Hochschule Offenburg
offenburg.university

ivESK Institut für verlässliche Embedded Systems und Kommunikationselektronik

E+I Elektrotechnik und Informationstechnik

Extension and Integration of an Abstract Interface to Cryptography Providers

Bachelor Thesis

Steve Wagner
EI-3nat

February 29, 2016

Prof. Dr. Axel Sikora
Dipl.-Phys. Andreas Walz
Institute of reliable Embedded Systems and Communication Electronics (ivESK)
Offenburg University of Applied Sciences

Statutory declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Offenburg, ______________________ ______________________

Date Signature

Abstract

English

This thesis presents the creation and implementation of a new Generic Cryptographic Interface.

This interface is a base of the main algorithm use in cryptography today. The interface has been implemented in a project developed in the Institute of Reliable Embedded Systems and Communications Electronics (ivESK) in the Offenburg University of Applied Sciences and is named emb::TLS.

This project use a open-source cryptographic software library, named LibTomCrypt, which is used for the calculation of each cryptographic algorithm.

The Generic Cryptographic Interface has been implemented to be used for the application, emb::TLS, and to use the cryptographic provider, LibTomCrypt.

This thesis presents this Generic Cryptographic Interface, with the main cryptographic algorithm that can be used today, and shows an example of an implementation in a TLS protocol project with an open-source cryptographic software library.

German

Diese Arbeit stellt die Herstellung und Implementierung von einem generischen kryptographischen Interface.

Dieses Interface ist ein Basis von die grundsätzlichen kryptographischen Algorithmen, die heute sehr viel benutzt ist.

Dieses Interface wird in ein Projekt, emb::TLS, implementiert, welche in der Institut für verlässliche Embedded Systems und Kommukationselektronik hergestellt ist.

Dieses Projekt benutzt eine open-source kryptographische Software library, LibTomCrypt, welche für die Berechnung für jeden Algorithmen benutzt ist.

Diese generisches kryptographisches Interface wird auf diese Projekt implementiert. Der emb::TLS Projekt wird diese Interface benutzen, welche der LibTomCrypt Provider benutzen wird.

Diese Arbeit stellt die Schritte für die Herstellung des Interfaces und die Implementierung des Interfaces in dem emb::TLS Projekt.

Acknowledgement

I would like to express my appreciation to Prof. Dr.-Ing Sikora for the supervision of this project, and the providing of constructive suggestions and critiques during the work.

I would also thank Dipl.-Ing. Walz for the assistance during the project.

Finally, I would like to thank my parents and friends for their support and encouragement during my studies.

Contents

List of Figures

List of Tables

1 Background

Nowadays, the security of data is very important and widely use into several domains, for example in Web browsers, e-mail, cell phones, bank cards, cars and medical implants. What is to understand under security is that the data remain private (confidentiality), meaning that no one, except whom is intended the data can understand the transmitted data, the data could not be modified during the transmission (integrity), and the data is authenticated (authentication), meaning that the sender can at any time be authenticated by the receiver. Several cryptographic algorithms are used today to obtain these specifications and to have the most secured communication as possible.

The TLS protocol, which is today widely deployed as security protocol for all kinds of Web-based applications for e-commerce and e-business for example, is part of most security systems available today.

This protocol can use several different cryptographic algorithms to secure a communication between a Client and a Server. The different use of these algorithms are described into cipher suites, to use the different function which can be provided by the TLS protocol.

Embedded systems, which are today more and more used because of this small size, is a example of systems which TLS protocol must be optimized. The optimization of this protocol is focus on the reduce of the calculation done for each cryptographic algorithm, because most of the embedded systems are powered by a battery, also with low autonomy, and embedded systems are slower than ordinary computers.

2 Motivation

In the institute of reliable Embedded Systems and Communication Electronics (ivESK) is a project named emb::TLS which has the goal to use the TLS protocol (see chapter 4.1) in embedded systems.

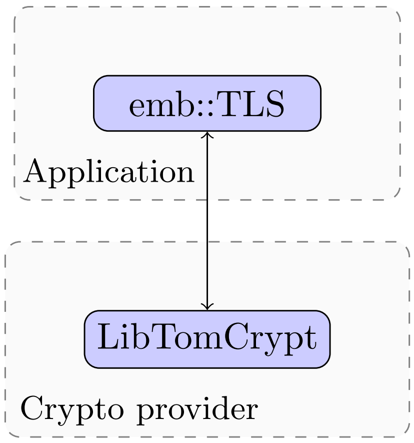


Figure 2.1: emb::TLS's project

For this, a cryptographic provider (LibTomCrypt) is used for the part of cryptographic calculation needed for the application. This cryptographic provider is an open-source cryptography software library. Problems with this implementation is that only LibTomCrypt is supported as cryptographic providers, meaning that no other libraries can be used without changing the complete implementation for emb::TLS.

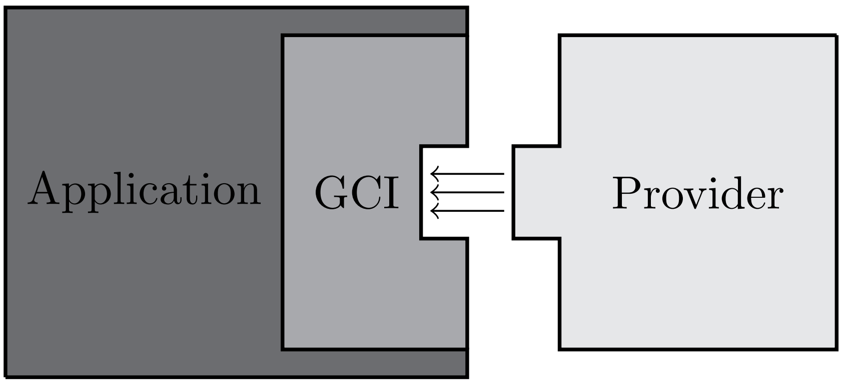


Figure 2.2: Goal of the new implementation

The goal of the project is therefore to create an interface, a Generic Cryptographic Interface (GCI), to have a base of the existing cryptography services and to have the possibility to easily change the providers only by changing some lines in the interface, instead of the complete implementation.
Through to this new interface other new cryptographic algorithms may be easier to add in the interface and to use for the application.
As shown on the figure 2.2, the interface is implemented in the application and nothing has to be changed. The requirements for this Generic Cryptographic Interface (GCI) are listed below:

1. No hidden states shall be used in the interface, meaning that the behavior of the cryptographic algorithm functions should only be affected by the input parameters. All parameters written in input of the function will be used for the cryptographic algorithm and nothing else.
2. Different cryptographic providers may be used for the cryptographic calculation.
   That could be open-source cryptographic software libraries or hardware-based cryptographic modules.
3. With the old version of the implementation (figure 2.1), the application and the provider interact, meaning that when the provider needs informations from the application, this one just send them and vice-versa. With this new implementation (figure 2.2) the interface break this interaction. To always have this interaction, the interface shall receive the information from the application and the provider. These informations shall be easily used by both (application and provider) when needed.

3 Introduction

3.1 Cryptography

Cryptography is a mathematical science for encryption and decryption of data. The meaning of the data should therefore be hidden to anyone who must not understand it, but should become understandable to whom the data is sent to.
Cryptanalysis is the complementary of the cryptography with the focus on the defeat of the cryptographic mathematical techniques.
Both terminology is named cryptology.

The most important security of the information is defined into:

- Confidentiality or privacy
  No one, except whom is intended, can understand the transmitted data
- Integrity
  No one can alter the transmitted message without the alteration is being detected
- Authentication
  The sender and the receiver can identify the destination of the data and identify themselves
- Non-repudiation
  The sender cannot deny at a later stage the transmission of the datas

The cryptographic mathematical techniques are grouped into several algorithms:

- Hash algorithm
- Signature algorithm
- Symmetric cipher algorithm
- Asymmetric cipher algorithm

3.1.1 Hash algorithm

Hash functions are widely used in several other cryptographic algorithms, as signature or cipher. A digest, which is a short, fixed-length bit-string, is the result of a computed message over a hash function. This digest is considered practically impossible to invert, meaning that impossible to recreate the input data with it. The main properties of a hash function are:

- it's quick to compute the digest for any message
- it's infeasible to generate a message from its digest
- it is infeasible to modify a message without changing the digest
- it is infeasible to find two different messages with the same digest.

Several hash algorithms are used today, with different properties shown on table 3.1.

Algorithm	Output (bit)	Input (bit)	Collisions found
MD5	128	512	yes [11]
SHA1	160	512	not yet
SHA-224	224	512	no
SHA-256	256	512	no
SHA-384	384	1024	no
SHA-512	512	1024	no

Table 3.1: Properties of hash algorithms

3.1.2 Digital signature algorithm

Digital signatures are one of the most important cryptographic algorithms that are widely used today. Digital signatures can be used with public key cryptography (asymmetric key, see section 3.1.4.2), which provides authentication, the receiver can verify the authenticity of the incoming information, and data integrity, the receiver can verify that the informations are intact. One of the most important properties of Digital signature is that it provides non-repudiation, meaning that the sender cannot claim he has not sent the data.

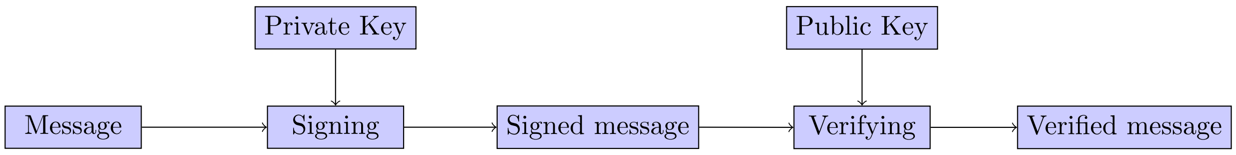


Figure 3.1: Introduction of a signature operation

As shown on the figure 3.1 a message, which could previously be hashed, is signed with the private key, which only one person has it. The signature can then be verified with the public key, which should be generated with the private key used to sign the message.

Three type of Digital signature are available:

- RSA signature scheme, which is the most widely used digital signatures scheme in practice. The Probalistic Signature Standard (PSS) RSA Padding [8] should be used to increase the security of the signature. This padding is a signature scheme based on the RSA cryptosystem (see section 3.2) and combines signature and verification with an encoding of the message (see section 3.1.4.2). It allows to the receiver to distinguish between valid and invalid messages.

- DSA signature scheme [12], which is a variant of the Elgamal signature [15]

- ECDSA signature scheme [12], which is closely related to the DSA scheme, but constructed in the group of an elliptic curve [3].

The security provides by each signature scheme is explained in section 3.2.

3.1.3 Message Authentication Code (MAC)

The Message Authentication Code (MAC) is widely used in practice. It has some properties with digital signatures (see 3.1.2), since it provides message integrity and message authentication, but do not provide non-repudiation, because of the use of symmetric keys. MAC is much faster than Digital signatures since it's based on block ciphers or hash functions.

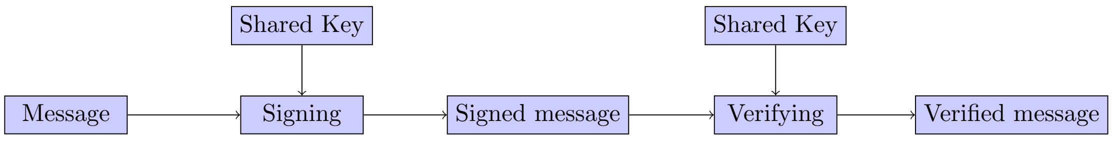


Figure 3.2: Introduction of a MAC operation

As shown on figure 3.2 MAC uses symmetric keys for both generating and verifying a message.

The properties of a MAC are:

- Cryptographic checksum
  Secure authentication tag for a given message (when use with hash functions)
- Symmetric
  Signing and verifying are done with symmetric keys
- Arbitrary message size
  The input can be of different sizes
- Fixed output length
  The signature generated by the MAC has a fixed output length
- Message integrity
  Any modification of the message during the transit will be detected by the receiver
- Message authentication
  The receiver is ensured of the origin of the message

The two mains MAC are:

1. MAC from Hash functions (HMAC) [18], which is a hash-based message authentication code
2. MAC from Block Ciphers (CBC-MAC) [16], which is a block cipher-based message authentication code

3.1.4 Cipher

The cipher is an algorithm uses for encryption of data (plaintext) and decryption of data (ciphertext). It's used on the internet, for the e-mail, on cellphones when calling, etc. Two main cipher algorithms exist, which are:

- Symmetric
- Asymmetric

3.1.4.1 Symmetric cipher algorithm

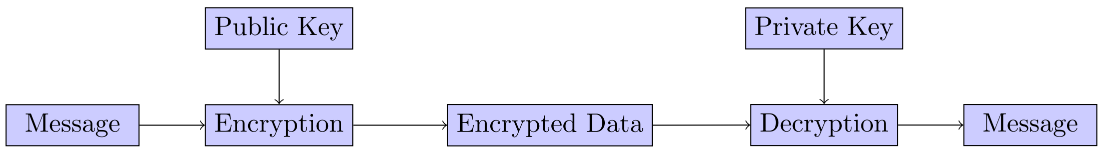


Figure 3.3: Introduction of a symmetric cipher operation

Symmetric-key algorithms are algorithms for cryptography that use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext in a communication. This is shown on figure 3.3. The key is often named shared secret key. There are two kinds of symmetric encryption:

1. Stream ciphers, which encrypts bits individually
   These are very deprecated today, because insecured. The most used of stream cipher is the RC4 [11].
2. Block ciphers, which encrypts an entire block of plaintext bits at a time with the same key
   Three main block cipher are used today:
   - Data Encryption Standard (DES), which is not considered secure against attacker because the DES key is too small [9]
   - Three subsequent of DES (3DES) is a very secure cipher which is still used today
   - Advanced Encryption Standard (AES) is the most widely used symmetric cipher today. It's a standard for the Internet security (IPsec) [5], TLS [4], the secure shell network protocol SSH (Secure Shell) [6] and numerous other security products around the world.

3.1.4.2 Asymmetric cipher algorithm

Asymmetric algorithm is a method for encryption and decryption of messages with public and private keys. This is shown on figure 3.4. One peer generates the private and public key together, keep the private key (meaning that he is the only one to have it) and sends the public key to every one he wants to communicate with. With the public key everyone can encrypt a message. Only the owner of the private key can decrypt this message, meaning that no one can understand the message through the communication and no one can decrypt it.

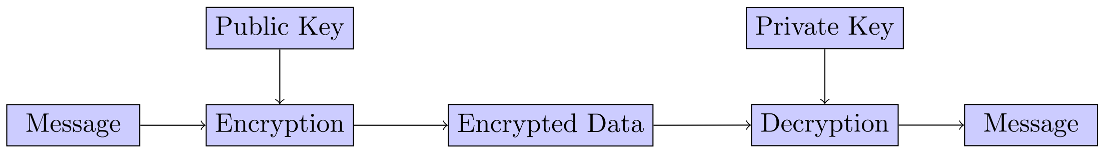


Figure 3.4: Introduction of an asymmetric cipher operation

Through the principle of private and public keys, asymmetric algorithms allow privacy (like symmetric algorithm) but integrity of data too, because only this one who creates the keys has the private key for decryption.

Three mains asymmetric cipher are used today:

- RSA [8] is currently the most widely used asymmetric cryptographic scheme, even thought elliptic curves and discrete logarithms (see below) are gaining ground. The applications, which RSA is in practice the most used are:
  - Encryption of small pieces of data, especially for key transport
  - Digital signatures (see section 3.1.2)
- Diffie-Hellman (DH) is a discrete logarithm (see section 3.2) which is most used in commercial cryptographic protocol like Secure Shell (SSH) [6], TLS [4], and Internet Protocol Security (IPSec) [5]
- Elliptic Curve of Diffie-Hellman (ECDH) is based on the conventional Diffie-Hellman explained above but using elliptic curves [3]

3.2 Security of cryptographic algorithms

All the keys use in cryptography have different sizes and different number of parameters to generate the key. This differences make the security level of each key algorithm. The security level (in bits) of a key algorithm corresponds to the number of operations needed (2^{nbits}) to break the security of it, meaning to find the solution of the operation for the key calculation.

Table 3.2 shows the different security level of the main cryptosystems existing in cryptography. We can see that a 80-bit symmetric key provides roughly the same security as a 1024-bit RSA key.

Algorithm Family	Cryptosystems	Security Level (bit)			
		80	128	192	256
Integer factorization	RSA	1024 bits	3072 bits	7680 bits	15360 bits
Discrete logarithm	DH, DSA, Elgamal	1024 bits	3072 bits	7680 bits	15360 bits
Elliptic curves	ECDH, ECDSA	160 bits	256 bits	384 bits	512 bits
Symmetric-key	AES, 3DES	80 bits	128 bits	192 bits	256 bits

Table 3.2: Table of security level of each key algorithms

Symmetric key

The security level of a symmetric key depends on the block cipher mode operation chosen (see Section 3.1.4.1).

Integer factorization

Integer factorization has the property to be based on large integers. This is used for the RSA algorithm. Compared to a symmetric algorithm, a RSA algorithm is several times slower. This is because of the many computations involved in performing RSA. A RSA is composed of a public exponent, a private exponent and a modulus.

Discrete logarithm

Several algorithms are based on discrete logarithms problems in finite fields. Diffie-Hellman and Digital Signature Algorithm (DSA) are examples of discrete logarithms.
For the Diffie-Hellman a prime and a generator are needed to use this algorithm. The length of the prime should be similar to the modulus of the RSA to provide strong security. The generator should be a primitive element. The prime and the generator represent the domain parameters of the Diffie-Hellman algorithm.
For the Digital Signature Algorithm (DSA) a prime, a divisor and a generator are needed to use this algorithm. Only the length of the prime and the divisor provide different security level of the DSA algorithm. This is shown table 3.3. The prime is represented by p, the divisor by q. The generator represents the subgroups with q elements. The prime, divisor and generator represent the domain parameters of the DSA algorithm.

p	q	Output (bits)	Security levels
1024	160	160	80
2048	224	224	112
3072	256	160	128

Table 3.3: Table of security levels of DSA domain parameters

Elliptic Curves

The Elliptic curves is the newest member of the three families of establishes public-key algorithms. It provides the same level of security as Integer factorizations or Discrete logarithms with shorter operands (see table 3.2). Elliptic curves are based on the generalized discrete logarithm problem, that's why it can be used with Diffie-Hellman or DSA protocols.
Elliptic curve Diffie-Hellman (ECDH) uses the same principle as standard Diffie-Hellman, only the domain parameters change. For a ECDH algorithm an elliptic curve should be used instead of the domain parameters. Several elliptic curves are normalized [3] and should be used in order to be secure.
Elliptic curve Digital Signature Algorithm (ECDSA) uses the same principle as standard DSA, only the domain parameters change. For a ECDSA algorithm an elliptic curve should be used instead with a prime order. Table 3.4 shows the security level of ECDSA.

q	Output (bits)	Security levels
192	192	96
224	224	112
256	256	128
384	384	192
512	512	256

Table 3.4: Table of security levels of ECDSA domain parameters

4 Cryptography in the TLS protocol

4.1 SSL/TLS Protocol

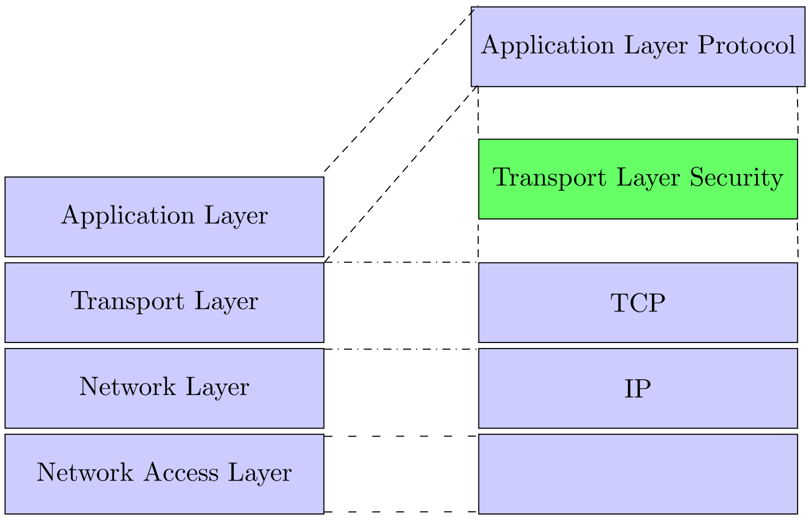


Figure 4.1: TLS protocol in OSI model

The Transport Layer Security (TLS) [4] provides communication security, privacy and data integrity, over the Internet. Through this protocol two communicating peers can communicate in a way which that is designed to prevent eavesdropping, tampering, or message forgery. The TLS protocol is situated between the Application Layer and the Transport Layer (e. g. TCP [13]) in OSI model (see figure 4.1).
The protocol is composed of two layers

1. Handshake Protocol
2. Record Protocol

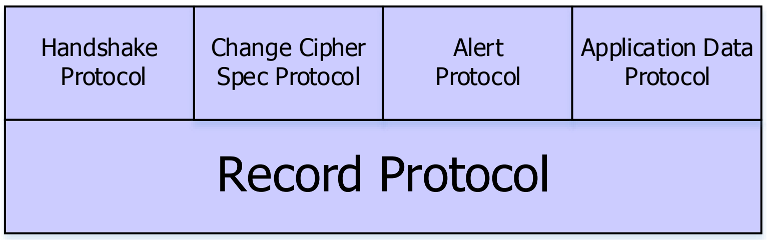


Figure 4.2: TLS Handshake Protocol

4.1.1 Record Protocol

TLS Record Protocol is used for encapsulation of higher-layer protocol data, such as TCP in case of the TCP/IP protocol [13]. It provides connection security with two basic properties:

1. Private connection, through symmetric cryptography algorithms (see section 3.1.4.1)
2. Reliable connection, through Message authentication Code (MAC) (see section 3.1.3)

4.1.2 Handshake Protocol

The Handshake protocol allows the two communicating peers (server and client) to authenticate each other and to negotiate a cipher suite (see section 4.1.3) and a compression method to transmit data. The Handshaking Protocol is composed of four protocols:

- Handshake Protocol
  (See above for the definition)

- Change Cipher Spec Protocol
  Allows to the communicating peers to signal a change of the ciphering strategy (see section 3.1.4)

- Alert Protocol
  Allows the communicating peers to signal potential problems and to exchange corresponding alert messages

- Application Data Protocol
  Is used for the secure transmission of application data

4.1.3 Cipher suites

A cipher suite [4] is used to cryptographically protect data in terms of authenticity, integrity and confidentiality. A cipher suite allows to know which key exchange, cipher and Message Authentication Code (MAC) will be used during the session. Figure 4.3 shows an example of a cipher suite.

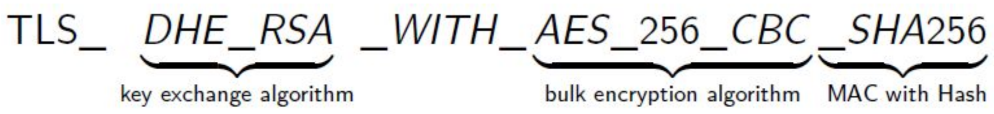

TLS_ DHE_RSA _WITH_ AES_256_CBC _SHA256
key exchange algorithm
bulk encryption algorithm
MAC with Hash

Figure 4.3: Example of a cipher suite

4.2 Cryptographic parts in the Handshake Protocol

This part describes step by step which cryptographic algorithms are used, how they work and why these algorithms are used, for each step of the Handshake Protocol [4].

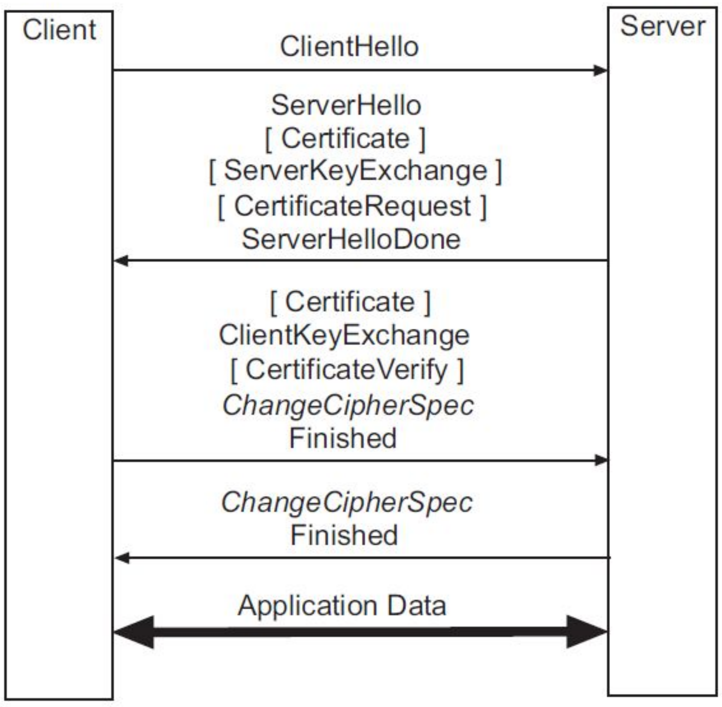


Figure 4.4: Handshake protocol (source [2])

Client Hello

A random number is generated as Client random number.
Creation of a hash function in the Client which will be used during the whole communication. Each received and transmitted data are added to this hash function, which allows to be sure that the two peer has become the same data. The data of the Client Hello are added to the hash function.
The cryptographic algorithms use in this message are:

- Random number (seed + random number)
- Hash algorithm (creation + update)

Server Hello

Generation of a Server random number. The cipher suite is chosen by the Server, which chooses with the list of cipher suites sent by the Client in the Client Hello.
Creation of a hash function in the Server which will be used during the whole communication. Each received and transmitted data are added to this hash function, which allows to be sure that the two peer has become the same data. The data from the Client Hello and of Server Hello are added to the hash function.
The cryptographic algorithms use in this message are:

- Random number (seed + generation)
- Hash algorithm (creation + update)

Certificate

The certificate message is sent from the Server. It contains certificates which each one contains public key certificates which contain public keys, digital signatures and other information. A public key is extracted and used to verify a digital signature which authenticate the server. The Client can also use this public key to verify the digital signature from the certificate and authenticates the Server. This message is added to the Client and Server hash function.
The Client sents its certificates when a Certificate Request has been sent from Server.
The cryptographic algorithm uses in this message is:

- Digital signature algorithm (verification)
- Hash algorithm (update)

Server Key Exchange

The Server Key Exchange is sent when Diffie-Hellman or Elliptic Curve Diffie-Hellman is used as key exchange. In this Server Key Exchange the domain parameters are sent, which are used to generate the public key of the client and of the server.
The public key of the server and the private key of the public key are used to generate the secret key.
The data of the Server Key Exchange are added to the hash function of the Client and the Server.
The cryptographic algorithms use in this message are:

- Diffie-Hellman/Elliptic Curve Diffie Hellman algorithm (Generation of key pair + calculation of the secret key)
- Hash algorithm (update)

Certificate Request

The Certificate Request message is used when the Server wants the certificates of the Client to authenticate it. The certificate message is added to the hash function of the Client and the Server.
Ony the hash algorithm (update) is used as cryptographic algorithm for this message.

Server Hello Done

The Server Hello Done is only added to the hash function of the two peer. Therefore is only the hash algorithm (update) used for this message.

Client Key Exchange

If the RSA is used as key exchange, the premaster key is computed with the Client and Server random number sent in the Client Hello and Server Hello with the HMAC function uses for PRF [4]. This premaster secret is encrypted with the public key of the Server, which is extacted from the certificate coming from the Server Hello. Only the Server which has the private key can decrypt this premaster key.
When the Server receives the Client Key Exchange, he computes the premaster key too. He can compare its premaster key with this one decrypted from the Client Key Exchange.

If the Diffie-Hellman or Elliptic Curve Diffie Hellman (ECDH) is used as key exchange, the public key generates by the Client, with the domain parameters send in the Server Key Exchange message, is sent to the Server. Then the Server can compute the secret key too.
The Client Key Exchange message is added to the hash function of the Client and the Server
The cryptographic algorithms use in this message are:

- Hash-based Message Authentication Code (HMAC) algorithm (Generation)
- Diffie-Hellman / Elliptic Curve Diffie-Hellman algorithm (Calculation of the secret key) if Diffie-Hellman / Elliptic Curve Diffie-Hellman is used as key exchange
- RSA algorithm (Decryption) if RSA is used as key exchange
- Hash algorithm (update)

Certificate Verify

The Client sent a digital signature, which has been done with the private key of the Client, to the Server, so can the Server authenticates the Client with the public extracts from the Client certificates. This message is added to the Client and Server hash function.
The hash algorithms use in this message are:

- Digital signature algorithm (Generation and verification)
- Hash algorithm (update)

Change Cipher Spec

This message allows to the communicating peers to signal transitions in ciphering strategies, meaning that the key exchange will be the symmetric one chooses in the cipher suite. No cryptographic algorithms are used for this message.

Finish

This message allows to the two peers to verify that the key exchange and authentication have been successful. The digest of the hash function uses since the beginning of the communication is computed. A Message Authentication Code is computed with the symmetric. The two results are encrypted with the symmetric key and the symmetric algorithm chooses in the cipher suite. The other peer can then decrypt the message with its symmetric key and compare the results. The message is added to the hash function of the two peers.
The cryptographic algorithms use in this message are:

- Symmetric cipher algorithm (encryption and decryption)
- Message Authentication Code (generation)
- Hash algorithm (computing digest, update)

Application Data

In this step, data can be transmitted over an insecure network without potential security problems. The data is signed with a Message Authentication Code (MAC) with the symmetric key. The data and the MAC are encrypted and decrypted with the symmetric key and the symmetric algorithm chooses in the cipher suites.
The cryptographic algorithms use in this message are:

- Message Authentication Code (generation)
- Symmetric cipher algorithm (encryption and decryption)

5 Generic Cryptographic Interface (GCI)

As explained in chapter 2, this Generic Cryptographic Interface (GCI) is implemented in the application and the cryptographic provider shall be easily changed for another one. The interface has to be optimized for other projects too, meaning that it should be as generic as possible. This interface should have a base of cryptography which can be easily adaptable for other applications and easily modifiable to use other providers.
Some constraints are added for the interface, which are:

- The link between the keys which could come from the application and needed in the provider and inversely shall be handled.
- No hidden states should be written in the function of the interface, meaning that no default parameters for an algorithm has to be written internally of a function.

In several books and on the internet (see Bibliography) the most important part of cryptography are listed below:

- Hash, which is oft used in signature and MAC algorithms
- Signature, which is widely used and also very important
- Message Authentication Code (MAC), which has some same properties as signatures, but some difference too, which make it important to have it in the interface.
- Cipher (symmetric and asymmetric), which are needed to encrypt and decrypt data

5.1 Context

As explained above, no hidden states should be written in the function of the interface, meaning that no default parameters for an algorithm has to be written internally of a function. For this constraint the principle of context is used. Through the context, a configuration of an algorithm can be done, meaning that in the application part, different parameters for this algorithm can be chosen and this configuration is saved internally in the interface (see figure 2.2). An ID of where is the configuration saved is returned to the application.
When an update has to be done, the ID is given and the interface knows that the update has to be done with this configuration. The cryptographic algorithms which the principle of context is used are:

- Hash algorithm
- Signature/MAC algorithm
- Cipher algorithm (symmetric and asymmetric)
- Diffie-Hellman

Through the context, only the parameters added in the configuration are used. No hidden state is therefore implemented. The other cryptographic services which don't need a context is because no configurations are needed to be saved.

5.2 Cryptographic services

5.2.1 Hash

A hash algorithm is an algorithm which is impossible to compute the input with the generated output (digest). It's used in some signature and Message Authentication Code (MAC), that's why is essential to have it in the interface. The hash algorithm in the interface is split into three main functions:

1. Configuration of the hash
2. Update of data
3. Calculation of the digest

Configuration of the hash

The hash algorithm has only one parameter to be configured, which is which algorithm should be used for hashing. The hash algorithm implemented in the interface are:

- MD5
- SHA1
- SHA224
- SHA256
- SHA384
- SHA512

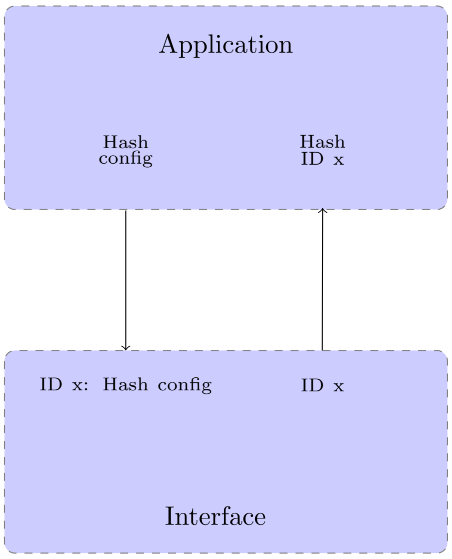


Figure 5.1: Hash - configuration

When the configuration is done this one should be sent to the interface which will save it in a context. The interface will then return an ID of the context, which corresponds of where is saved the configuration. This is shown on figure 5.1

Update of data

When the configuration is done, several updates can be done. The principle on an update is to add a data which we want to hash.

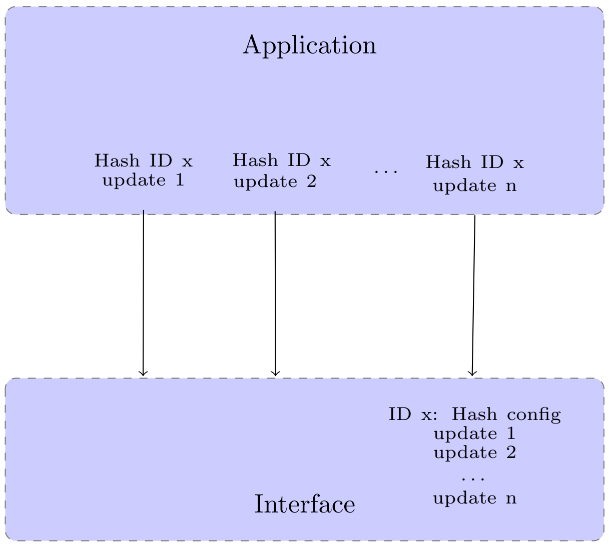


Figure 5.2: Hash - update

As shown in the figure 5.2, the ID received in the configuration part has to be used to add a data. The ID and the data has therefore sent to the interface, which knows through the context ID that it should hash the data with the configuration saved in this context.

Calculation of the digest

When all the data we want to hash are sent to the interface, the interface can calculate the digest. As shown in figure 5.3, by passing the ID which contains the configuration and all the updated data, the interface will, through the provider, calculate the digest. One of the disadvantages of this part is when the digest is calculated, all the updated data and the configuration are lost, meaning that we cannot use them again to calculate another hash with other data. This problem is solved in chapter 5.3

5.2.2 Generate key pair

Some parts of cryptography need keys to work, like signatures and ciphers. The key can be a symmetric, which is the same key for the two peers in a communication or asymmetric, which is different for the two peers in a communication. The goal of this function is to generate the asymmetric keys, public key and private key. To generate a symmetric key, other methods should be used like Diffie-Hellman. In the interface, three kinds of key pair can be generated, each having a different configuration:

- RSA key pair
  The size of the key should be configured (1024 bits, 2048 bits, 4096 bits, etc.)

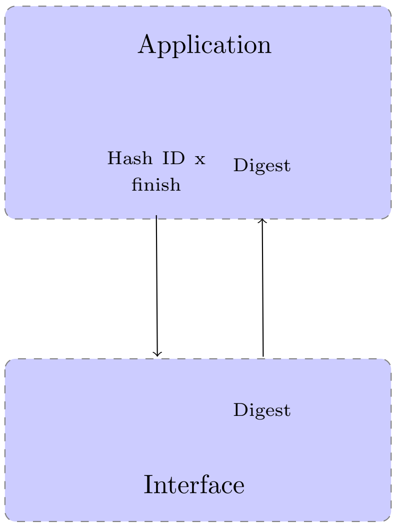


Figure 5.3: Hash - finish

- DSA key pair
  The domain parameters can be configured if needed or internally generated

- ECDSA key pair (Elliptic curve)
  The type of the elliptic curve should be configured

When the configuration is done, this one should be sent to the interface. The keys will then be generated through a cryptographic provider. The keys are, however, returned as IDs, which is one of the principles of the interface. For more details about the key ID see chapter key management 5.4

5.2.3 Digital signature - Message Authentication Code (MAC)

The digital signature and the MAC are widely used today. Both provide integrity and authentication of a message. Only the digital signature provides non-repudiation more. MAC is much faster than digital signature through the use of symmetric keys. As some specifications of certain provider, the signature/MAC function has two possibilities of use:

1. Signing
2. Verifying

Each one is split into three functions:

- Configuration of the signature
- Update of data
- Generate a signature/Verify a signature

Configuration of the signature

For the generation and verification of a signature this part is the same (only the name of the function changes). First should the signature be configured. Several parameters have to be configured which are:

- the signature/MAC algorithm, which could be:
  - RSA
  - DSA
  - ECDSA
  - CMAC (MAC from Block Ciphers)
  - HMAC (MAC from Hash functions)
- A hash algorithm, if in the updated data should be first hashed before signing, or if the HMAC algorithm is used
- The padding, if the RSA algorithm is used
- The block mode, the padding and the initialization vector, if the CMAC algorithm is used
- The private key, if RSA, DSA or ECDSA is used to generate a signature or the public key if the same algorithm is used to verify a signature

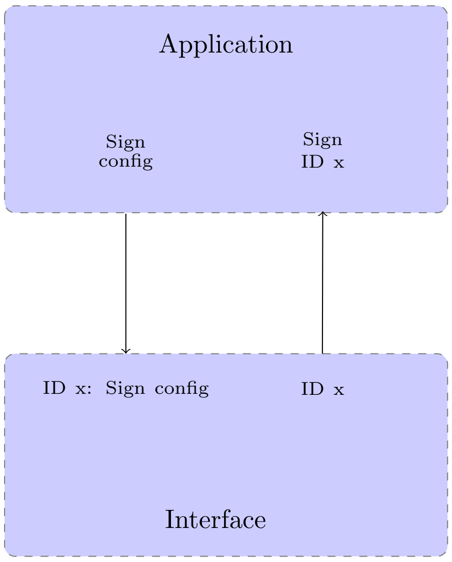


Figure 5.4: Signature - configuration

As shown figure 5.4, when the configuration is done, this one is saved in a context in the interface. An ID of the context is returned, which indicated where is the configuration saved.

Update of data

When the configuration is done, several updates could be done.
The principle of the update is to add data which will be used to generate or verify a signature.

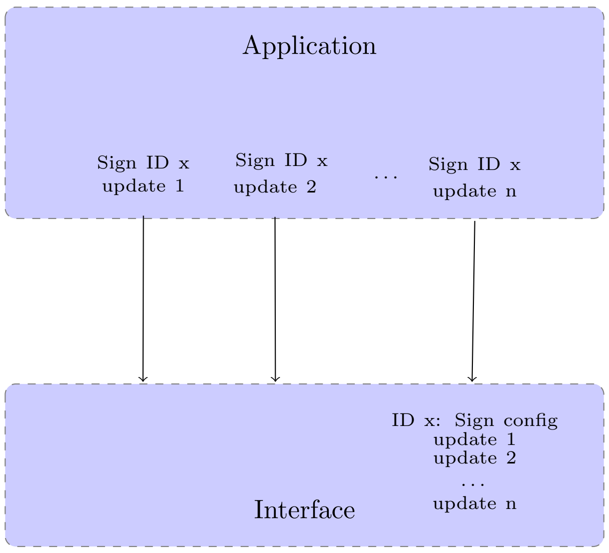


Figure 5.5: Signature - update

As shown in figure 5.5, to use the correct configuration, the ID of the context, returned when the configuration is saved, should be used. The data and the ID is then sent to the interface, which will sign this data with the configuration saved in this context.

Generate a signature/Verify a signature

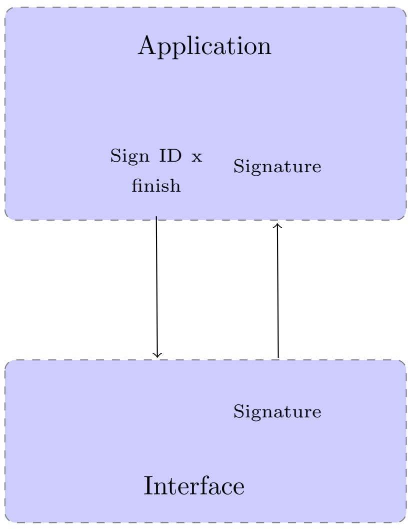


Figure 5.6: Signature - finish

In this part the generation and the verification are different. For the generation , the whole updated data and the configuration will be signed with the private key added in the configuration for the digital signature but with a symmetric key for the MAC.
For the verification, the signature we want to verify should be added to the function. Then the updated data will be signed, but with the public key for the digital signature, and the same symmetric key as the generation of the signature for the MAC. Then the added signature (which is done with the private key) could be compared with the "signature" computed to verify. The most important part of the verification is that the private key, which the signature is done and the public key for the verification, should be generated together for the digital signature and the same symmetric key should be used for the MAC.

5.2.4 Cipher (symmetric and asymmetric)

A cipher, as explained in 3.1.4, which could be symmetric or asymmetric, is an algorithm for encrypting and decrypting data. This concept is therefore used in the interface and split into three main functions which are:

- Configuration of the cipher
- Encryption of a plaintext
- Decryption of a ciphertext

Configuration of the cipher

Several parameters are to be configured for the cipher algorithm and depends particularly of which cipher algorithm is used. The cipher algorithm is split into three main algorithm:

- Symmetric stream cipher algorithm
  Today very deprecated but is, however, implemented in the interface if comparison has to be done. Only the RC4 stream cipher is implemented in the interface. Other stream ciphers can, however, be easily added in the interface if needed. Nothing more as the algorithm has to be configured for the use of it.

- Symmetric block cipher algorithm
  Three kinds of symmetric block cipher algorithms are used today and therefore implemented in the interface:
  - Data Encryption Standard (DES)
  - 3DES, three subsequent DES encryption
  - Advanced Encryption Standard (AES)

  Each symmetric block cipher algorithm needs a mode of operation, named block mode in the interface, which depends on the size of data we want to encrypt.
  The block modes implemented in the interface are:
  - Electronic Code Book mode (ECB)
  - Cipher Block Chaining mode (CBC)
  - Cipher Feedback mode (CFB)
  - Output Feedback mode (OFB)
  - Counter mode (CTR)
  - Galois Counter Mode (GCM)

- Asymmetric algorithm
  Only the RSA algorithm is implemented for this part of the interface. In practice to use the

RSA algorithm this one should use a padding to increase the security of it. The best known padding in the Public-Key Cryptography Standard (PKCS) and is, of course, implemented in the interface.

The most important thing for a cipher is, of course, the key! For a symmetric cipher, stream or block, the key is only a shared key. For an asymmetric cipher, if an encryption will be done, a public key should be added. If a decryption will be done, a private key should be added to the configuration.

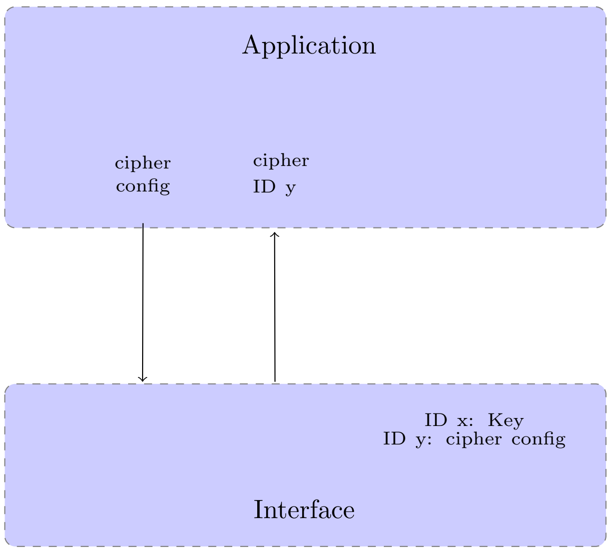


Figure 5.7: Cipher - configuration

When the configuration is done this one should be sent to the interface which will save it in a context. The interface returned an ID of the context, which corresponds of where is saved the configuration if this one should be used in the future. The key added to the function is an ID of the key which is already saved in the interface. This principle is shown in figure 5.7

Encryption of a plaintext

When the configuration is done, a encryption can be done. To encrypt data, the ID of the context (where is the configuration saved) should be added to the function with the data to encrypt (plaintext). The interface will, through a provider, calculate the ciphertext of the plaintext with the configuration saved previously in the context. This principle is shown figure 5.8

Decryption of a ciphertext

When the configuration is done, a decryption can be done. To decrypt a data, the ID of the context (where is the configuration saved) should be added to the function with the data to decrypt (ciphertext). The interface will, through a provider, calculate the plaintext of the ciphertext with the configuration saved previously in the context. This principle is shown figure 5.9

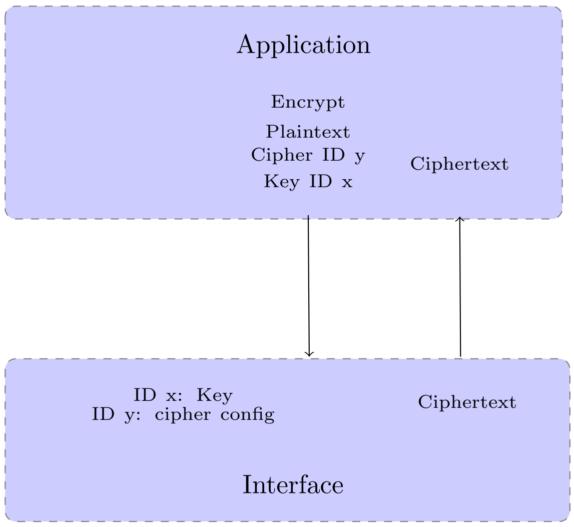


Figure 5.8: Cipher - Encryption

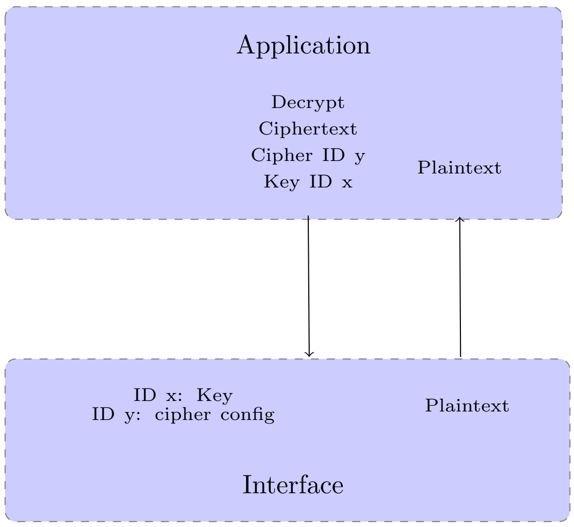


Figure 5.9: Cipher - Decryption

5.2.5 Diffie-Hellman

Diffie-Hellman key exchange is a specific method of securely exchanging cryptographic keys over an insecure network. It starts with asymmetric keys to finish with a symmetric, shared key which is the same for the two peer in a communication.
The Diffie-Hellman key exchange should therefore have the possibility to generate key pairs and to calculate the shared key.
That's why in the interface the Diffie-Hellman protocol is split into three main functions:

- Configuration of the Diffie-Hellman protocol
- Generation key pair
- Calculation of the shared key

Configuration of the Diffie-Hellman protocol

The Diffie-Hellman key exchange and the Elliptic Curve of Diffie-Hellman key exchange can be used in the interface. For the configuration, one of these key exchanges should be chosen.
For the Diffie-Hellman key exchange the domain parameters can be added. If no domain parameters have been added, the interface will, through the provider, generate them.
For the Elliptic Curve of Diffie-Hellman key exchange, the type of curve should be configured.

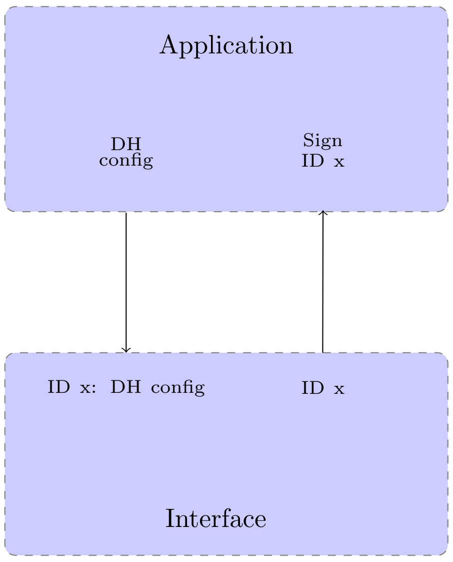


Figure 5.10: DH - Configuration

As shown on figure 5.10, when the configuration is done, this one should be sent to the interface which will save it in a context. An ID of the context will be returned to the application, which indicates the placement of the context, also of the configuration.

Generation of the key pair

When the configuration is done, key pair can be generated. It's the same principle as explain in the chapter 5.2.2, but for the Diffie-Hellman it's a little bit different. The private key of the Diffie must not go out of the interface. That's why when the key should be generated only the ID of the public key will go out of the function. The private key will be saved in the context, same context where is the configuration saved. This is why Diffie-Hellman is not implemented with the asymmetric cipher algorithm, where the public key and private key are going out of the function and don't stay in the context. This is shown on figure 5.11

Calculation of the shared key

When the public key of the peer is received, the shared key could be generated. To do it, the context with the configuration and the private, and the public key of the peer should be added to the

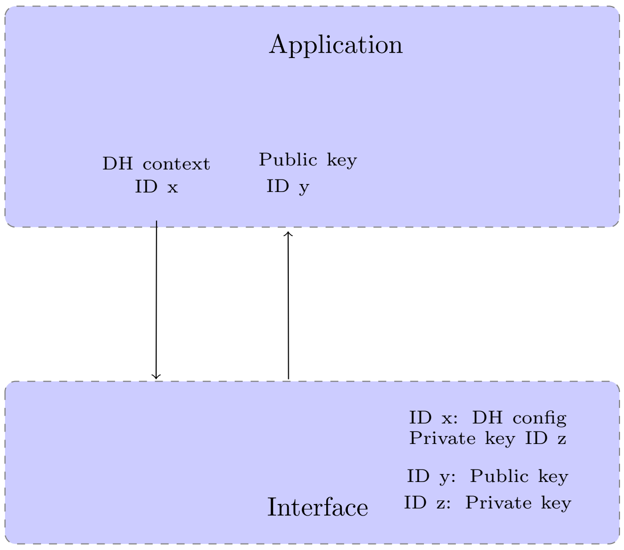


Figure 5.11: Diffie-Hellman - Generate key pair

interface. This one will, through a provider calculate the shared key and returned the ID of this one. This is shown on figure 5.12

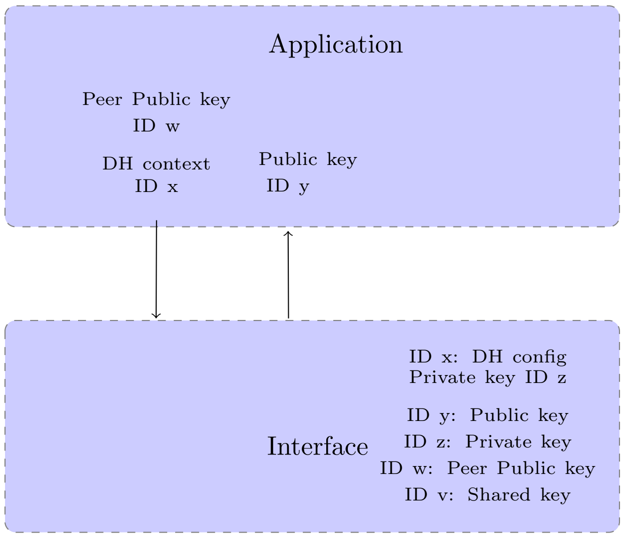


Figure 5.12: Diffie-Hellman - Calculation of the shared key

5.2.6 Random number generator

A random number generator is a computational or physical devices for generating a sequence of numbers which are impossible to predict better than a random chance.
It exists two main random number generator:

1. True Random Number Generators (TRNG)
   Are characterized that the output cannot be reproduced. They are based on physical processes like semiconductor noise or clock jitter in digital circuits, etc..

2. Pseudorandom Number Generators (PNRG) Generated sequences which are computed from an initial seed value. PRNGs possess good static properties, meaning their output approximates a sequence of true random numbers. This is shown on figure **??**

For the interface, Random number generators are very important, to generate keys or for the ciphers, for example.
For the use of a True Random Number Generator (TRNG), with hardware-based cryptographic modules for example, only the function to get a random number is needed.
For the use of Pseudo Random Number Generator (PRNG), with a cryptographic software library for example, a function to generate the initial seed value is, furthermore, needed.

5.3 Clone of context

As explained in 5.2.1 and 5.2.3 when the digest (for the hash) and the signature (for the digital signature/MAC) is calculated, no more data can be added to the context. This is a problem for the use of this interface in TLS projects (emb::TLS for example). Several solutions were introduced which are:

1. Use two contexts at the same time.
   This wasn't very efficient, because we should know at the beginning the number of times a digest will be calculated, which determines the amount of context we have to create at the beginning.

2. Create a context when the digest is calculated.
   The disadvantage of this idea was that the whole data use previously has to be saved. For applications, which are used in embedded systems, like emb::TLS, For systems, like embedded systems, with memory constraints, this is not possible.

3. Clone the context. This is the solution uses for the interface.

As shown figure 5.13, when we need to compute a result, but the whole data added previously are needed for a future result, the solution is to clone the context, meaning that the whole data added and the configuration is copied in another context.
Then one context could be used to compute the result and the other one to add other data when needed.

5.4 Key management

In the previous version of the emb::TLS project interacted directly the provider and the application. This interaction was used for example to generate and to become directly a key or to compute something with a key coming from the other peer. With the interface this is not possible anymore. That's why the key management is required for the interface. Through the interface the keys can be saved and used by the provider to compute something or by the application to send it to another peer.

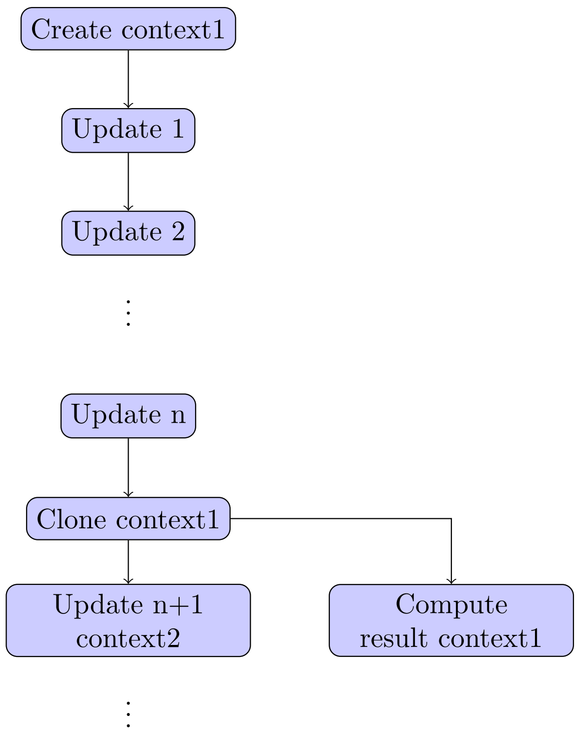


Figure 5.13: Context - clone example

Put a key

As said in chapter 5.2.2 Generate key pair, when a key is generated, the ID of the key is returned to the application. This is because the key is saved in the interface, through the function " Put key ".

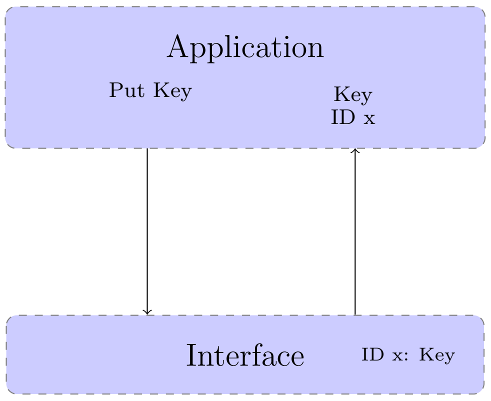


Figure 5.14: Key management - put a key

As shown on figure 5.14, when a key should be stored in the interface, the key is added to the function. The interface saves it in a specific place and return the ID of where is the key stored.

Get a key

When the key should be used to be sent to another peer or to compute something like a signature, it should be possible to get it.

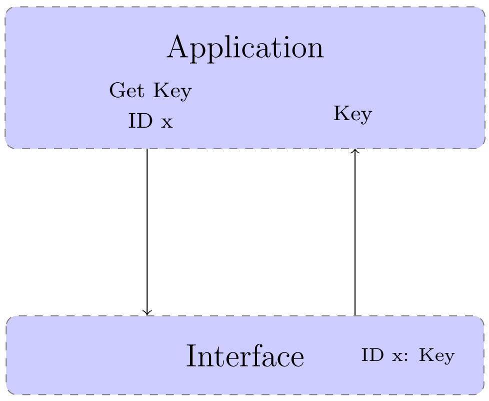


Figure 5.15: Key management - get a key

As shown on figure 5.15, when the key is needed by the application, the ID should be sent the interface which will return the key stored at this ID.
The key stays in the interface if it's still needed.

Delete a key

When the key is not needed anymore, it should be possible to delete it to free space for other keys.

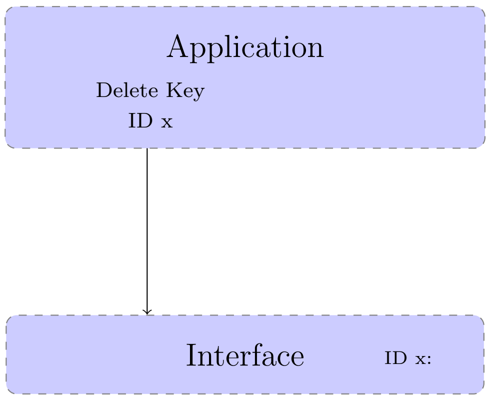


Figure 5.16: Key management - delete a key

As shown on figure 5.16, when the key is not needed anymore, the ID of it is sent to the interface, which will delete it.

6 Implementation

In this chapter what is implemented and what was the problems in the implementation will be explained. The implementation has been done in two parts. The first part is the implementation of the interface in the application, emb::TLS, and the second part the implementation of a provider, LibTomCrypt, in the interface.

6.1 Interface for an application

The implementation of the interface for the application is, in fact, just a change of was it was already done. The function of the interface which was implemented are:

- Hash algorithm
- Signature algorithm (Generation and verification)
- Message Authentication Code (HMAC)
- Cipher algorithm (symmetric, asymmetric, for encryption and decryption)
- Random number generator
- Diffie Hellman (generation of key pair and calculation of the secret key)
- Key management
- Context management

The key pair generator function isn't used in this application, because when the RSA, DSA or ECDSA are used as key pair exchange, the private key and public key are in the Certificates.
All parameters in the application which need memory allocation was replaced as well as possible with context and key management. For the configuration of each context for any algorithm, all parameters must be configured, and if a parameter isn't needed, this one must be configured like "Name_of_the_parameters"_None. Before getting an ID of a context and of a key, this ID must be equal to "-1", because some context does different steps depending on the value of the incoming ID in the interface.
The problems occurred during the implementation of the interface for the application was:

- When sending a Server Key Exchange, the parameters to send are in a certain order. In the old implementation, this part was done with functions from the provider, but these functions are not handled by the new interface. The solution was to rewrite this function but directly in the application part and without using the functions from the provider. To do that, the sequence to send the parameters should be known, which are represented in figure 6.1. With the figure 6.1 and the use of the software Wireshark to understand the meaning of what was done in the old implementation, it was easier to rewrite this part of the Server Key Exchange. This problem and solution is the same for receiving a Server Key Exchange, and quick the same for sending and receiving the public key in the Client Key Exchange.

- When extracting the public key of a certificate, when the Certificates are coming from the Server, this part has been done with functions from the provider in the old implementation. The functions use ASN.1 DER encoding rules [7] function, which are very complex and this is not the goal of the project to rewrite this sort of function. That's why another source file, named “gci_tomcrypt”, has been created to use the function from the provider, but with the goal to not have any part of the provider written in the application part.

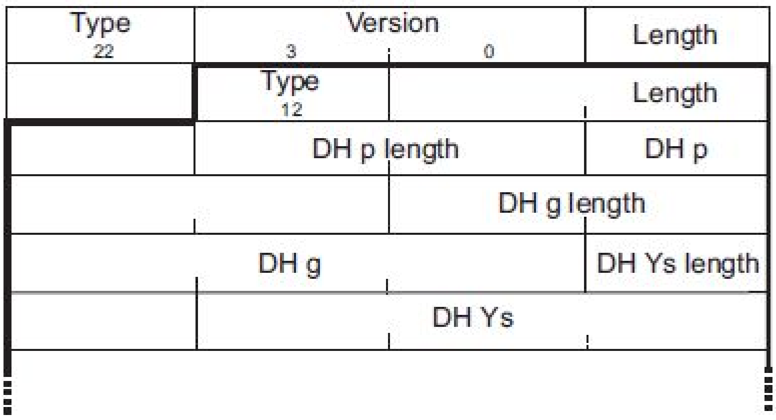


Figure 6.1: Server Key Exchange - Diffie-Hellman as key exchange (source [2])

6.2 Provider for the interface

The implementation of the LibTomCrypt provider to the Generic Cryptographic Interface (GCI) is important to do all the calculation needed for each algorithm needed by the application.
For the hash algorithm context (includes the clone of the context of the hash algorithm), the hash algorithms implemented are:

- MD5 algorithm, which is used by the emb::TLS application and works
- SHA1 algorithm, which is used by the emb::TLS application and works
- SHA224 algorithm, which isn’t used by the emb::TLS application and also not tested
- SHA256 algorithm, which is used by the emb::TLS application and works
- SHA384 algorithm, which isn’t used by the emb::TLS application and also not tested
- SHA512 algorithm, which isn’t used by the emb::TLS application and also not tested

All the hash algorithms have been implemented, but not all have been tested, because they aren’t used by the emb::TLS application.
For the generation of the signature, the signature/Message Authentication Code (MAC) algorithms implemented are:

- Hash-based Message Authentication Code (HMAC), which is used by the emb::TLS application and works
- RSA algorithm, which is used by the emb::TLS application and works

The signature/Message Authentication Code (MAC) algorithms, which aren’t implemented are:

- Cipher-based Message Authentication Code (CMAC)
- DSA
- ECDSA

These algorithms weren’t implemented, because they aren’t used by the emb::TLS application.
The clone of a signature isn’t implemented too, because it’s not needed for the emb::TLS application

For the verification of the signature, the signature/Message Authentication Code (MAC) algorithm implemented is the hash-based Message Authentication Code (HMAC). The other algorithms (see above, for the generation of a signature) aren't needed for the emb::TLS application.
For the generation of key pair (RSA, DSA or ECDSA) none has been implemented, because none is needed for the emb::TLS application.
For the cipher (symmetric and asymmetric) algorithms, these implemented are:

- RC4, tested and works
- DES, implemented but not tested
- 3DES, tested and works
- RSA, tested and works

The block mode for the symmetric algorithms implemented are:

- CBC
- CFB
- ECB
- GCM
- OFB

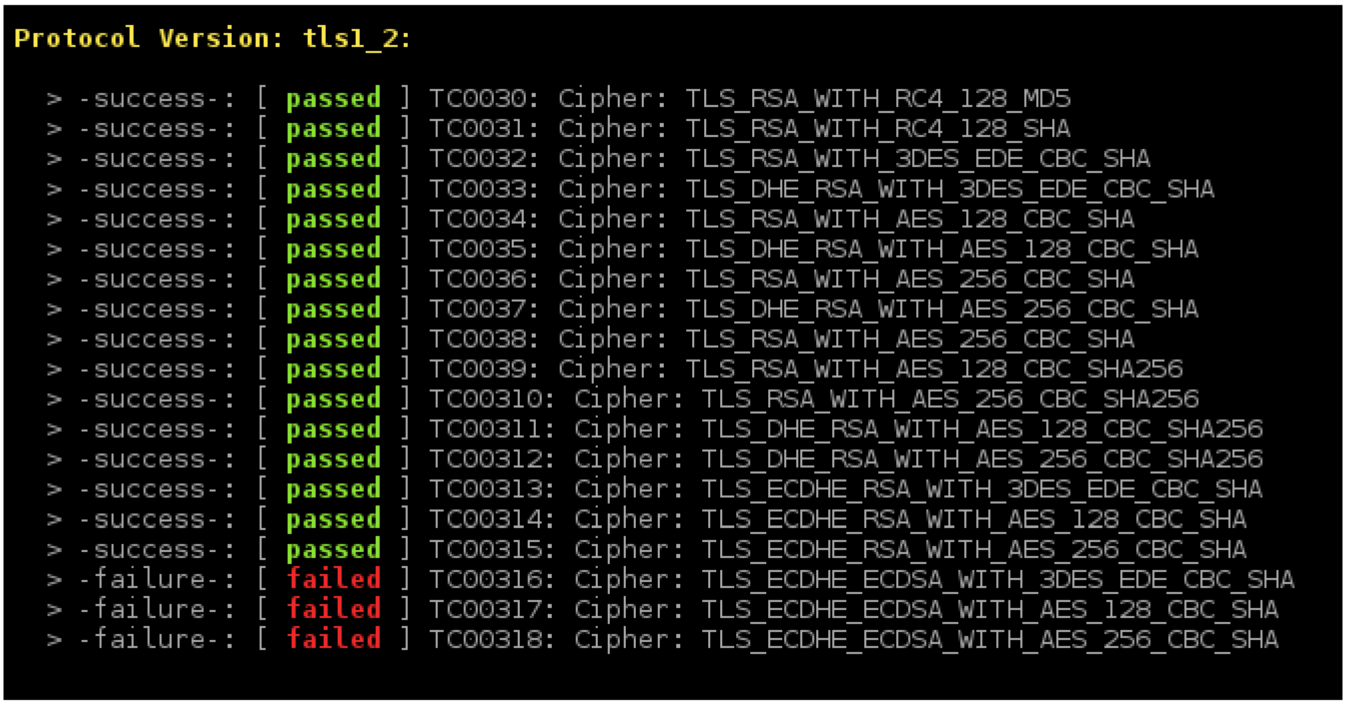
Only the CBC block mode is used by the emb::TLS application, the others are implemented, but not tested.
Generation of a Random Number (included the seed) is implemented and works
Diffie-Hellman (and Elliptic Curve of Diffie-Hellman (ECDH)) is implemented and works.
The context management and key management are implemented and work too.
For the algorithms, which has been implemented but not tested, are implemented, because the implementation doesn't change a lot compared to this which was implemented and worked (Hash algorithm and Block mode for the symmetric cipher).
Problems occurred for the implementation of the LibTomCrypt provider for the Generic Cryptographic Interface:

- Generation of the signature
  The provider needs, of course, the private key but the public key too to generate the signature. In the interface, only one key at a time can be added to the context. To resolve this problem, first should a context for the signature be created, where the configuration of the signature and the private key are saved. To begin, the context ID must be initialized to "-1". This context ID will, after the context is created, be greater or equal to "0". Then the function to create the context should be used a second time with the context ID previously returned. The public key should be added at this time. The interface checks that the context ID is equal to "-1", which is not the case. Another context won't be created, but the public key will be added to the context. The context has, now, the private and public key to generate the signature.

- When the encryption of the Message Authentication Code (MAC) has to be done, depending on the TLS version, if it's greater or equal to TLS 1.1, an Initalialization Vector (IV) has to be added to the cipher context before encrypting. The problem is that, when we arrived at this step, we cannot recreate a cipher context with the key needed to encrypt, because this key is, at this step, no more available. The only solution is to use the function to create a context and to use the context ID returned previously, when the context was created with the symmetric key. The context ID is different to "-1", also a new context won't be created, but the new Initialization Vector (IV) [20] will be added to this context.

7 Results

In this chapter, the result of the implementation will be explained. To test the implementation, several cipher suites have been used. These cipher suites are shown in the figure 7.1. As we can see, almost all cipher suites work, only the cipher suites which use ECDSA as key exchange don't work. This is because the ECDSA implementation has been done in another project and the merge between the two project wasn't done, because of the time.



```
Protocol Version: tls1_2:

  > -success-: [ passed ] TC0030: Cipher: TLS_RSA_WITH_RC4_128_MD5
  > -success-: [ passed ] TC0031: Cipher: TLS_RSA_WITH_RC4_128_SHA
  > -success-: [ passed ] TC0032: Cipher: TLS_RSA_WITH_3DES_EDE_CBC_SHA
  > -success-: [ passed ] TC0033: Cipher: TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA
  > -success-: [ passed ] TC0034: Cipher: TLS_RSA_WITH_AES_128_CBC_SHA
  > -success-: [ passed ] TC0035: Cipher: TLS_DHE_RSA_WITH_AES_128_CBC_SHA
  > -success-: [ passed ] TC0036: Cipher: TLS_RSA_WITH_AES_256_CBC_SHA
  > -success-: [ passed ] TC0037: Cipher: TLS_DHE_RSA_WITH_AES_256_CBC_SHA
  > -success-: [ passed ] TC0038: Cipher: TLS_RSA_WITH_AES_256_CBC_SHA
  > -success-: [ passed ] TC0039: Cipher: TLS_RSA_WITH_AES_128_CBC_SHA256
  > -success-: [ passed ] TC00310: Cipher: TLS_RSA_WITH_AES_256_CBC_SHA256
  > -success-: [ passed ] TC00311: Cipher: TLS_DHE_RSA_WITH_AES_128_CBC_SHA256
  > -success-: [ passed ] TC00312: Cipher: TLS_DHE_RSA_WITH_AES_256_CBC_SHA256
  > -success-: [ passed ] TC00313: Cipher: TLS_ECDHE_RSA_WITH_3DES_EDE_CBC_SHA
  > -success-: [ passed ] TC00314: Cipher: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA
  > -success-: [ passed ] TC00315: Cipher: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA
  > -failure-: [ failed ] TC00316: Cipher: TLS_ECDHE_ECDSA_WITH_3DES_EDE_CBC_SHA
  > -failure-: [ failed ] TC00317: Cipher: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA
  > -failure-: [ failed ] TC00318: Cipher: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA
```

Figure 7.1: Result of the implementation

8 Conclusion

Through this project I get a lot of knowledge about cryptography. It allows me to learn all the main cryptography algorithms use in cryptography, the functioning of them and why these algorithms are used today.
I learned a lot about the TLS protocol too. It allows me to understand all the step needed to get a secured communication, and that a lot of verification has to be done to be sure that the communication is secure.

8.1 Achieved work

The project is, unfortunately, completely achieved, but globaly the most important parts are done.
The project was split into four parts.

1. Get the knowledge about the main cryptography part and where it could be used in the TLS protocol
2. Design of the new interface
3. Implementation of the interface into the emb::TLS application
4. Implementation of the LibTomCrypt provider into the Generic Cryptographic application

The achieved work in this project is the design of the new Generic Cryptographic Interface (GCI)), and globally the implementation of the interface in the application and the implementation of the provider in the interface.
This is not completely done, because all the cipher suites shown in the chapter **??** doesn't work in all case. There is two parts in the project, when the application, emb::TLS, work as Client and as Server.
The figure 7.1 shown the result for emb::TLS as Client. When emb::TLS work as Server, all the cipher suites with the Diffie-Hellman and the Elliptic Curve Diffie-Hellman (ECDH) don't work. The problem should come in the Server Key Exchange part, because this is the only part which changed compared to the other cipher suites.

8.2 Future work

The future work which could be done with this project would be to finish the implementation of the emb::TLS as Server (the Diffie-Hellman and Elliptic Curve Diffie-Hellman, which don't work), the implementation of the ECDSA algorithm as key exchange, which was done in another project, the test of other cipher suites to increase the use of this new Generic Cryptographic Interface (GCI), and the most important would be to write the documentation of the new Generic Cryptographic Interface, for other future projects which will need this interface as support.

Bibliography

Books

[1] Christof Paar and Jan Pelzl. *Understanding cryptography: a textbook for students and practitioners*. Springer Berlin Heidelberg, Berlin; Heidelberg [u.a.], 2. corr. printing edition, 2010.

[2] Ph.D. Rolf, Oppliger. *SSL and TLS: Theory and Practice*. Artech House, eSECURITY Technologies; Beethovenstrasse 10; CH-3073; Gümligen; Switzerland, 2009.

Internet

[3] S. Blake-Wilson, N. Bolyard, V. Gupta, C. Hawk, and B. Moeller. Elliptic curve cryptography (ecc) cipher suites for transport layer security (tls). RFC 4492, RFC Editor, May 2006. `http://www.rfc-editor.org/rfc/rfc4492.txt`. URL: `http://www.rfc-editor.org/rfc/rfc4492.txt`.

[4] T. Dierks and E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, RFC Editor, August 2008. `http://www.rfc-editor.org/rfc/rfc5246.txt`. URL: `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[5] S. Frankel, R. Glenn, and S. Kelly. The aes-cbc cipher algorithm and its use with ipsec. RFC 3602, RFC Editor, September 2003.

[6] K. Igoe and J. Solinas. Aes galois counter mode for the secure shell transport layer protocol. RFC 5647, RFC Editor, August 2009. `http://www.rfc-editor.org/rfc/rfc5647.txt`. URL: `http://www.rfc-editor.org/rfc/rfc5647.txt`.

[7] TELECOMMUNICATION STANDARDIZATION SECTOR (ITU-T) INTERNATIONAL TELECOMMUNICATION UNION. Itu-t recommendation x.690 (2002) | iso/iec 8825-1:2002, information technology - asn.1 encoding rules: Specification of basic encoding rules (ber), canonical encoding rules (cer) and distinguished encoding rules (der), 2002. URL: `https://www.itu.int/ITU-T/studygroups/com17/languages/X.690-0207.pdf`.

[8] J. Jonsson and B. Kaliski. Public-key cryptography standards (pkcs) #1 Rsa cryptography specifications version 2.1. RFC 3447, RFC Editor, February 2003. `http://www.rfc-editor.org/rfc/rfc3447.txt`. URL: `http://www.rfc-editor.org/rfc/rfc3447.txt`.

[9] S. Kelly. Security implications of using the data encryption standard (des). RFC 4772, RFC Editor, December 2006.

[10] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. Hmac: Keyed-hashing for message authentication. RFC 2104, RFC Editor, February 1997. `http://www.rfc-editor.org/rfc/rfc2104.txt`. URL: `http://www.rfc-editor.org/rfc/rfc2104.txt`.

[11] A. Popov. Prohibiting rc4 cipher suites. RFC 7465, RFC Editor, February 2015. http://www.rfc-editor.org/rfc/rfc7465.txt. URL: http://www.rfc-editor.org/rfc/rfc7465.txt.

[12] T. Pornin. Deterministic usage of the digital signature algorithm (dsa) and elliptic curve digital signature algorithm (ecdsa). RFC 6979, RFC Editor, August 2013. http://www.rfc-editor.org/rfc/rfc6979.txt. URL: http://www.rfc-editor.org/rfc/rfc6979.txt.

[13] Jon Postel. Transmission control protocol. STD 7, RFC Editor, September 1981. http://www.rfc-editor.org/rfc/rfc793.txt. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[14] S. Turner and L. Chen. Updated security considerations for the md5 message-digest and the hmac-md5 algorithms. RFC 6151, RFC Editor, March 2011. http://www.rfc-editor.org/rfc/rfc6151.txt. URL: http://www.rfc-editor.org/rfc/rfc6151.txt.

[15] Wikipedia. Elgamal signature scheme — wikipedia, the free encyclopedia, 2015. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=ElGamal_signature_scheme&oldid=697104703.

[16] Wikipedia. Cbc-mac — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=CBC-MAC&oldid=704419839.

[17] Wikipedia. Digital signature — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=Digital_signature&oldid=705058306.

[18] Wikipedia. Hash-based message authentication code — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=Hash-based_message_authentication_code&oldid=704669944.

[19] Wikipedia. Hash function — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=Hash_function&oldid=704553479.

[20] Wikipedia. Initialization vector — wikipedia, the free encyclopedia, 2016. [Online; accessed 28-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=Initialization_vector&oldid=706266589.

[21] Wikipedia. Public-key cryptography — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=Public-key_cryptography&oldid=706619212.

[22] Wikipedia. Symmetric-key algorithm — wikipedia, the free encyclopedia, 2016. [Online; accessed 24-February-2016]. URL: https://en.wikipedia.org/w/index.php?title=Symmetric-key_algorithm&oldid=706507554.

Appendices

Appendix A

Documentation Interface

This documentation lists all functions use in the Generic Cryptographic Interface (GCI) and explains step by step the working of each cryptographic services definied in the interface.