***Tropyc*: A Pattern Language for Cryptographic Software**

Alexandre M. Braga *Cecília M. F. Rubira*
Ricardo Dahab

Relatório Técnico IC–99-03

Janeiro de 1999

Tropyc: A Pattern Language for Cryptographic Software[†]

Alexandre M. Braga Cecília M. F. Rubira Ricardo Dahab

State University of Campinas
Institute of Computing
P.O. Box 6176
13081-970 Campinas-SP-Brazil
phone/fax:+55+19+7885842
e-mail:{972314,cmrubira,rdahab}@dcc.unicamp.br

Abstract

This work describes *Tropyc*, a pattern language for cryptographic software based on a generic object-oriented cryptographic architecture. Nine patterns are described: *Information Secrecy*, *Sender Authentication*, *Message Integrity*, *Signature*, *Signature with Appendix*, *Secrecy with Integrity*, *Secrecy with Sender Authentication*, *Secrecy with Signature*, and *Secrecy with Signature with Appendix.* They are classified according to four fundamental objectives of cryptography (confidentiality, integrity, authentication and non-repudiation) and compose a closed set of patterns for this domain. These patterns have the same dynamic behavior and structure. We abstracted these aspects into a *Generic Object-Oriented Cryptographic Architecture* (**GOOCA**).

Key words: cryptography, pattern language, design patterns, software architecture, object orientation.

1 Introduction

Modern cryptography is been widely used in many applications, such as word processors, spreadsheets, databases, and electronic commerce systems. The widespread use of cryptographic techniques and the present interest and research on software architectures and patterns led us to cryptographic software architectures and cryptographic patterns. In this work, we present a generic object-oriented cryptographic architecture plus a set of cryptographic patterns, based on that architecture, classified according to the objectives of cryptography [MvOV96] (which are confidentiality, integrity, authentication and non-repudiation, as summarized in Appendix A) and organized as a pattern language for cryptographic software.

This paper is targeted to software engineers who have to deal with cryptography-based security requirements of object-oriented applications, but do not have much experience

[†]An early version of this paper was published in the 5th Pattern Languages of Programming (PLoP'98) Conference

in cryptography. People looking for general software architectures for their cryptographic software or components are another group of interest. This work is organized as follows: Section 2 proposes *Tropyc*, a pattern language for cryptographic software, and a pattern describing a generic software architecture for cryptographic object-oriented applications. Section 3 is an application example using the pattern language: a tool for electronic purchase and on-line distribution of hypertext documents over the Internet, called *PayPerClick*. Section 4 contains patterns related to cryptographic services. The cryptography-based security aspects of *PayPerClick* are implemented as instantiations of our cryptographic patterns in Section 5. Conclusions and future work are in Section 6. A brief introduction to cryptography is shown in Appendix A.

2 A Pattern Language for Cryptographic Software

In this section we propose *Tropyc*, a pattern language for cryptographic software. Our proposal focuses on three goals: (*i*) the definition of a Generic Object-Oriented Cryptographic Architecture (GOOCA for short); (*ii*) the description of cryptographic services as patterns and (*iii*) the organization of these cryptographic patterns as a pattern language. Table 2 summarizes all patterns, their scopes and purposes. *Tropyc* deals with two kinds of forces treated separately: those acting over a specific cryptographic service and those concerned with instantiating GOOCA. Patterns in Section 4 focus on the requirements and constraints acting over services and their compositions. GOOCA focuses on the second type, that is, those forces acting over the instantiation of our generic structure and behavior. In cryptography, the ends of a communication channel are usually called Alice and Bob; Eve is usually an adversary eavesdropping the channel.

Pattern	**Scope**	**Purpose**
GOOCA	Generic	generic software architecture for cryptographic applications
Information Secrecy	Confidentiality	provide secrecy of information
Message Integrity	Integrity	detect corruption of a message
Sender Authentication	Authentication	authenticate the origin of a message
Signature	Non-repudiation	provide the authorship of a message
Signature with Appendix	Non-repudiation	separate message from signature
Secrecy with Integrity	Confidentiality and Integrity	detect corruption of a secret
Secrecy with Sender Authentication	Confidentiality and Authentication	authenticate the origin of a secret
Secrecy with Signature	Confidentiality and Non-repudiation	prove the authorship of a secret
Secrecy with Signature with Appendix	Confidentiality and Non-repudiation	separate secret from signature

Table 2: The Cryptographic Patterns and Their Purposes.

2.1 Pattern Language Summary

Tropyc offers a set of ten closely related patterns and supports the decision making process of choosing which cryptographic services address application requirements and user needs.

When on-line communication or exchange of information through files is used, sometimes, due to great sensitiveness of data, it is necessary to guarantee its (*Information*) *Secrecy*. However, secrecy alone does not prevent either data modification or replacement. Particularly in on-line communication, granting *Message Integrity* and (*Sender*) *Authentication* is also important. There are some contexts in which it is necessary to prevent an entity from denying her actions or commitments. For example, some form of *Signature* is necessary when purchasing electronic goods over the Internet. Sometimes, the cryptography-based security requirements of applications lead to compositions of services in order to provide *Secrecy with Integrity*, *Secrecy with Sender Authentication* or *Secrecy with Signature*. Cryptography is time consuming, so algorithm performance is always important. *Signature* can be speeded up if a *Signature with Appendix* is used. For efficiency, *Secrecy with Signature with Appendix* is usually implemented instead of *Secrecy with Signature*. All the above situations share some aspects of structure and behavior, which can be abstracted as a *Generic Object-Oriented Cryptographic Architecture* (GOOCA).

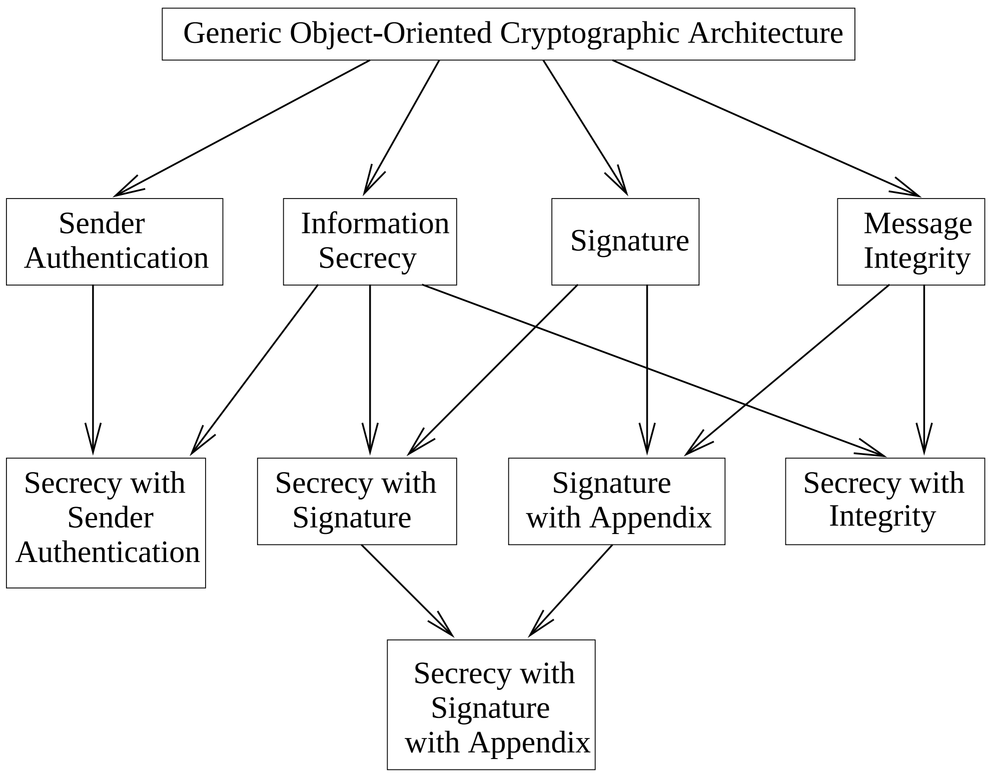

Figure 1: Cryptographic Design Patterns and Their Relationships.

Figure 1 is a directed acyclic graph of dependences among patterns. An edge from pattern A to pattern B means pattern B is generated from pattern A. GOOCA generates the micro-architecture for the four basic patterns. All other patterns are combinations of these. Thus, all nine cryptographic patterns instantiate GOOCA. A walk on the graph is directed by two questions: What cryptographic services should be used to address application requirements and user needs? And how should the cryptographic component be structured to obtain easy reuse and flexibility?

2.2 Generic Object-Oriented Cryptographic Architecture

Context Two objects, Alice and Bob, exchange data through messages. They need to perform cryptographic transformations, alone or in combinations, on the data. They want a cryptography component that is flexible and that could be easily reused with other cryptographic transformations.

Problem How to design a flexible object-oriented micro-architecture for a cryptographic design in order to facilitate component reuse?

Applicability

- When a separation of concerns between cryptographic non-functional requirements and application functionality should be enforced.
- When many cryptographic transformations must be accomplished and the application should be independent of what transformation is performed.
- When a generic interface to several kinds of cryptographic services is necessary.

Forces

- The dependencies between cryptographic features and application code should be minimized in order to facilitate reuse.
- The readability of programs with cryptographic code should be increased.
- The performance of cryptographic transformation algorithms should be preserved.

Solution Alice performs a cryptographic transformation on the data before sending it to Bob. Bob receives the message and performs the reverse transformation to recover the data. Alice and Bob must agree on what transformation to perform and share or distribute keys, if necessary. The class diagram shown in Figure 2 generalizes the cryptographic transformation into an abstract interface and distinguishes the Sender and Receiver roles from the Codifier and Decodifier roles. GOOCA is a higher level abstraction for all other cryptographic patterns. All cryptographic patterns instantiate that structure and dynamics. The GOOCA has two template classes, Alice and Bob, and two hook classes, Codifier and Decodifier, as shown in Figure 2. The class Codifier has a hook method $f()$, which performs a cryptographic transformation over x. The class Decodifier has a hook method $g()$, which performs the reverse transformation, $x = g(f(x))$. The transformation and its reverse are based on the same cryptographic algorithm. Figure 3 shows GOOCA dynamic behavior.

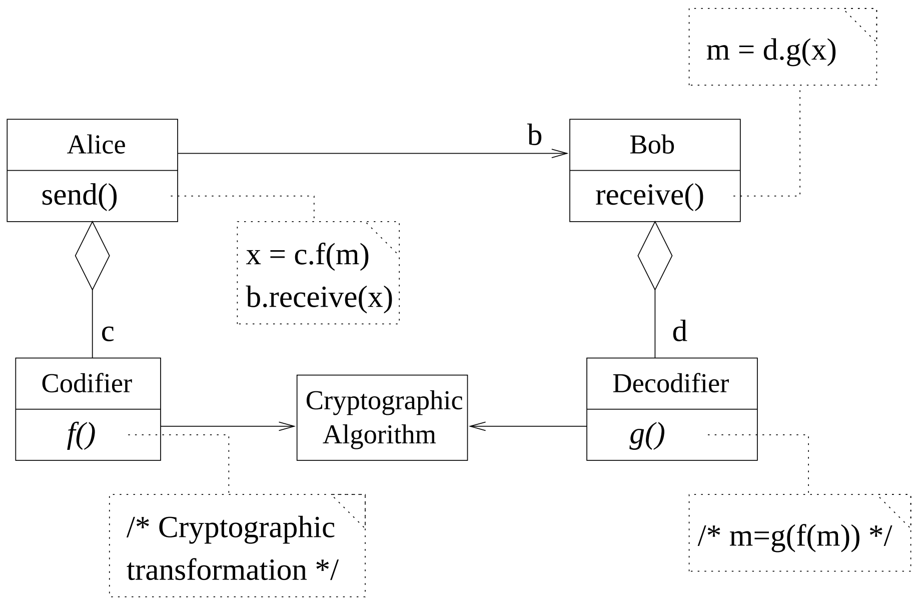

Figure 2: GOOCA Structure.

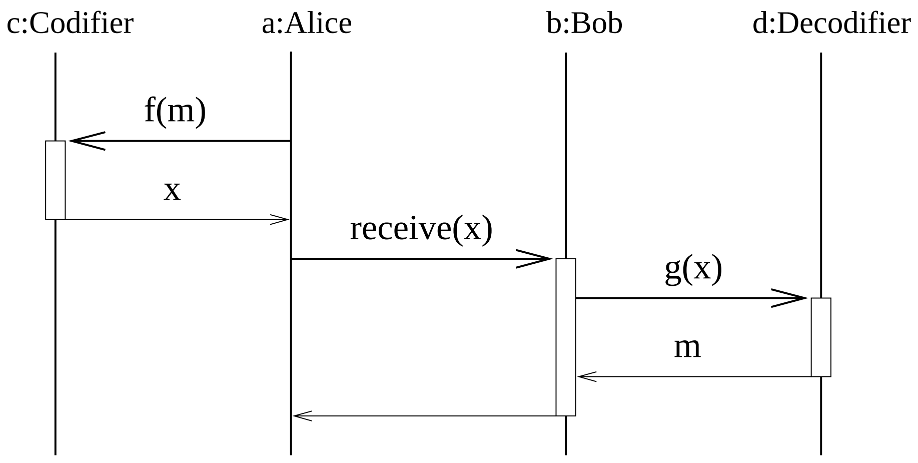

Figure 3: GOOCA Dynamics.

Consequences

- All cryptographic transformations will present a common behavior that could be generalized into a flexible design.
- *Ad hoc* implementations of cryptographic facilities could have a better performance than GOOCA instantiations, which are characterized by insertion of indirections into the code.
- Flexible and adaptable systems with cryptography-based security requirements can be more easily obtained when cryptographic algorithms are decoupled from their implementations, and these two are, in turn, decoupled from the cryptographic services those systems use.
- Cryptography-based security requirements are usually non-functional requirements of general purpose applications and should not pollute application functionality. The explicit separation of concerns in two kinds of requirements facilitates readability and reuse.

Implementation Factors

- This pattern can be easily adapted to deal with file storage and recovery. In such a situation, the send and receive messages could be replaced by store and recovery ones, respectively. Also, the reference Alice has to Bob can be avoided.
- Computational reflection can be used to make explicit the distinction of concerns and to restrict cryptographic code to a meta-level, responsible for interception of communication or storage requests and encryption of intercepted data.
- Before secure communication begins, a previous negotiation step is necessary in order for the parties to agree on which transformation will be performed, and also to exchange information such as keys and algorithm parameters.
- Eve's role depends on the instantiation. Basically, she can replace, modify or insert her own messages into the communication channel.

Example An electronic payment system, due to its strong security requirements, is better designed as an instantiation of GOOCA. Such an application, called *PayPerClick*, is designed and implemented in sections 3 and 5, respectively.

Known Uses All the cryptographic patterns described in Section 4, and widely used in systems as [HY97, Her97, CGHK98, HN98, BDR98], are instantiations of GOOCA.

Related Patterns Some well known patterns can be used when instantiating GOOCA. The *Strategy* [GHJV94, 315] pattern can be used to obtain algorithm independence. The *Bridge* [GHJV94, 151] pattern can be used to obtain implementation independence. The *Abstract Factory* [GHJV94, 87] pattern can be used in the previous negotiation step to decide which algorithm or implementation to use. The *Observer* [GHJV94, 293], *Proxy* [GHJV94, 207], and *Client-Dispatcher-Server* [BMR$^+$96, 323] patterns can be used to obtain location transparency. The *Forwarder-Receiver* [BMR$^+$96, 307] pattern could be combined with the cryptographic patterns in order to offer secure and transparent interprocess communication, in such a way that Alice becomes part of the Forwarder and Bob is incorporated into the Receiver. The *State* [GHJV94, 305] pattern could also be used to provide state dependent behavior, such as turning the security of the channel on and off. The *NullObject* [MRBV97, 5] pattern can be used to design a null transformation. The *Reflection* pattern [BMR$^+$96, 193] can be used to decouple application functionality and cryptography. Lower level security aspects, such as cryptography, can be encapsulated into a *Security Access Layer* [YB97, 16].

3 *PayPerClick*: An Electronic Payment System

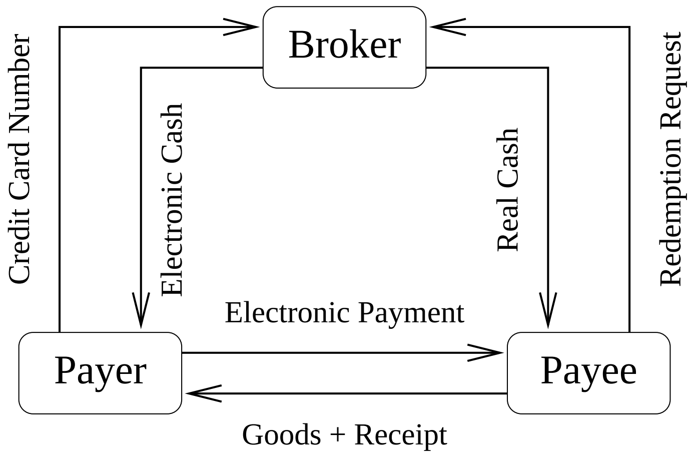

Figure 4: Electronic Commerce Participants.

In order to illustrate the applicability of this pattern language, we will use its cryptographic patterns over a large example, the design of an electronic commerce application. We will consider only the cryptography-based security aspects of such applications. Other aspects, such as distribution and fault tolerance, will not be considered. Patterns of Section 4 address particular aspects of this example. Figure 4 shows the three main entities of electronic commerce applications and the flow of money and sensitive data among them [AJSW97]. The Broker is usually a bank or a financial institution like Visa or MasterCard. The Payee can be an Internet access provider and Payers are the customers. Payers make resquests to Brokers for electronic cash, which can be debited in Payers' credit cards.

Payers can use their electronic cash to buy (electronic) goods over the Internet. They can request a receipt, which is emitted by Payees. Payees request redemption to Brokers, who redeem Payees for sale of goods to Brokers' clients. Usually, real money flows only from Brokers to Payees. Two good sources of information about electronic payment systems are [FD98, HY97].

Our application is a tool for electronic purchase and on-line distribution of hypertext documents based on the model of Figure 4. Hypertext documents are accessible through links to HTML pages and visualized through web browsers. This kind of application is useful for sale of on-line books through links in their table of contents. Customers interested in a specific book section can buy only the necessary information clicking on a hyper-link. We call this tool *PayPerClick* [BDR98]. The *Composite* [GHJV94, 163] pattern can be used to compute document value and fingerprint. The fingerprint is an MDC (see Appendix A.1) of a specific tree of hyper-documents using a specific traversal policy. The Payer part of the application can be a Java applet or a Netscape plug-in, which communicates with its web server (Payee or Broker). Payers usually have an electronic wallet. Brokers emit electronic cash which can be divided in a fixed amount of electronic coins, which are used in payments. Figures 5 and 6 show how to model a *PayPerClick* payment transaction as an instantiation of GOOCA. Two patterns are used in these diagrams, *Sender authentication* and *Signature.* The first is used to authenticate the payment; the second, to sign the receipt.

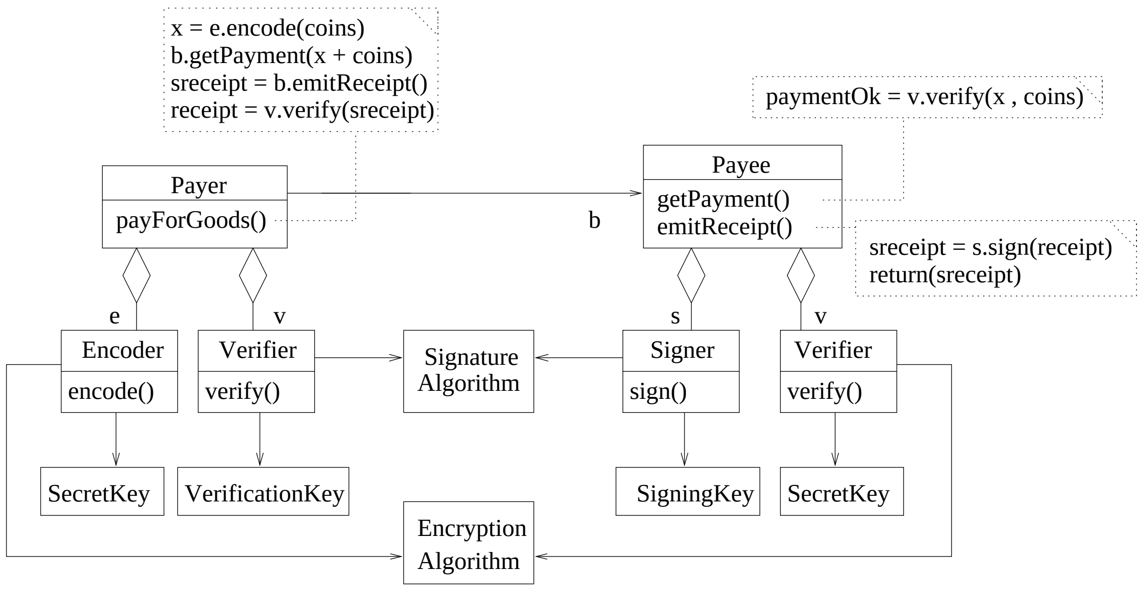

Figure 5: Payment Structure.

4 Cryptographic Service Patterns

This Section contains patterns related to the cryptographic services supporting the cryptographic objectives [MvOV96] (summarized in Appendix A).

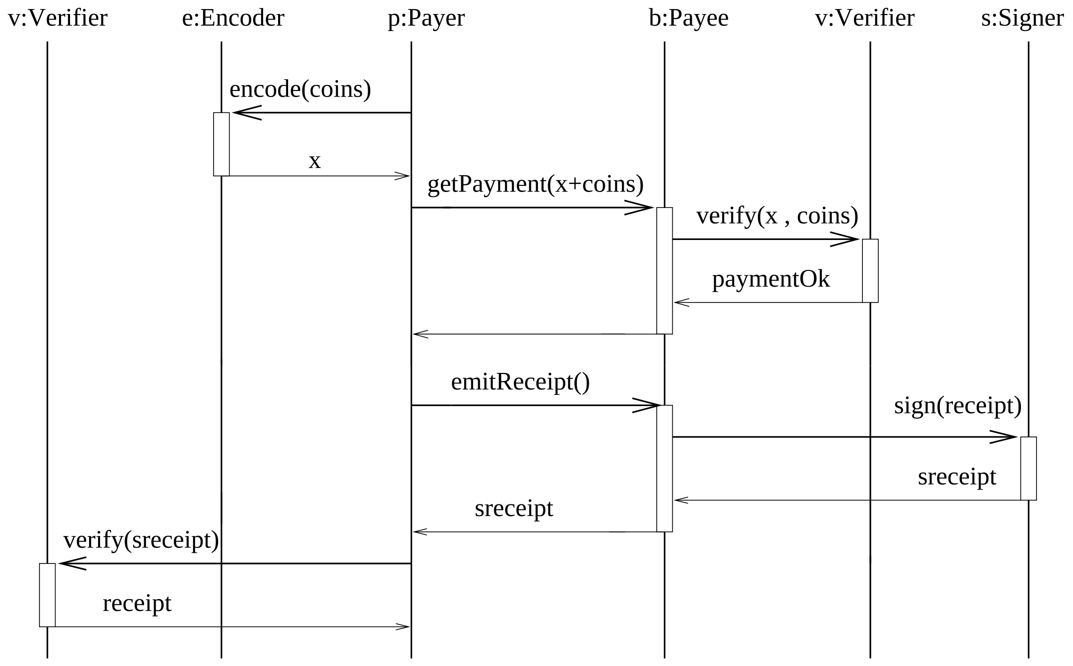

Figure 6: Payment Dynamics.

4.1 Information Secrecy

Context Alice wants to send sensitive messages to Bob. She wants to keep these messages secret from Eve, whom Alice suspects may be trying to read her messages.

Problem How can Alice send a message to Bob in such a way that Eve cannot possibly read its content?

Applicability

- When two entities need to share confidential information.
- When is necessary to decouple encryption from communication or storage.

Forces

- The cost of encryption must not be greater than the intrinsic value of the message being encrypted.
- Eve cannot, in any situation, gain access to the message contents. This is the main requirement of this pattern.
- The cost Eve has to pay to break a message must be much greater than the value attributed to that message.

Solution This pattern supports encryption and decryption of data. Alice and Bob previously agree on a cryptographic engine and on a shared secret key (if public key cryptography is used, Bob must first obtain Alice's public key). Bob enciphers the message and sends it to Alice. Alice deciphers the enciphered message and recovers the original message.

Consequences

- Encryption is slow. The security of an encrypted information is in the encryption key and the strength of a cryptosystem is based on the secrecy of a long key. However, the longer the key, the slower is the algorithm.
- Eve cannot read the message contents, but she can always replace or modify the encrypted message or perform some other kind of attack.
- Errors of transmission or in the storage device can potentially make the recovery of original message impossible.

Implementation Factors

- Private or secret keys must be kept protected from unauthorized copy or modification.
- An infrastructure to distribute or make public keys broadly available is necessary.
- Some sort of error detection is usually required in order to avoid message losses.
- A choice must be made of the cryptosystem to be used (symmetric secret key, asymmetric public key or hybrid).

Example When a Payer makes a cash request in *PayPerClick*, he sends the Broker his credit card number encrypted. Payer encrypts his card number with his encryption key and sends it to Broker within a cash request. Broker decrypts card number with his decryption key and use it to debit the cash amount requested.

Known Uses A common use of this pattern is the encryption of electronic mail [Her97].

4.2 Message Integrity

Context Alice sends large messages to Bob. He wants to verify the integrity of the received messages, which Eve may have modified or replaced. Alice and Bob do not share cryptographic keys, so they cannot sign messages or authenticate them with Message Authentication Codes (MACs) (see Section A.1).

Problem How can Bob determine whether a message he received has been modified?

Applicability

- When the occurrence of errors of transmission or in storage must be detected.
- When detection of unauthorized modification of data is necessary.
- When generating unique identifiers for messages or data records.

Forces

- The mechanism for integrity checking should not allow Eve to easily produce a valid but fake message that would stand the verification procedure.
- The mechanism should be robust against small accidental modification of data.
- The mechanism should be cheap to use.

Solution Alice and Bob agree to use a Modification Detection Code (MDC) (see Section A.1). Alice computes the MDC of the message and sends both message and MDC to Bob. Bob computes the MDC of the message and compares it with that received from Alice. If they match, the message is genuine.

Consequences

- It is necessary to verify a relatively small MDC to determine whether a large amount of data has been modified.
- Eve still has the ability of substituting both the message and the corresponding MDC.
- MDCs by themselves do not guarantee the authorship of a message.

Implementation Factors

- A message must be bound to its corresponding MDC in such a way that Bob could not be confused during verification by trying to match a wrong pair. In communication, messages and MDCs could be sent to Bob separately (in such a case some sort of synchronization is necessary) or the MDC could be attached to the message. In storage, it is common to keep the MDC and corresponding data separate.
- Integrity of messages can be obtained by sending each message at least twice and comparing the copies. Furthermore, sending short messages twice may be less expensive than computing and sending a relatively large MDC.

Example In *PayPerClick*, electronic payments must have their integrity preserved in order for the Payee verification to succeed; so, at least one MDC should be computed for each payment. Payer generates the payment package of coins, compute the MDC of it and sends both payment and MDC to Payee, which verifies its correctness.

Known Uses Two common uses of MDCs are detection of file modification caused by viruses and generation of pass-phrases to produce cryptographic keys. Also, MDCs could be used as unique identifiers of electronic coins in electronic commerce applications [FD98].

4.3 Sender Authentication

Context Alice and Bob want to exchange messages, but they cannot distinguish their own messages from the ones Eve may have included into the communication channel. Also, they have the ability to share secrets in a secure way.

Problem How can genuine messages be distinguished from spurious ones?

Applicability

- When the occurrence of errors during transmission or storage must be detected.
- When detection of corruption or unauthorized modification of data is necessary.
- When it is necessary for Alice and Bob to certify themselves of the origin of exchanged messages

Forces

- It must be hard for Eve to forge authentic messages.
- The authentication mechanism should detect accidental data modification.
- The mechanism should be cheap to use; that is, cheaper than the intrinsic value of the data being authenticated.

Solution Alice and Bob agree previously on a shared secret key and a cryptographic algorithm to generate Message Authentication Codes — MACs (see Appendix A.1). Alice computes the MAC of the message and sends both, message and MAC, to Bob. Bob computes the MAC again and compares it with the one received from Alice. If they match, the message is genuine and must have been sent by Alice, because, other than Bob, only Alice knows the secret key and can compute the correct MAC.

Consequences

- A message can only be authenticated if an information recognized only by the communicating parts is intrinsically associated with it. If the authentication information is a function of both data and a secret, the replacement of messages or parts of them can be detected.
- Authorship cannot be proved to a third party, since both sides can compute valid MACs.

Implementation Factors

- A secure means to exchange and keep a secret key is necessary, since the security of a MAC is in the secrecy of the key.
- As with MDCs, a message must be bound to its corresponding MAC.
- MACs can be implemented in many ways. Two common possibilities are symmetric cryptosystems and cryptographic hash functions.

Example Eve could have the ability of substitute her coins for someone else's. MACs can be used to detect payment substitutions. The Payer generates a payment package of coins, computes a MAC of it and sends both payment and MAC to Payee, who verifies its correctness and authenticity. Such a situation works well for the Payee, who always receives valid payments. However, a Payer can always repudiate old payments and still request redemption to Broker. If the value of payments is relatively large and coin losses are frequent, *Signature* could be a better solution to this problem.

Known Uses MACs can be used to authenticate IP packages over the Internet [CGHK98].

4.4 Signature

Context Alice sends messages to Bob, but they cannot distinguish their own messages from the ones Eve may insert into the communication channel. Furthermore, Alice can later dispute the authorship of messages actually sent by her. In such a situation, Bob cannot prove to a third party that only Alice could have sent that message. Also, Alice has a public/private key pair and her public key is widely available.

Problem How to guarantee that messages have a genuine and authentic sender in such a way that Alice cannot repudiate a message sent by her to Bob?

Applicability

- When non-repudiation of messages must be guaranteed.

Forces

- Signatures must be dependent from the data being signed. Otherwise, they could easily be copied and tied to a different message. As a consequence, signatures must also guarantee data integrity.
- Signatures must be hard to forge or alter.
- The cost of signing must be substantially cheaper than the data being signed.
- It must be possible to verify the authenticity of a signature independently of its author.

Solution Alice and Bob agree on the use of a public key digital signature protocol (see Appendix A.1). Alice enciphers a message with her private key and sends the signed message to Bob. He deciphers the signed message with Alice's verification key. If the encrypted message makes sense to Bob, then, since only Alice's signing key could have been used to generate a meaningful message after decryption by Bob, it must be true that Alice is the sender of that message.

Consequences

- Signatures are usually as large as the data being signed, sometimes producing an intolerable overhead.
- Verifying the message authorship is based solely on the secrecy of the author's key.

Implementation Factors

- Public key cryptographic algorithms are generally used to generate digital signatures.
- A secure means of storing the author's private key is necessary.
- An infrastructure to distribute or make public keys broadly available is necessary.

Example This pattern is used in *PayPerClick* in two situations of non-repudiation of origin: cash emitting by the Broker and receipt emitting by the Payee. In the first, Broker produces a cash amount, signs it and sends this signed cash to Payer, which verifies the cash authenticity. In the second, a Payee verifies if a single coin was emitted by a specific Broker.

Known Uses Digital signatures are used in electronic commerce applications in the authentication of customers and merchants [FD98]. Also, they can be used to guarantee authenticity and non-repudiation of information obtained over the Internet [HN98].

4.5 Signature with Appendix

Context Alice and Bob sign messages they exchange in order to prevent modifications or replacement and to provide *Signatures.* However, they have limited storage and processing resources and the messages they exchange are very large and produce large signatures.

Problem How to reduce the storage space required for a message and its signature while increasing the performance of the digital signature protocol?

Applicability

- When a message can be separated from its signature.
- When the digital signature protocol requires an improvement of performance.

Forces Similar to those of the *Signature* pattern.

Solution Two patterns are combined to solve this problem: *Signature* and *Message Integrity.* This pattern implements a digital signature protocol over a message hash value, which is an MDC. Alice computes a hash value of the message and signs it. Both message and signed hash value are sent to Bob. Bob decrypts the signature and recovers the hash value. He then computes a new hash value and compares it with the one recovered from the signature. If they match, the signature is true.

Consequences

- When no technique to reduce signature size is used, digital signatures are usually as large as the data being signed. However, if messages are small, the inclusion of a new computation step to reduce the signature size is not necessary. An important trade-off is the relation between the impact of additional computations over performance and the ratio between the size of messages and their signatures.

- The combination of weak MDCs with *Signature* can potentially decrease the security of digital signature protocols.

Example Non-repudiation of receipt is obtained by signing a payment receipt. In *PayPerClick*, including a raw digital signature into the receipt, for each node in the hyper-document's tree, could not be practical, due to a large number of large documents. Even if a *Signature with Appendix* is computed for each tree node, a large number of nodes can still result in a large receipt. On the other hand, signing a single fingerprint of a hyper-document's tree is a *Signature with Appendix* of that tree. *PayPerClick* binds a receipt to its corresponding purchased hyper-documents, including a fingerprint of that hyper-document's tree into the receipt being signed.

Known Uses When a user of an Internet application must digitally sign information, for efficiency reasons, it is better to produce small signatures [CGHK98]. Signed applets are a means by which Java code can be authenticated. JDK uses *Signature with Appendix* to produce small signatures for large amounts of code [Knu98].

4.6 Secrecy with Integrity

Context Alice and Bob exchange encrypted messages, but they cannot detect modification or replacement of encrypted messages by Eve. Also, they do not want to share a secret key for authentication purposes alone.

Problem How to preserve the integrity of an encrypted message without loss of secrecy?

Applicability

- When the occurrence of errors during secret transmission or storage must be detected.
- When detection of unauthorized modification of secrets is necessary.

Forces

- It is desirable that integrity may be verified without disclosure of the secret.
- Granting secrecy and data integrity at the same time should not happen at the expense of one or the other. That is, applying mechanisms for integrity and secrecy separately, should be at least as effective as applying a method which simply composes both features. For instance, it should not be any easier to decipher a message in the presence of some MDC than it is without it.

Solution Two previous patterns are combined to solve this problem: *Information Secrecy* and *Message Integrity.* The MDC can be computed over the original not encrypted message. Both encrypted message and MDC are sent to Bob. This pattern only requires one public/private key pair (or a shared secret key) for encryption purposes.

Consequences

- Malicious replacements of messages can still cause valid data to become completely garbled, or change its meaning, after decryption.
- The computation and verification of MDCs cause a decrease of performance.

Implementation Factors There are two ways of implementing this pattern: computing MDC over encrypted message or computing MDC over message before encryption. In the first, transmission errors can be detected before decryption. In the second, if the message structure is unknown, small transmission errors can only be detected after decryption and MDC verification.

Example If the encrypted card number arrives corrupted at Broker, maybe due to transmission errors, it will not be decrypted successfully. Broker should have the ability to detect encrypted message corruption which may cause it to use a wrong but valid card number. During a *PayPerClick* cash request, Payer should compute an MDC of the card number, encrypt the number and send both MDC and encrypted number to Broker, who decrypts the number and verifies its correctness.

4.7 Secrecy with Sender Authentication

Context Alice and Bob use public key cryptography to exchange encrypted messages. Eve may intercept messages, but she cannot read their contents. However, she can replace or modify these messages in such a way that Alice and Bob cannot detect these modifications or replacements.

Problem How can Alice authenticate the origin of an encrypted message without loss of secrecy?

Applicability

- When the occurrence of errors during secret transmission or storage must be detected.
- When detection of unauthorized modification of secrets is necessary.
- When detection of encrypted message replacement is necessary.

Forces Similar to the previous composition pattern.

Solution Two previous cryptographic patterns are combined to solve this problem: *Information Secrecy* and *Sender Authentication.* The MAC should be computed over the original, not the encrypted message. Both encrypted message and MAC are sent to Bob. The secret key used to compute the MAC must be different from that used for encryption.

Consequences

- *Sender Authentication* restricts the number of entities who can produce genuine encrypted messages, but do not grant authorship.
- *Sender Authentication* inserts a new step in both the encryption and decryption processes in order to compute and verify a MAC, which can result in a decrease of performance.

Implementation Factors

- If Alice and Bob use secret key cryptography for encryption and they are the only ones who share that secret, *Sender Authentication* is redundant and useless, except for granting an extra degree of security.
- Again, there are two possible implementations: to compute authentication before encryption or to authenticate the encrypted message.

Example In a *PayPerClick* cash request, if secret key cryptography is used for encryption of the card number and the secret key is shared only by a pair Payer/Broker, a simple form of sender authentication is obtained. The Payer can use a MAC to provide sender authentication and the secret key used in MAC computations must be different from that one used for encryption purposes.

Known Uses Secrecy and authentication can be combined in order to secure IP packages over the Internet[CGHK98].

4.8 Secrecy with Signature

Context Alice and Bob exchange encrypted messages, but they cannot prove the authorship of an encrypted message. Furthermore, Eve can modify, replace or include messages into the communication channel in such a way that Alice and Bob cannot detect the spurious messages. Alice and Bob already share keys for secrecy purposes.

Problem How can Bob prove the authorship of an encrypted message without loss of secrecy, in such a way that its integrity and origin authentication are also implicitly guaranteed?

Applicability

- When non-repudiation of a secret must be guaranteed.

Forces Similar to the two previous composition patterns.

Solution Two previous cryptographic patterns are combined to solve this problem: *Information Secrecy* and *Signature*. Alice signs a message with her signing key, encrypts the signed message with Bob's encryption key and sends it to Bob. Bob deciphers the encrypted message with his decryption key and verifies the signed message with Alice's verification key.

Consequences

- *Signature* can grant authorship of encrypted messages by inserting an intermediate step into the encryption/decryption process, causing a loss of efficiency.

Implementation Factors

- It is recommended that different keys or key pairs be used for encryption and signing.

- Again, this pattern can be implemented in two different ways according to the order of operations. Since the signed message is the signature itself, detection of modification can only be done with both decryption and verification. In order to prevent this situation, encrypted messages can be concatenated to a known header before signing. This strategy does not work with encryption of a signed message plus a known header, when the encryption mode is ECB. A better strategy is the use of *Secrecy with Signature with Appendix*. Information about block ciphers and operation modes can be found in [Sch96].

Example When sending his/her credit card number over the Internet, a user requires its secrecy. On the other hand, a vendor upon receiving such a number, needs the assurance of non-repudiation by the sender in the future. In a *PayPerClick* cash request, if secret key cryptography is used to encrypt the card number and the secret key is shared only by a pair Payer/Broker, a simple and limited form of sender authentication can be obtained implicitly.

4.9 Secrecy with Signature with Appendix

Context Alice and Bob exchange encrypted signed messages to prevent modification or replacement and to achieve secrecy and non-repudiation. They need to manage limited storage and processing resources. However, the messages they exchange are very large and produce large signatures.

Problem How to reduce the amount of memory necessary to store a message and its signature, while increasing system performance, without loss of secrecy?

Applicability

- When a secret must be separated from its signature.
- When the digital signature protocol requires an improvement of performance.

Forces Similar to previous pattern combinations.

Solution Two patterns are combined to solve this problem: *Information Secrecy* and *Signature with Appendix*. Alice computes a hash value of the message and signs it with her signing key. She then encrypts the original message with Bob's encryption key. Both encrypted message and signed hash value are sent to Bob. He deciphers the encrypted message with his decryption key and decrypts the signed hash value with Alice's verification key. Bob computes a new hash value of the message and compares it with that received from Alice. If they match, the signature is true.

Consequences The inclusion of a new processing step to reduce the signature size within a computation that already has two processing phases, one to encrypt/decrypt data and another to compute or verify a signature, is a difficult decision. Again small losses of performance must be negligible in order to save a relatively large amount of space and reduce the amount of data to be transmitted.

Implementation Factors This pattern can be implemented in two ways: signing of message MDC before message encryption, or signing of an encrypted message's MDC.

Example Electronic forms usually contain a lot of sensitive information which requires secrecy and non-repudiation. A typical cash request form could have fields for credit card information such as number, expiration date, card type and owner; other fields may contain the amount of cash requested, value of coins, etc. The use of digital signatures to guarantee non-repudiation of that sensitive data, which should also be encrypted, can potentially result in large signatures. *Secrecy with Signature with Appendix* applied over all form fields can solve the above problem with a substantial performance improvement.

5 Programming Cryptographic Patterns

The Java code in this Section corresponds to a *PayPerClick* transaction and uses Java Cryptographic Architecture (JCA), available since JDK 1.1. The classes `SecureRandom`, `Signature`, `SignedObject`, `PublicKey` and `PrivateKey` are part of JCA. Classes `Cipher`, `SealedObject` and `SecretKey` belong to Java Cryptographic Extension (JCE). Information about the Java cryptographic API can be found in [Knu98]. Classes `Signer` and `Verifier` perform digital signing and verification, respectively. A `Signer` should be initialized with a `PrivateKey` and a `Signature` engine. A `Verifier` takes a `Signature` engine and a `PublicKey`. Method `Signer.sign()` returns a `SignedObject` containing the object being signed and its digital signature. Method `Verifier.verify()` takes a `SignedObject` and returns true if the verification succeeds.

```
class Signer{
    private Signature engine;
    private PrivateKey key;

    public Signer(Signature engine, PrivateKey key){
        this.key = key; this.engine = engine;}

    public SignedObject sign(Serializable o){ return(new SignedObject(o,key,engine));}}

class Verifier{
    private Signature engine;
    private PublicKey key;

    public Verifier(Signature engine, PublicKey key){
        this.key = key; this.engine = engine;}

    public boolean verify(SignedObject o){ return(o.verify(key,engine));}}
```

The `Payer` class contains a `Signer` to sign payments, an `Encipher`, used to encrypt credit card numbers, and two instances of `Verifier`, one used to verify payment receipts emitted by the `Payee`, another to verify cash emitted by the `Broker`. A `Vector` is used as a simple electronic wallet. `Payer` has two methods, `payForGoods()`, which emits a payment to `Payee` and requests a signed receipt, and `cashRequest()`, which is used to ask `Broker` for money. Method `payForGoods()` emits a `SignedObject` containing a payment of `price` coins to a `Payee`. The required amount of coins is removed from the wallet, the payment is signed and sent to `Payee`, from whom a receipt is requested. A payment transaction succeeds if the payment succeeds and the receipt is authentic. Method `cashRequest()` asks `Broker` for an `amount` of electronic money, which should be debited from `Payer` credit card. The number of the `Payer` card is sent to `Broker` into a `SealedObject`. A `SignedObject` containing cash is received, verified and used to fill the `Payer`'s wallet, if the previous verification succeeded.

```
class Payer{
    private Verifier vPayee, vBroker;
    private Encipher e;
    private Signer s;

    private SignedObject receipt;
    private String myCardNumber = "0001 0002 0003 0004";
    private Vector wallet;

    public Payer(Signer s, Encipher e, Verifier vPayee, Verifier vBroker){
        this.s = s; this.e = e; this.vPayee = vPayee; this.vBroker = vBroker;}

    public boolean payForGoods(Payee p, int price){
        boolean ok = true;
        Vector payment = new Vector();
        for(int i = 0; i < price; i++) {
            Object coin = wallet.firstElement();
            payment.addElement(coin);
            wallet.removeElement(coin);
        }
        ok &= p.getPayment(s.sign((Serializable) payment));
        receipt = p.emitReceipt();
        ok &= vPayee.verify(receipt);
        if (ok) System.out.println(receipt.getObject());
        return(ok);
    }

    public boolean cashRequest(Broker b, int amount){
        boolean ok;
        ok = b.getCreditCard(e.encrypt(myCardNumber));
        SignedObject o = b.emitCash(amount);
        ok &= vBroker.verify(o);
        if(ok) wallet = (Vector) o.getObject();
        return(ok);
    }}
```

Class `Payee` contains a `Signer` to sign receipts and two instances of `Verifier`. One to verify payments signed by `Payer`, and another to verify single coins, emitted and signed by `Broker`, contained in payment. `Payee` has two methods, `emitReceipt()`, which emits a signed receipt of payment, and `getPayment()`, used to verify and account for payments and coins. Method `emitReceipt()` returns a `SignedObject`, which contains the number of valid coins received since the last receipt was emitted. This implementation does not consider the purchased goods for which this receipt is being emitted. A better solution should contain the fingerprint of the purchased document. Method `getPayment()` takes a `SignedObject` and verifies if it is a valid payment with valid coins in it. Payments are verified with the `Payer` verifier; coins are verified with the `Broker` verifier. The method returns true if all verifications succeed.

```
class Payee{
    private Signer s;
    private Integer coinCounter = new Integer(0);
    private Verifier vBroker, vPayer;

    public Payee(Signer s, Verifier vBroker,Verifier vPayer ){
        this.s = s; this.vBroker = vBroker; this.vPayer = vPayer;}

    public SignedObject emitReceipt(){
        String str = (coinCounter.intValue()≠1?"s":"");
        String receipt = "I received " + coinCounter.toString() + " coin"+str+
                         " from You.  Since last receipt was emitted.";
        this.coinCounter = new Integer(0);
        return(s.sign(receipt));
    }

    public boolean getPayment(SignedObject payment){
        boolean ok;
        int counter = coinCounter.intValue();
        ok = vPayer.verify(payment);
        Vector coins = (Vector) payment.getObject();
        for(int i = 0; i < coins.size();i++) {
            ok &= vBroker.verify((SignedObject) coins.elementAt(i));
            if(ok) this.coinCounter = new Integer(++counter);
        }
        return(ok);
    }}
```

The class `Broker` contains a `Decipher`, used to decrypt Payer's card number, and a `Signer` to authenticate cash he emits. `Broker` has two methods: `getCreditCard()`, which receives a `SealedObejct` containing Payer's encrypted card number, and `emitCash()`, used to generate an `amount` of coins. In `emitCash()`, an amount of cash is a `Vector` in which each coin is a `SignedObject` containing a random value. Cash amount is also contained in a `SignedObject`.

```
class Broker{
    private Decipher d;
    private Signer s;

    public Broker(Decipher d, Signer s){ this.d = d; this.s = s;}

    public boolean getCreditCard(SealedObject o){
        System.out.println("Card Number is "+d.decrypt(o));return(true);}

    public SignedObject emitCash(int amount){
        Vector cash = new Vector(amount);
        SecureRandom sr = new SecureRandom();
        byte[] random = new byte[20];
        sr.nextBytes(random);
        for(int i = 0; i<amount; i++) cash.addElement(s.sign(new String(random)));
        return(s.sign(cash));
}}
```

6 Conclusions and Future Work

Cryptographic support is becoming a default feature in many applications. In order to facilitate the design, implementation and reuse of flexible and adaptable cryptographic software, the architectural aspects of cryptographic components and the patterns that emerge from them should be considered. In this work, we present a pattern language for cryptographic software. We consider our pattern language to be complete and closed into the cryptographic services domain for two reasons: (*i*) The nine patterns, except GOOCA, represent all valid combinations of four well stablished cryptographic objectives. (*ii*) The cryptographic patterns are widely used in many applications [HY97, Her97, CGHK98, HN98] and supported by many cryptographic APIs [Knu98, Kal95, CSS97]. However, other auxiliary patterns and pattern languages supporting infraestructure services for cryptosystems could be possible. *Tropyc* documents the current usage of cryptographic techniques and the experience of cryptographic software practitioners. Therefore, it can be used to guide the decision making process for the design of cryptographic features.

7 Acknowledgments

This work is partially supported by CNPq; PRONEX-Finep, grant 107/97 for Ricardo Dahab and FAPESP, grants 97/11128-3 for Alexandre Melo Braga and 96/15329 for LSD-IC-UNICAMP. We would like to thank our shepherd Robert Orenstein and the workshop session members at the PLoP'98 conference, whose suggestions and comments were very important during the writing of the present version of this paper.

A Basic Cryptographic Concepts

Historically associated to encryption, modern cryptography is a broader subject, encompassing the study and use of mathematical techniques to address information security problems, such as confidentiality, data integrity and non-repudiation. Usually, four objectives of cryptography are considered [MvOV96]: confidentiality, integrity, authentication, and non-repudiation. Accordingly, there are four basic cryptographic services: (*i*) encryption/decryption to obtain secrecy or privacy, (*ii*) MDC (Modification Detection Code) generation/verification, (*iii*) MAC (Message Authentication Code) generation/verification, and (*iv*) digital signing/verification. These four services can be combined in specific and limited ways to produce more specialized services.

Confidentiality is the ability to keep information secret except from authorized users. **Data integrity** is used to guarantee information has not been modified without permission, which includes the ability to detect unauthorized manipulation. **Sender (origin) authentication** corresponds to the assurance by the communicating parties, of the origin of an information transmitted through an insecure communication channel. **Non-repudiation** is the ability to prevent an entity from denying its actions or commitments in the future.

A.1 Cryptographic Services

Secret or symmetric key cryptography is the set of cryptographic techniques in which a single key is used both to encrypt and decrypt data. The key is a shared secret between two or more entities. In public key cryptography, a pair of different keys is used, one key for encryption, another for decryption. The encryption key is publicly known, so it is called the **public key**. The corresponding decryption key is a secret known only by the key pair owner, so it is called the **private key**. In public key cryptography, it is computationally infeasible to deduce the private key from the knowledge of the public key.

Traditionally, the two ends of a communication channel are called Alice and Bob. Eve is an adversary eavesdropping the channel. Alice wants to send an encrypted message to Bob; she encrypts message m, the plain text, with an encryption key $k1$ and sends the encrypted message c, the cipher text, to Bob, that is, $c = f(m, k1)$. Bob receives the encrypted message and deciphers it with a decryption key $k2$ to recover m, that is, $m = g(c, k2)$ and $g = f^{-1}$. If public key cryptography is used, Alice uses Bob's public key to encrypt messages and Bob uses his private key to decrypt messages sent from anyone who used his public key. However, if symmetric key cryptography is used, Alice and Bob share a secret key used to encrypt and decrypt messages they send to each other. That is, $k1 = k2$. Two good sources of information about cryptography are [MvOV96, Sch96].

A hash function is a mathematical function that takes as input a stream of variable length and returns as a result a stream of fixed length, usually much shorter than the input. One-way hash functions are hash functions in which it is computationally easy to compute the hash value of an input stream, but it is computationally difficult to determine any input stream corresponding to a known hash value. A cryptographic hash function is a one-way collision-resistant hash function; that is, it is computationally difficult to find two input streams that result in the same hash value. Hash values produced by cryptographic hash functions are also called Modification Detection Codes (MDCs for short) and are used to guarantee data integrity. Message Authentication Codes (MACs for short) are usually implemented as hash values generated by cryptographic hash functions which take as input a secret key as well as the usual input stream. MACs are used to provide data authentication and integrity implicitly.

Digital Signatures are electronic analogs of handwritten signatures, which serve as the signer's agreement to the information a document contains, and also as evidence that could be shown to a third party in case of repudiation. A basic protocol of digital signatures based on public key cryptography is: (a) Alice encrypts a message with her private key to sign it. (b) Alice sends the signed message to Bob. (c) Bob decrypts the received message with Alice's public key to verify the signature. Digital signatures must provide the following features: (i) They are authentic: when Bob verifies a message with Alice's public key, he knows she signed it; (ii) they are unforgeable: only Alice knows her private key; (iii) they are not reusable: the signature is a function of the data being signed, so it cannot be used with other data; (*iv*) they cannot be repudiated: Bob does not need Alice help in order to prove she signed a message; (*v*) the signed data is unalterable: any modification of the data invalidates the signature verification.

A.2 Common Attacks

In a brute-force attack, Eve tests all possible valid keys to decrypt a cipher text of a known plain text in order to find out the correct key. If Eve could obtain the private key of Alice or Bob (or their secret shared key), all the other attacks could be easily performed. Eve can attack a cryptosystem in four basic ways: (*i*) She can eavesdrop the channel. Eavesdropping an open channel is easy. However, in order to understand eavesdropped messages of a cryptographically secured channel, the key (or keys) being used by Alice and Bob are required. (*ii*) She can re-send old messages. This attack is possible if messages do not have temporal uniqueness, which can be obtained using timestamps or by changing keys periodically. (*iii*) She can impersonate one of the communicating ends of the channel. In such a case, Eve plays the role of Alice or Bob, either by deducing a secret key or by successfully substituting her public key for Alice's (Bob's) without Alice's (Bob's) knowledge. (*iv*) She can play the role of the man-in-the-middle. In order to perform the man-in-the-middle attack successfully, Eve must have obtained the private keys (or the secret shared key) of Alice and Bob, or impersonate both Alice and Bob. In such a situation, Eve can intercept encrypted messages from Alice (Bob) to Bob (Alice), decrypts them with Alice's (Bob's) decryption key and re-encrypt them with her own encryption key before re-sending.

A.3 Auxiliary Services

An important issue of cryptographic services is whether they are supported by an infrastructure which provides a strong and secure set of auxiliary services such as generation, agreement, distribution and storage of cryptographic keys. Usually, key generation algorithms are based on random number generators. Public keys are usually distributed together with their digital certificates, which are packages of information attesting the ownership and validity of a cryptographic key. These certificates are usually signed by a trusted third party, called a Certification Authority (CA). A private or secret key must be kept protected from unauthorized copy and modification; this can be done in two ways: (*i*) it can be stored in a tamper-proof hardware; (*ii*) it can be stored in an encrypted and authentic form in general purpose hardware, such as random access memories, magnetic disks and tapes. This requires a Key Encryption Key (KEK for short) which, in turn, must be protected.

References

[AJSW97] N. Asokan, Philippe A. Janson, Michael Steiner, and Michael Waidner. The State of the Art in Electronic Payment Systems. *IEEE Computer*, pages 28–35, September 1997.

[BDR98] Alexandre M. Braga, Ricardo Dahab, and Cecília M. F. Rubira. PayPerClick: Um Framework para Venda e Distribuição On-line de Publicações Baseado em Micropagamentos. In *SBRC'98 - 16o Simpósio Brasileiro de Redes de Computadores*, page 767, Rio de Janeiro, RJ, Brasil, May 1998. Extended summary.

[BMR$^+$96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns.* John Wiley and Sons Ltd., Chichester, UK, 1996.

[CGHK98] Pau-Chen Cheng, Juan A. Garay, Amir Herzberg, and Hugo Krawczyk. A Security Architecture for the Internet Protocol. *IBM Systems Journal*, 37(1):42–60, 1998.

[CSS97] Common Security Services Manager Application Programming Interface, Draft 2.0. http://www.opengroup.org/public/tech/security/pki/index.htm, June 1997.

[FD98] Lucas Ferreira and Ricardo Dahab. A Scheme for Analyzing Electronic Payment Systems. In *14th ACSAC - Annual Computer Security Applications Conference (ACSAC'98)*, Scottsdale, Arizona, December 1998.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Publishing Company, April 1994.

[Her97] Michael Herfert. Security-Enhanced Mailing Lists. *IEEE Network*, pages 30–33, 1997.

[HN98] Amir Herzberg and Dalit Naor. Surf'N'Sign: Client Signatures on Web Documents. *IBM Systems Journal*, 37(1):61–71, 1998.

[HY97] Amir Herzberg and Hilik Yochai. Minipay: Charging per Click on the Web. *Computer Networks and ISDN Systems*, 1997.

[Kal95] B. Kaliski. Cryptoki: A Cryptographic Token Interface, Version 1.0. http://www.rsa.com/rsalabs/pubs/PKCS/html/pkcs-11.html, April 1995.

[Knu98] Jonathan B. Knudsen. *Java Cryptography.* O'Reilly and Associates, 1998.

[MRBV97] Robert C. Martin, Dirk Riehle, Frank Buschmann, and John Vlissides, editors. *Pattern Languages of Program Design 3.* Addison-Wesley, 1997.

[MvOV96] Alfred J. Menezes, Paul C. van Orschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1996.

[Sch96] Bruce Schneier. *Applied Cryptography — Protocols, Algorithms , and Source Code in C.* John Wiley and Sons, 2nd edition, 1996.

[YB97] Joseph Yoder and Jeffrey Barcalow. Application Security. *PLoP'97 Conference, Washington University Technical Report 97-34*, 1997.