delay obtained from the above analysis against the one obtained from the simulation experiments of the previous section. We show results for 100, 200 and 500 MIPS for some indicative scenarios such as DES, AES(256)Enc and 3DES. One may easily conclude that the analytic results capture satisfactorily the qualitative behavior in terms of the delay. Observe, however, that the absolute delay values are slightly lower in the analysis than in the simulation. This is in accordance to our expectation and owes to (i) the use of the independence approximation for the arrival process of the second queue; (ii) the disregard of the ceiling function in the computation of the number of encryption blocks that correspond to one user packet of size S_d, and (iii) the disregard of the ceiling function in the computation of the space overhead that is added to each protected packet.



(a)

(b)

(c)

Fig. 6: Total delay obtained from the analytical and the simulation model as a function of the actual data rate for DES, AES(256) Enc and 3DES security scenarios and for (a) 100 MIPS (b) 200 MIPS and (c) 500 MIPS processing rate at the mobile device.

7 Integration of Security in the ACE model

The integration of security into the CASCADAS framework consists in deploying ACEs that implement specific functionalities to protect the system for external attacks. As it is defined in the Security Architecture [28], ACEs of type B have the key role of providing “hard” security services. This consists in embedding cryptographic algorithms and cryptographic functionalities into the so called *Repository Functionality* of the ACE [27]. Thus, the services provided by ACEs of type B are invoked only when security needs to be in place. To give a broader view of the concept of security, first this deliverable briefly describes the ACE conceptual model and then an example is presented to show the integration of a security functionality in a ACE.

7.1 ACE conceptual model

As defined in [27] complex service functionalities are created by ACEs which collaborate and exploit their own capabilities. In this sense, two possible compositions of simple service functionalities are possible:

1. ACEs collaborate loosely to provide a service, while every single ACE remains visible as an independent entity (external model).
2. ACEs aggregate to form a new ACE by providing a service through mutual collaboration (internal model).

The specific functionalities of an ACE are described in the Specific Part which consists of the Functionality Repository and the Self-Model. In the first case, ACEs communicate their capabilities and exploit externally their functionalities to provide a complex service. The second case is more complex as it requires the formation of the so called composed ACE which groups all specific functionalities of the aggregated ACEs in the Functionality Repository.

The communication of ACEs is defined by means of the Goal Achievable (GA) and Goal Needed (GN) protocol which consists in ACEs advertising the capabilities they can provide in order to form complex services. More details on the ACE conceptual model can be found in [27] and in the upcoming prototype implementation provided by Work Package 1.

7.2 Security ACE

The deployment of a security ACE follows the general model and structure defined for ACE. This ensures easy development and, at the same time, it guarantees porting capabilities of security in a common ACE structure. The reason behind this choice is that security is not required in all the communications ACEs are involved and security can be either "aggregated" or simply "exploited" when requested. Moreover, this enables the use of different cryptographic primitives in different contexts which are given by the communication settings, device capabilities and content of the communication. As we have discussed in Section 6, not all basic ciphering algorithms are suitable for the communication with mobile devices, which are characterized by limited processing resources. Thus, it is important for an ACE to choose the best option in terms of cryptographic algorithm available.

Other design choices have been considered for the deployment of security functionalities, e.g. the introduction of specific "hard" cryptographic mechanisms in the Gateway, component that handles the communication of the ACE. This choice would have guaranteed the easy-use of cryptographic functions at the cost of non-flexible control over the algorithm for ciphering/deciphering, signing and hashing information. The *static* choice of including the above mentioned capabilities into the Gateway implies that the replacement of the algorithm or the function could only be achieved with the creation of multiple instances of the same service ACE, which implements different cryptographic primitives. Moreover, the communication context may not require protection for the service itself or messages and security would increase computational complexity unnecessarily.

Herein, for the sake of clarity, we describe security services without distinction between the internal and the external model of the ACE as this will simplify the discussion of the solution concept, but it will not limit the description of the applicability of the security functionality. Moreover, we assume the co-existence of multiple ACEs in one trusted domain, like a laptop. These ACEs aggregate or exploit each other capabilities to form complex services on demand. Services can be formed also across trusted domains but one secure aggregate component should exist in each trust domain to ensure security is in place if required. However, tiny devices may represent an exception to this model as they can embed few ACEs, none of which implements security due to hardware constrain capabilities. The concept of trusted domain will be clarified in Section 8 where we discuss security in the application scenario designed to demonstrate CASCADAS features.

7.2.1 Communication Model

The Communication Model for security ACEs follows the Goal Achievable (GA) and Goal Needed (GN) message exchange protocol defined in [27]. ACEs discovery other ACEs by issuing a Goal Achievable message that states which services it can provide to another ACE. This ACE can match the GA with the specific functionalities declared in the Goal Needed messages to complete its function. As discussed in the CASCADAS Security Architecture document [28], we have maintained low level granularity for the definition of the capabilities of the "security" ACE: each simple component has one possible operation or service implemented in the functionality repository.

Main task of a "security" ACE is to provide security services to other ACEs. The message that is used by an ACE to state what kind of job it is able to provide is specified in the Goal Achievable: It has a semantic description of the job. For the purpose of security, the description of the service is seen within a general framework that includes a set of several ACEs and more detailed description that characterize the ACE itself. For instance, the Goal Achievable for an ACE might be a symmetric cipher function (general description) implementing DES (characterizing function).

As discussed above, the execution of a security service may require functionalities that are not included in the ACE. The Goal Needed is a sort of request, with a semantic description attached, which specifies what kind of functionalities the ACE needs from other ACEs, to achieve its goals. This implies that any ACE should be able, given a GA, to semantically match it with its Goal Needed (GN) in order to properly answer to the received GA message.

As defined above, the specification of the service is semantically described and it is part of the Goal Achievable. However, the same functionality defined in the repository can accomplish different tasks if combined with other ACEs. For instance, a symmetric key algorithm like TripleDes can use as input a key of 168 or of 112 bytes or better a HMAC can use different hash functions, e.g. SHA-1 and MD5, to provide data integrity and data authentication. A complex example can be given by an ACE 1 which can provide data authentication and integrity, but to authenticate the data it needs a shared key between the source (ACE 1) and destination (ACE 2); for instance, this shared key can be derived with the Diffie-Hellmann key agreement protocol (see Section 4.1) specified in other ACEs, see Fig. 7. In this last example the Goal Needed of ACE 1 and 2 will specify Diffie-Hellman and it will match the Goal Achievable of ACE 3 and ACE 4.

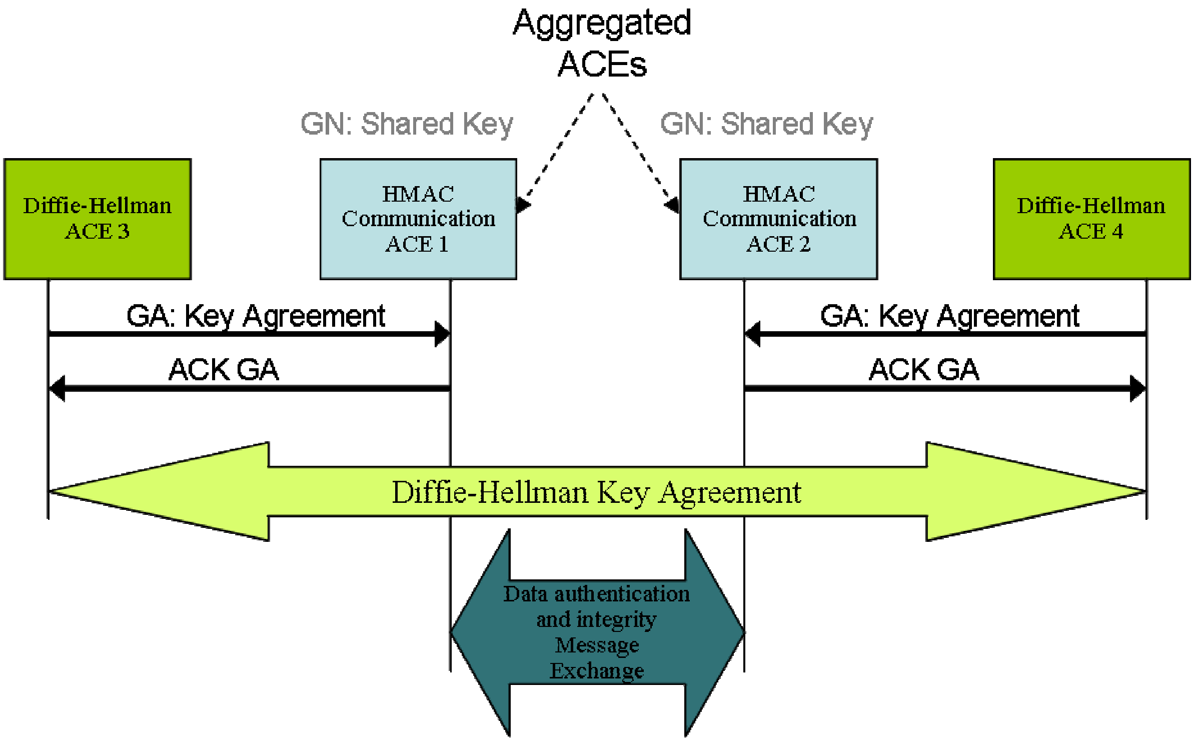


Fig. 7: ACE Communication Model: ACE 1 and ACE 2 want to communicate securely. They exploit the capabilities of ACE 3 and 4 to agree on a symmetric key used to provide data authentication and integrity during the message exchange.

This formation of complex services enables the malfunctioning components to be replaced easily and different algorithms or functions can be aggregate or exploited in accordance to the context of communication. For instance, in the example depicted in Fig. 7 ACEs 1 and 2 can use either SHA-1 or MD5 as hash functions by considering what is available in their trusted domain.

7.2.2 Functionality Repository

The Functionality Repository enables specific functionalities to get deployed into the ACE instance and get accessed via ACE events model studied in Work Package 1. It has two distinct roles as storage for the functionality and the underlying classes, and as internal element of ACE as it has an interface for the communication with other *organs*[2] inside the ACE.

The deployment of the security functionality follows the framework defined in Work Package 1. The library that includes the classes of the functionality is stored in the *Repository Functionality* as well as a configuration file formatted in XML that describes the functionality implemented in the library. The XML file represents the internal view of the functionality for other components. Each functionality is identified by a unique name inserted in the XML descriptor as well as the input parameters and the output. The output consists of the results obtained from the execution of the functionality and it states what events should be created in response.

[2] We refer to the terminology used for the ACE model. Organs are the internal components of the ACE.

7.3 Example of Security ACE

This section discusses the porting of a functionality implementing security into the ACE. Herein, we focus on the definition of the XML descriptor for the security functionality which follows the guidelines for the deployment of ACEs. We also discuss the Self-Model XML description for the same security functionality which defines what are the states of the service and more important contains the specification of the Goal Achievable and Goal Needed. Specific cases can be derived from this example easily. Then, we discuss how ACEs are used in the scenario shown in Fig. 7.

```
1 <functionality id="HMACDigest">
2         <black-box-description>
3                 <input>
4                         <param name="algorithm" type="java.lang.String"/>
5                         <param name="provider" type="java.lang.String"/>
6                         <param name="key" type="javax.crypto.SecretKey"/>
7                         <param name="message" type="byte[]"/>
8                 </input>
9                 <output name="digest" type="byte[]"/>
10         </black-box-description>
11         <advanced-call-details>
12                 <call class-name="cascadas.security.HMACDigest" method-name="digestHMAC">
13                         <arg ref="message"/>
14                 </call>
15                 <functionality-return ref="digest"/>
16         </advanced-call-details>
17 </functionality>
```

Fig. 8: XML description of the ACE that implements HMAC

Fig. 8 shows the code that describes the functionalities of a ACE capable of calculating the HMAC value of a message given a shared key as input. The functionality is identified by the unique id (line 01) and the input parameters and output are specified in the *black-box-description.* In this specific example, the ACE requires as input the specific algorithm (e.g. HMAC-MD5 or HMAC-SHA1), the provider that implements the cryptographic functions and the shared key.

```
<selfModel>
        <plan id="Plan1" default="true">
                <states>
        <!-- All possible states of the Plan1 are defined here -->
        <state id="state1">
            <friendlyName>Ready</friendlyName>
            <confidence>1</confidence>
        </state>
        <state id="state2">
            <friendlyName>HMACDigest</friendlyName>
            <confidence>1</confidence>
        </state>
    </states>
    <transitions>
        <!-- All possible transitions of the Plan1 are defined here -->
        <transition id="tr1">
            <source>state1</source>
            <destination>state2</destination>
            <condition></condition>
            <guard_condition></guard_condition>
            <action>HMACDigest(algorithm=HmacSHA1_provider=BC_message=message)</action>
            <goal_needed>
                <Assert>
                    <And>
                        <Atom closure="universal">
                            <Rel>GN</Rel>
                            <slot>
                                <Ind>SharedKey</Ind>
                                <Var>?key</Var>
                            </slot>
                        </Atom>
                    </And>
                </Assert>
            </goal_needed>
            <goal_achieved>SHA1_HMACDigest</goal_achieved>
        </transition>
    </transitions>
    <!-- ======== PLAN CREATION RULES ========== -->
……………
```

```
            <!-- ======== PLAN MODIFICATION RULES ========== -->
................
        </plan>
</selfModel>
```

Fig. 9: Self-Model description of the ACE that implements HMAC

The Goal Achievable and the Goal Needed for this functionality are specified in Fig. 9. In this case the Goal Achievable is simply the possibility to generate message authentication and integrity while the Goal Needed is the creation of a shared key between two entities that can be used to authenticate the data. This key can be generated by using the capabilities of another ACE described by the XML code in Fig. 10. This ACE is capable of generating a key giving as input the algorithm and the java provider used for the implementation.

```
<functionality id="GeneretorKeys">
    <black-box-description>
        <input>
            <param name="algorithm" type="java.lang.String"/>
            <param name="provider" type="java.lang.String"/>
        </input>
        <output name="key" type="javax.crypto.SecretKey"/>
    </black-box-description>
                <simple-call-details class-name="cascadas.security.GeneretorKeys" method-name="generation"/>
    <functionality-return ref="key"/>
    <output-event-mappings>
        <mapping event="cascadas.ace.event.ServiceResponseEvent">
            <value ref="key"/>
        </mapping>
    </output-event-mappings>
</functionality>
```

Fig. 10: XML description of the ACE with Key Generation functionalities

The ACE capable of generating Keys does not have any Goal Needed specified as shown in Fig. 10, but its Goal Achievable matches the Goal Needed of the ACE implementing HMAC.

```
<selfModel>
```

```
    <plan id="Plan1" default="true">
        <states>
            <!-- All possible states of the Plan1 are defined here -->
            <state id="state1">
                <friendlyName>Ready</friendlyName>
                <confidence>1</confidence>
            </state>
            <state id="state2">
                <friendlyName>GeneratorKeys</friendlyName>
                <confidence>1</confidence>
            </state>
        </states>
        <transitions>
            <!-- All possible transitions of the Plan1 are defined here -->
            <transition id="tr1">
                <source>state1</source>
                <destination>state2</destination>
                <condition></condition>
                <guard_condition></guard_condition>
                <action>GeneratorKeys(algorithm=HmacSHA1_provider=BC)</action>
                <goal_needed></goal_needed>
                <goal_achieved>Key</goal_achieved>
            </transition>
        </transitions>
        <!-- ======== PLAN CREATION RULES ========== -->
………….
    </plan>
</selfModel>
```

Fig. 11: Self Model description of the ACE that implements a key generation functionality

In Fig. 7 we have presented an example of two ACEs that want to exchange data in such a way that the data are authenticated and modification during the transmission can be detected. The ACE specified in Fig. 8 is used to ensure data authentication and integrity. The input shared key for the communication can be generated by exploiting the functionalities of the ACE described in Fig. 10. However, more complex protocols can be used to generate keys as depicted in Fig. 12, where it is described the ACE that implements Diffie-Hellman key agreement protocol.

```
<functionality id="KeyAgreementDH">
        <black-box-description>
                <input>
                        <param name="algorithm" type="java.lang.String"/>
                        <param name="provider" type="java.lang.String"/>
                        <param name="parameters" type="java.lang.String"/>
                        <param name="fileName" type="java.lang.String"/>
                        <param name="sharedKey" type="javax.crypto.SecretKey"/>
                        <param name="publicKey" type="java.security.PublicKey"/>
                        <param name="privateKey" type="java.security.PrivateKey"/>
                        <param name="keyPair" type="java.security.KeyPair"/>
                </input>
                <output name="sharedkey" type="javax.crypto.SecretKey"/>
        </black-box-description>
        <advanced-call-details>
                                        <call class-name="cascadas.security.KeyAgreementDH" method-name="genDhParams">
                        <return ref="parameters"/>
                </call>
                                        <call class-name="cascadas.security.KeyAgreementDH" method-name="writeParameters">
                        <arg ref="parameters"/>
                        <arg ref="fileName"/>
                </call>
                                        <call class-name="cascadas.security.KeyAgreementDH" method-name="genDHKeys">
                        <arg ref="parameters"/>
                        <return ref="keyPair"/>
                </call>
                                        <call class-name="cascadas.security.KeyAgreementDH" method-name="writeKeyToFile">
                        <arg ref="sharedKey"/>
                        <arg ref="fileName"/>
                </call>
                <call class-name="cascadas.security.KeyAgreementDH" method-name="agreement">
                        <arg ref="privateKey"/>
                        <arg ref="publicKey"/>
                <return ref="sharedKey"/>
            </call>
        </advanced-call-details>
        <output-event-mappings>
            <mapping event="cascadas.ace.event.ServiceResponseEvent">
                <value ref="sharedKey"/>
            </mapping>
        </output-event-mappings>
</functionality>
```

Fig. 12: XML description of the ACE that implements Diffie-Hellman

8 CASCADAS Autonomic toolkit and application case scenario

This section summarizes the contribution of WP4 to the CASCADAS autonomic toolkit with particular emphasis to the application scenario. As presented in the previous section, the role of WP4 is focused on the delivery of libraries that implement basic cryptographic functions to secure the CASACADAS system. The first version of the Open Secure CASCADAS toolkit consists in a practical selection of cryptographic algorithms and hash functions with respect to the heterogeneous nature of the devices. This will help to understand what the performances of these algorithms and functions are with respect to key size, device capabilities and cryptosystem type.

As shown in Section 7.3, security functionalities can be defined in line with the definition of the ACEs provided by WP1. The work targets the Specific Part (Repository Functionality and Self-Model) of the ACE structure. Security functionalities will be part of the "Specific Interface" which contains security functionalities, implemented in the "Specific Feature" by means of cryptographic libraries, which characterize the ACE behaviour, as described in Section 0. For the sake of clarity we have presented a semantic description of the job the ACE is able to do (GA) and the indispensable and essential actions and conditions to accomplish it (GN), as defined in Section 0.

The first release of the Open Secure Toolkit is centred on ACEs of type B, as they provide specific cryptographic protection of the CASCADAS system. The application of the security functionalities to the application testbed could be done on demand. We foresee security services as network and application services that can be aggregated and used when needed. The definition and the implementation of the security components are compliant with the ACEs structure.

8.1 Auction-pervasive advertisement scenario

In the context of the auction-based pervasive scenario, security has a key role to ensure the protection of the information (bids) and to control the system as a whole. ACEs need to interact in a secure way to place bids and to communicate so that no intruders can impersonate bidders. We propose to exploit the security features implemented in specific cryptographic ACEs, identified as ACEs of type B in the security architecture [28], to provide security services to nodes if requested. Herein, we suggest security features in the context of the proposed scenario. These concepts might be useful as guidelines for the developers to implement security features in the application.

To comply with the requirement of light ACEs, simple functionalities for each ACE, we envision the formation of virtual security domains constituted by ACEs under the same administrative control that have different capabilities and form the aggregated component. This new aggregate component will participate in the auction as bidder.

An example of virtual security domain is a user's laptop running several ACEs with different functions which aggregate to implement the required complex functionality: one ACE that is capable of participating in the bidding process, one ACE that has symmetric key encryption functionalities, one ACE that implements a Diffie-Hellman key establishment protocol, one ACE that functions as PRGN, one ACE that is capable of computing HMAC, and so on. The ACEs can communicate among each other without any encryption as they belong to the same pre-trusted domain and the message exchange (GA-GN protocol) is local.

With this setting in mind, we can envision that the seller will exploit the capabilities of the ACE implementing signatures to digitally sign the advertisement message that is sent to the Auction Center. The Auction Center will use a hash function-capable ACE to send the List of Actual items to enforce the integrity of the list.

Due to the time-critical settings in the auction-pervasive scenario, we cannot rely on time-demanding computation algorithms for the bidders when they bid. The auctioneer will eventually inform bidders about the current price, thus, the bid message might not be encrypted, but the content of the message needs to be protected from forgery and to be authenticated. A solution to guarantee message integrity and authentication is to use HMAC for data authentication and data integrity. However, this requires the bidder to share a symmetric key with the auctioneer: this key will be generated by the bidder itself (ACE PRGN), encrypted with the public key of the auctioneer (disseminated by the Auction Center with the list of items) and signed by the bidder. This will ensure that only the auctioneer can read the key and this key can be used for subsequent messages, because if a bidder is interested in an item he is likely to participate actively in the auction.

Finally, the seller will inform the bidders when the auction ends by signing the proper message. In case of repudiation of a bid, the Auction Center will behave as third party as the seller will send the highest bid to the Auction Center with the message digitally signed by the winning bidder.

9 Cryptographic libraries

In this section, we provide some details of implementing above concepts in practice. This documentation is based on the *Java Cryptography Architecture* [26] which specifies how to develop cryptographic functionalities for the Java Platform. Sun provides a Java Cryptography Extension for the implementation of encryption, key generation, key agreement and Message Authentication Code algorithms. The name of the Sun provider is "SunJCE" which is already specified in the security functionalities of Java during the installation and it the implemented classes are contained in the *javax.crypto* package. The Java Cryptography Architecture has a built-in support to extend the cryptographic implementation by using functionalities developed by other providers, which we exploit to use the cryptographic algorithms implemented by Bouncy Castle [25]. In particular, this support consists in guaranteeing implementation interoperability among different providers, like the use of keys or verification of signatures and so on.

For the CASCADAS autonomic toolkit, we have considered both the Sun Java Cryptographic extension (SunJCE provider) and the Bouncy Castle Crypto (BC provider) as they provide the implementation of most of the cryptographic algorithms available and an extensible documentation for the classes and methods. As introduced above, the interoperability of providers enables the specification of cryptographic providers either in during the development of the application or statically in the java security policy. The first solution has been adopted to develop sample security features, implemented by the Bouncy Castle provider, which are available in the CASCADAS repository at trunk\WP4\Security-examples\

Hereafter, we present some examples based on Java *crypto* class and on *Bouncy Castle crypto class* to generate Keys and Certificates in Java and how to use them. We will be mainly referring to sources of information [20][23][24][25].

Basic steps in sending information securely are as follows:

- Generate public-key, private-key pair and obtain a digital certificate.
- Agree upon a symmetric encryption scheme (e.g. AES), hash scheme (e.g. MD5) and signature scheme (e.g RSA).
- Generate a shared key (by using DH key exchange protocol) and use that to encrypt the information (a digital document).
- Calculate the hash of the information.
- Sign the hash with private key.
- Send encrypted text, along with the signed copy of it.

The task of generating the key-pair and Digital Certificates DC are related to PKI, and java `keytool` tool can be used to perform these steps [24]. The `keytool` tool can be used to:

- Create private keys and their associated public key certificates: `keytool -genkey` command.
- Issue certificate requests, which you send to the appropriate certification authority: `keytool -certreq` command.
- Import certificate replies, obtained from the certification authority you contacted `keytool -import` command.
- Import public key certificates belonging to other parties as trusted certificates `keytool -import` command.
- Manage your keystore.

Java's JCE and BouncyCastle crypto provide a class `KeyAgreement` to implement the DH protocol [20][25]. The following steps are involved:

- The method `KeyGenerator` can be used to generate public keys to be used for shared key. `KeyAgreement` objects are created using the `getInstance` factory methods of the KeyAgreement class. `getInstance` takes as its argument the name of a key agreement algorithm. `public static KeyAgreement getInstance(String algorithm);`

- To initialize a `KeyAgreement` object, we use one of its `init` methods. For DH algorithm, we also pass prime modulus p and a base generator g as its parameters: `public void init(Key key, AlgorithmParameterSpecparams);`
- In the next phase, we call the `doPhase` method:
  `public Key doPhase(Key key, boolean lastPhase);`
  The `key` parameter contains the key to be processed by that phase. In most cases, this is the public key of one of the other parties involved in the key agreement. The `lastPhase` parameter specifies whether or not the phase to be executed is the last one in the key agreement: A value of `FALSE` indicates that this is not the last phase of the key agreement (there are more phases to follow), and a value of `TRUE` indicates that this is the last phase of the key agreement and the key agreement is completed. In the example of Diffie-Hellman between two parties, we call `doPhase` once, with `lastPhase` set to TRUE. In the example of Diffie-Hellman between three parties, we call `doPhase` twice: the first time with `lastPhase` set to `FALSE`, the 2nd time with `lastPhase` set to `TRUE`.
- We compute the shared secret by calling one of the `generateSecret` methods:
  `public byte[] generateSecret();`
  `public SecretKey generateSecret(String algorithm);`

We can use Java Package `javax.crypto.interfaces` to generate various intermediate parameters, e.g. prime p, generator g by calling its interfaces [23]:

- DHKey: This interface marks public/private keys in the Diffie-Hellman key exchange algorithm.
- DHPrivateKey: This interface marks a private key in the Diffie-Hellman key exchange algorithm.
- DHPublicKey: This interface marks a public key in the Diffie-Hellman key-exchange algorithm.

An implementation of Diffie-Hellman is given in the CASCADAS repository as reference to show the easy use of security features.

10 Conclusion

In this deliverable we presented the basic security functions and services that can be readily integrated in the CASCADAS software framework. The goal of this deliverable is to provide practical, hands-on experience on the use of software libraries (compatible with the development environment selected for the project) that offer basic security services to applications and communication protocols. We purposely flavored the Deliverable to lean towards engineering problems when deploying secure applications: hence, this deliverable does not represent the research effort that the Partners involved in Work-Package 4 are carrying on.

This first release of the software toolkit for security also includes a detailed analysis of performance issues that are of paramount importance: for every security service or cryptographic function we provided experimental proof of its performance and we greatly discussed on best-practices when enhancing an existing application or protocol with security mechanisms.

We enriched this deliverable with a case study that focus on resource-constrained environment in which the autonomic services deployed using the CASCADAS architecture would require strong ent-to-end security. For this purpose, we present an analytical and simulation-based analysis of the IPSec framework, when used to secure communications between mobile devices. This case study is well suited for the application scenario that we target in the project.

Finally we discussed how to integrate an open-source cryptographic library in the ACE model of CASCADAS, with an example on how to establish a secure communication channel. Moreover, we pinned down how security services can be integrated in the application scenarios devised for the demonstration of CASCADAS activities.