Análise Comparativa e Proposta de
Extensão à Arquitetura Criptográfica Java

Alexandre Melo Braga

Dissertação de Mestrado

Análise Comparativa e Proposta de Extensão à Arquitetura Criptográfica Java

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Alexandre Melo Braga e aprovada pela Banca Examinadora.

Campinas, 21 de Setembro de 1999.

Ricardo Dahab (Orientador)

Cecília M. F. Rubira (Co-orientadora)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Instituto de Computação
Universidade Estadual de Campinas

Análise Comparativa e Proposta de Extensão à Arquitetura Criptográfica Java

Alexandre Melo Braga[1]

Setembro de 1999

Banca Examinadora:

- Ricardo Dahab (Orientador)
- Profa. Dilma Menezes da Silva
  Departamento de Ciência da Computação (IME-USP)
- Prof. Cláudio Leonardo Lucchesii
  Departamento de Ciência da Computação (IC-UNICAMP)
- Prof. Paulo Lício de Geus (Suplente)
  Departamento de Ciência da Computação (IC-UNICAMP)

[1]Supported in part by CAPES, FAPESP, grant 97/11128-3, and ALFA Project for Exchange of Post-graduates between Latin America and Europe

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Braga, Alexandre Melo

B73a Análise comparativa e proposta de extensão à arquitetura criptográfica java / Alexandre Melo Braga -- Campinas, [S.P. :s.n.], 1999.

Orientador : Ricardo Dahab

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia. 2. Programação orientada a objetos (Computação). 3. Engenharia de software. I. Dahab, Ricardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 01 de setembro de 1999, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Dilma Menezes da Silva
IME-USP

Prof. Dr. Cláudio Leonardo Lucchesi
IC - UNICAMP

Prof. Dr. Ricardo Dahab
IC - UNICAMP

"Far better it is to dare mighty things, to win glorious triumphs, even though checkered by failure, than to take rank with those poor spirits who neither enjoy nor suffer too much, because they live in the gray twilight that knows not victory nor defeat."

É melhor lançar-se à luta em busca do triunfo, mesmo expondo-se ao insucesso, a formar filas com os pobres de espírito que nem sofrem muito, nem gozam muito, porque vivem nessa penumbra cinzenta que não conhece vitória ou derrota.

Theodore Roosevelt (1858 - 1919)

Prefácio

No limiar da sociedade da informação, cada computador pessoal, aparelho de televisão ou telefone celular contém, ou conterá muito em breve, software para comunicação em rede. Esse software deve garantir as propriedades de segurança (integridade, autenticação, sigilo e não repúdio) de uma grande variedade de atividades, tais como comércio eletrônico, correio eletrônico, acesso a bases de dados distribuídas, teleconferência, etc. A segurança de informações baseada em criptografia, antes conhecida apenas por um grupo pequeno de especialistas, hoje preocupa uma boa parcela da comunidade de software, quer na indústria, quer na academia.

Por outro lado, software não é mais desenvolvido como há quatro décadas. Soluções monolíticas programadas artesanalmente para problemas específicos não são econômicas. A composição de componentes e a reutilização em larga escala são duas características que devem estar presentes nos softwares de segurança modernos.

Esta dissertação combina técnicas de engenharia de software e segurança de informações. O resultado é um arcabouço de software que não somente oferece reutilização em larga escala de mecanismos da criptografia, mas (principalmente) captura o conjunto de cenários fundamentais de utilização das técnicas criptográficas.

Atividades interdisciplinares às vezes exigem traduções de conceitos de uma disciplina para outra. Esta dissertação oferece uma abordagem inovadora para o tratamento dos mecanismos da criptografia: uma redefinição ou reescrita destes mecanismos como um conjunto fortemente coeso de padrões de projeto. Este conjunto de padrões auxiliará engenheiros de software inexperientes em segurança a identifcar os serviços mais adequados às suas necessidades.

Agradecimentos

No dia 2 de Março de 1997 eu cheguei a Campinas para iniciar o Mestrado em Ciência da Computação na Unicamp. Como todo início, o primeiro semestre daquele ano não foi fácil. Não somente por ter cido o primeiro semestre do mestrado, mas por ter cido o início de um período de grandes mudanças na minha vida. A paisagem era nova e as pessoas também. Logo depois veio a procura por orientação e projeto de pesquisa, seguidas de um fase de muito trabalho (e alguma diversão). Já se passaram quase dois anos e meio desde então e, emquanto eu escrevo estas linhas, aqui do outro lado do oceano, não consigo evitar um sentimento de dever cumprido acompanhado de uma gratidão imensa por todos aqueles que, de uma maneira ou de outra, direta ou indiretamente, contribuíram para o sucesso do meu mestrado.

Eu gostaria de agradecer a minha família, pelo apoio incondicional e compreensão de que eu precisava buscar o meu destino longe de casa, e a meus orientadores pela amizade e orientação, sem as quais esta dissertação não teria sido possível.

Eu também gostaria de agradecer aos amigos que me acolheram em Campinas e na Unicamp. A lista seria enorme; para não pecar pelo esquecimento, eu apenas faço estas referências gerais: os meninos da república; os membros do MST-4 (Movimento dos Sem Tese 4 de Maio); os colegas da turma de mestrado de 1997; os colegas do Laboratório de Sistemas Distribuídos; o meu escravo de iniciação científica; todos os colegas da pós-graduação, professores e funcionários do instituto de computação da Unicamp; os meus amigos de Belém, sempre presentes apesar da distância; os colegas, professores e funcionários do Departamento de Sistemas Distribuídos, Instituto de Sistemas de Informação, da Universidade Técnica de Viena.

Conteúdo

Prefácio vi

Agradecimentos vii

1 Introdução Geral 1

2 Composing Cryptographic Services: A Comparison of Six Cryptographic APIs 4
- 2.1 Introduction . . . 4
- 2.2 Cryptographic Goals and Patterns . . . 5
  - 2.2.1 Cryptographic Patterns . . . 6
- 2.3 Comparative Analysis . . . 9
  - 2.3.1 General Evaluation . . . 9
  - 2.3.2 How CAPIs Perform Signing . . . 11
  - 2.3.3 How CAPIs Support Service Composition . . . 15
  - 2.3.4 Pattern Support . . . 16
- 2.4 Conclusions and Future Work . . . 17

3 Tropyc: A Pattern Language for Cryptographic Object-Oriented Software 18
- 3.1 Introduction . . . 18
- 3.2 *PayPerClick* - An Electronic Payment System . . . 19
- 3.3 A Pattern Language for Cryptographic Software . . . 21
  - 3.3.1 Pattern 1: Secure-Channel Communication . . . 23
  - 3.3.2 Pattern 2: Information Secrecy . . . 26
  - 3.3.3 Pattern 3: Message Integrity . . . 28
  - 3.3.4 Pattern 4: Sender Authentication . . . 30
  - 3.3.5 Pattern 5: Signature . . . 32
  - 3.3.6 Pattern 7: Secrecy with Sender Authentication . . . 35
  - 3.3.7 Pattern 8: Secrecy with Signature . . . 37

3.3.8 Pattern 9: Signature with Appendix . . . 38
3.3.9 Pattern 10: Secrecy with Signature with Appendix . . . 39
3.4 Deploying the Cryptographic Pattern Language . . . 41
3.5 Conclusions . . . 46

4 A Reflective Variation for the Secure-Channel Communication Pattern 47
4.1 Introduction . . . 47
4.2 Reflective Secure-Channel Communication Pattern . . . 48
4.3 Conclusions and Future Work . . . 59

5 A Meta-Object Library for Cryptography 60
5.1 Introduction . . . 60
5.2 Cryptographic Services and Patterns . . . 61
5.3 Meta-Object Library . . . 62
5.3.1 Securing Keys with Meta Proxies . . . 63
5.3.2 Reflecting Over Transformations . . . 64
5.3.3 Composing Cryptographic Services . . . 67
5.3.4 The Underlying Cryptographic Service Library . . . 69
5.4 Reflective Framework for Cryptography . . . 72
5.5 MOLC's Reconfiguration Policy . . . 74
5.6 MOLC Programming Overview . . . 77
5.7 Conclusions and Future Work . . . 81

6 The Role of Patterns in an Object-Oriented Framework for Cryptography 82
6.1 Introduction . . . 82
6.2 Cryptographic Patterns Overview . . . 83
6.2.1 Reflective Secure-Channel Communication . . . 87
6.3 Pattern Languages Generate Frameworks . . . 88
6.4 Documenting the Framework . . . 89
6.4.1 Using the Framework . . . 90
6.4.2 Designing the Framework . . . 94
6.5 Performance Evaluation . . . 100
6.6 Conclusion and Future Work . . . 104

7 ReflectiveMiMi: A Reflective Security Wrapper to the MiMi E-Commerce Tool 106
7.1 Introduction . . . 106
7.2 MiMi E-Commerce Tool . . . 107

7.3 ReflectiveMiMi Wrapper 108
7.4 Performance Evaluation 112
7.5 Conclusions and Future Work 115
7.6 Acknowledgments 116

8 Conclusão Geral 117

A Basic Cryptographic Concepts 120
A.1 Cryptographic Mechanisms 120
A.2 Common Attacks 122
A.3 Auxiliary Services 123

Bibliografia 124

Lista de Figuras

2.1 Relationships Among Cryptographic Services. 6
2.2 Pattern Distribution Over the Regions of Cryptography Services. 7
2.3 Matrix of Cryptographic Services Combinations. 8
2.4 State Diagram for Cryptographic Transformations. 10
2.5 Cryptographic APIs and the Mechanisms They Support. 10
2.6 Cryptographic APIs and the Patterns they Support. 16

3.1 Participants in an Electronic Payment Transaction 20
3.2 Cryptographic Design Patterns and Their Relationships. 22
3.3 Class Diagram for the Secure-Channel Communication Pattern. 24
3.4 Sequence Diagram for the Secure-Channel Communication Pattern. 25
3.5 Class Diagram for the Payment Transaction. 41
3.6 Sequence Diagram for the Payment Transaction. 42

4.1 Secure-Channel Communication Structure. 49
4.2 Secure-Channel Communication Dynamics. 49
4.3 Reflective Secure-Channel Communication Structure. 51
4.4 Reflective Secure-Channel Communication Dynamics. 52
4.5 Ted Structure. 54
4.6 Ted Dynamics. 55

5.1 Architecture of MOLC. 63
5.2 Abstract Meta-Objects for Generic Transformations. 65
5.3 Meta-Objects for Cryptographic Services and Their Compositions. 66
5.4 Message Hierarchy for Cryptographic Service Selection. 68
5.5 Relationships among Meta-Objects and Adapters. 70
5.6 Secure Objects Hiearachy. 71
5.7 Cryptographic Service Verification Exceptions. 72
5.8 A Reflective Object-Oriented Framework for cryptography-based Security. 73
5.9 Runtime configuration for the example application. 75
5.10 Summary of the Reconfiguration Policy Applicability. 76

6.1 Secure-Channel Communication Structure. 84
6.2 Cryptographic Design Patterns and Their Relationships. 85
6.3 Reflective Secure-Channel Communication Structure. 88
6.4 Sequencing of Actions During Method Interception. 93
6.5 Levels for Framework's Design Documentation. 94
6.6 Framework High-Level Organization. 96
6.7 Patterns in the Design of the Cryptographic Framework. 97
6.8 Instantiations of Adapter and NullObject Patterns. 99
6.9 MetaEncryptionParams' super classes. 99
6.10 Cost for Method Interception and Re-sending. 102
6.11 Cost for Some Cryptographic Transformations Performed by Meta Objects in a MicroSparc. 103
6.12 Cost for Some Cryptographic Transformations Performed by Meta Objects in a SparcStation-5. 103
6.13 Cost for Storing Objects of Different Sizes, with and without Security in a MicroSparc 100 MHz. 104
6.14 Cost for Storing Objects of Different Sizes, with Meta-Level Security in a MicroSparc 100 MHz. 105

7.1 Flow of Sensitive Data through MiMi Participants. 107
7.2 Coin Class Hierarchy. 110
7.3 ReflectiveMiMi Class Diagram. 111
7.4 Time Required by MiMiOrder to Receive # Coins With and Without Encryption in the Meta Level. 113
7.5 Time Required to Perform Encryption and Decryption with Integrity Checking, With and Without Reflection. 113
7.6 Time Consumed by HotJava (With Encryption) and MiMiVendor (With Encryption in the Meta Level) to Process a Purchase Transaction. 114

A.1 A Typical Cryptosystem. 121

Lista de Classes, Interfaces e Techos de Código

3.4.1 Classes Signer and Verifier 43
3.4.2 Class Payer 44
3.4.3 Class Payee 45
3.4.4 Class Broker 46
4.2.1 UncorruptedObject 56
4.2.2 Ted 57
4.2.3 Method Ted.main() 57
4.2.4 AliceAndBob 58
5.6.1 Creating Cryptographic Meta Objects 78
5.6.2 Composing Cryptographic Meta Objects 79
5.6.3 Executing The Base-Level Application 79
5.6.4 Protecting Keys in Meta Objects 80
5.6.5 The Glue Program 80
6.4.1 SaveObject.java — Saves Serialized Objects 90
6.4.2 SecureObject.java — Application's Main Program 92
6.4.3 MetaEncryptionParams.java 100
6.4.4 SealedObject.java 101

Capítulo 1

Introdução Geral

Esta dissertação aborda o uso de técnicas avançadas de estruturação de software na implementação de requisitos de segurança; em particular, um arcabouço de software criptográfico capaz de oferecer facilidade de uso e reutilização em larga escala.

Programadores de software criptográfico se preocupam, geralmente, com algoritmos e protocolos. Tendo em vista a preferência, justificada, dos desenvolvedores de software criptográfico pela implementação eficiente (veloz) e eficaz (segura) de tais algoritmos e protocolos, a produção de bibliotecas de software para serviços de segurança reutilizáveis e fáceis de usar tem recebido pouca atenção. Mecanismos criptográficos geralmente recebem implementações monolíticas ou são agrupados em coleções de funções pouco relacionadas. Assim, bibliotecas criptográficas disponíveis atualmente não provêem, de forma satisfatória, a combinação de mecanismos e não oferecem interfaces de programação simples, dificultando a tarefa de programadores não especialistas em criptografia. Em contraste, tais mecanismos criptográficos não costumam ser usados isoladamante, mas em combinações apropriadas.

A criptografia tem papel fundamental no uso comercial das redes abertas de computadores, como a Internet. O enorme aumento no uso destas redes para troca de informações valiosas tem contribuído para o aumento na procura por serviços de segurança de computadores. Mecanismos de criptografia estão presentes não somente em aplicações com requisitos de segurança fortes, como por exemplo os sistemas de pagamento eletrônico, mas também em softwares de uso geral, tais como processadores de texto e correio eletrônico. Consequentemente um número grande de programadores usa serviços criptográficos em seus produtos, mas muito poucos são especialistas em segurança ou criptografia. Nesse contexto, bibliotecas criptográficas reutilizáveis e fáceis de usar podem inibir a proliferação de implementações pobres e o uso incorreto dos serviços de segurança.

Segurança baseada em criptografia é geralmente um requisito não funcional ou adminstrativo (isto é, aqueles não diretamente relacionados às finalidades da aplicação) em

aplicações de uso geral. Tecnologias que promovem a separação explícita entre as responsabilidades da aplicação e os serviços de segurança facilitam o reuso em larga escala de software criptográfico e diminuem a carga de conhecimento sobre criptografia exigida dos programadores.

Esta dissertação possui dois tópicos fortemente relacionados. Primeiro, uma abordagem inovadora para o projeto e implementação de mecanismos criptográficos: o tratamento destes mecanismos como uma linguagem de padrões de projeto capaz de auxiliar engenheiros de software sem experiência em técnicas criptográficas a tratar os requisitos de segurança das aplicações. Segundo, um arcabouço de software para criptografia usado tanto no desenvolvimento de software com requisitos de segurança fortes, como na adição a posteriori de mecanismos criptográficos a softwares de terceiros e sistemas legados. Técnicas avançadas de estruturação de software, tais como padrões de projeto, estilos de arquitetura de software e reflexão computacional, são usadas neste arcabouço para proporcionar facilidade de uso e reutilização em larga escala de mecanismos criptográficos. Este arcabouço está implementado na linguagem de programação Java e usa uma arquitetura de software reflexiva para esta linguagem [OGB98].

Um estudo comparativo de bibliotecas criptográficas [BDR99b] mostrou que elas não satisfaziam a algumas características desejáveis em usos práticos dos mecanismos criptográficos: reutilização em larga escala, facilidade de uso e composição. Por outro lado, o interesse em padrões e arquiteturas de software e a existência de soluções bem conhecidas para problemas recorrentes de segurança de informações motivaram o desenvolvimento de uma linguagem de padrões para software criptográfico orientado a objetos [BRD99c, BRD98a, BRD98b]. Este conjunto de padrões de projeto é uma abordagem inovadora no uso das técnicas criptográficas, e valoriza a combinação de mecanismos.

Uma biblioteca de metaobjetos criptográficos [BDR99a] foi construída sobre a interface de programação criptográfica Java e usa um protocolo de metaobjetos para esta linguagem. Assim como os padrões, esta biblioteca também valoriza a composição de mecanismos. Uma variação reflexiva do padrão criptográfico fundamental [BRD99a] documenta a separação explícita entre responsabilidades da aplicação e serviços de segurança; esta separação é necessária à obtenção de facilidade de uso. Esta biblioteca de metaobjetos foi ampliada e constitui um arcabouço de software para criptografia [BRD99b, BDR99a] usado tanto no desenvolvimento de software com requisitos de segurança fortes, como na adição a posteriori de mecanismos criptográficos a software de terceiros e sistemas legados. Este arcabouço porporciona reutilização de software criptográfico em larga escala.

Finalmente, o arcabouço de software criptográfico foi usado para adicionar, a posteriori, integridade de dados e sigilo a uma ferramenta experimental para pagamento eletrônico. Esta tarefa [Bra99] foi realizada com pouca intrusão na aplicação alvo (sendo completamente transparente em alguns casos) e apresentou uma porcentagem bastante

alta de reutilização.

Esta dissertação é a coleção de relatórios técnicos e artigos científicos, publicados ou submetidos para publicação em conferências internacionais, obtidos durante este projeto de pesquisa. O restante desta dissertação está organizada da seguinte forma: o Capítulo 2 [BDR99b] compara alguns exemplos da abordagem tradicional para implementação de mecanismos criptográficos e aponta fraquezas nestas abordagens; o Capítulo 3 [BRD98a, BRD99c] apresenta uma abordagem inovadora para a compreensão dos mecanismos da criptografia como um conjunto fortemente coeso de padrões de projeto; o Capítulo 4 [BRD99a] oferece uma variação reflexiva para o padrão de projeto fundamental do Capítulo 3 e prepara o terreno para a biblioteca de metaobjetos criptográficos apresentada no Capítulo 5; a hierarquia de classes e o funcionamento da biblioteca de metaobjetos criptográficos são tratados pelo Capítulo 5 [BDR99a]; o Capítulo 6 [BRD99b] estabelece o relacionamento intrínseco entre os padrões de projeto criptográficos dos capítulos 3 e 4 e um arcabouço de software baseado nos metaobjetos do Capítulo 5; o Capítulo 7 [Bra99] apresenta um estudo de caso no qual o arcabouço de software para criptografia é usado para adicionar integridade de dados e sigilo a um software experimental de comércio eletrônico; o Capítulo 8 contém as conclusões obtidas com esta pesquisa. o Apêndice A contém uma breve introdução à criptografia (extraída de [BRD99c]).

Capítulo 2

Composing Cryptographic Services: A Comparison of Six Cryptographic APIs

2.1 Introduction

Modern software systems, such as electronic commerce applications, usually have strong cryptography-based security requirements, which usually need either the composition of several cryptographic mechanisms as higher-level services or the combination of cryptographic services in a *quasi*-transparent way. Although the number of cryptographic mechanisms and their valid combinations is small, most of the presently widely used Cryptographic Application Programming Interfaces (CAPIs) do not support the full set of valid mechanism combinations. How easily an unsupported combination of mechanisms can be obtained from the supported ones is, in our opinion, an important criteria for CAPI evaluation and is directly related not only to the CAPI's reusability but also to it's cryptographic unawareness. Cryptographic awareness is the amount of knowledge about cryptography required by the application programmer [gcs96]. We have documented the set of cryptographic mechanism combinations in a pattern language for cryptographic software, called *Tropyc* [BRD98a].

In this work, *Tropyc* is used to evaluate six widely used CAPIs. Compliance to *Tropyc* means that the CAPI has the basic mechanisms required to either offer higher-level cryptographic services or instantiate the corresponding cryptographic patterns. Cryptographic service is a high-level, usually more complex, cryptographic work performed/offered by an entity, based on the corresponding cryptographic mechanisms, upon receiving requests from its clients. The cryptographic services are the following subset of the security services defined by ISO [iso98] (also called cryptographic goals [MvOV96]): data confidentiality,

data integrity, authentication and non-repudiation. Accordingly, cryptographic mechanisms are the following subset of ISO's security mechanisms [iso98]: encryption, digital signatures, data integrity and authentication exchange.

We present a comparative study of six cryptographic APIs with respect to how they support *Tropyc*'s patterns. Our goal is to contrast the cryptographic mechanisms offered by widely used cryptographic APIs with the complete set of cryptographic patterns. Particularly, we want to answer three questions: (*i*) What patterns are supported by each API? (*ii*) How easily are they supported? (*iii*) Does the lack of any pattern influence the usefulness of the API in any way? Other CAPI comparisons, based on quite different criteria, can be found in [Tea97, msc].

This text is organized as follows. Section 2.2 summarizes the goals of modern cryptography and gives an overview of *Tropyc* and how it can be used to evaluate CAPIs. Section 2.3 compares the cryptographic libraries and analyzes their support to *Tropyc*. Section 2.3.2 analyses, using function interfaces, the approaches used by CAPIs for the signing data and compose cryptographic mechanisms. Conclusions and future work are in Section 2.4.

2.2 Cryptographic Goals and Patterns

Modern cryptography addresses four security goals [MvOV96] or services [iso98]: confidentiality, integrity, authentication, and non-repudiation. Accordingly, there are four basic cryptographic mechanisms: (*i*) encryption/decryption, (*ii*) MDC (Modification Detection Code) generation/verification, (*iii*) MAC (Message Authentication Code) generation/verification, and (*iv*) digital signing/verification. These four mechanisms can be combined in specific and limited ways to produce more high-level ones and are the building blocks for security services as well as security protocols. Confidentiality is the ability to keep information secret except from authorized users. Data integrity is used to guarantee that 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.

As shown in Figure 2.1, the cryptographic mechanisms corresponding to the services for data integrity, sender authentication and (digital) signatures relate to each other as follows: MACs support data integrity, signatures support both sender authentication and data integrity as well as non-repudiation. Encryption, which supports confidentiality, is orthogonal to the other cryptographic mechanisms and can be combined with each of them. It is important to notice that Figure 2.1 is related to cryptographic mechanisms

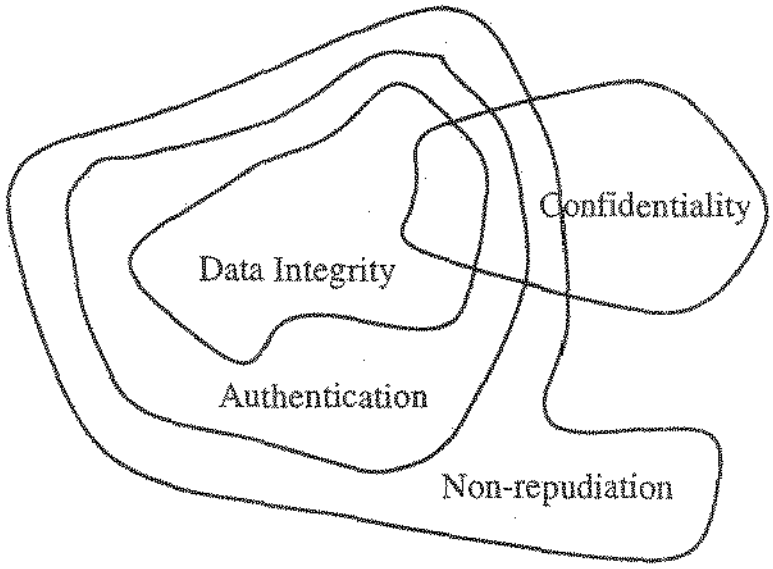

Figura 2.1: Relationships Among Cryptographic Services.

and services, it does not necessarily work for security protocols.

2.2.1 Cryptographic Patterns

The basic cryptographic services are invoked in appropriate combinations with other services and mechanisms in order to satisfy application requirements. Particular cryptographic mechanisms can be used to implement the basic services. Software systems may implement particular combinations of the basic cryptographic services for direct invocation. We have proposed in [BRD98a], a pattern language which addresses the proper combinations of cryptographic services, when security aspects are so important that they cannot be delegated (relegated) to the communication or storage subsystem and should be treated by the application itself [SRC84]. The cryptographic design patterns corresponding to the basic cryptographic services and their compositions are summarized in Table 2.2.1. The codes in column Code of Table 2.2.1 are used in Figures 2.2 and 2.3 to name the cryptographic patterns.

Pattern Name	Code	Scope	Purpose
Information Secrecy	IS	Confidentiality	Provides secrecy of information
Message Integrity	MI	Integrity	Detects corruption of a message
Sender Authentication	SA	Authentication	Authenticates the origin of a message
Signature	S	Non-repudiation	Provides the authorship of a message
Signature with Appendix	SAp	Non-repudiation	Separates message from signature
Secrecy with Integrity	SI	Confidentiality and Integrity	Detects corruption of a secret
Secrecy with Sender Authentication	SSA	Confidentiality and Authentication	Authenticates the origin of a secret
Secrecy with Signature	SS	Confidentiality and Non-repudiation	Proves the authorship of a secret
Secrecy with Signature with Appendix	SSAp	Confidentiality and Non-repudiation	Separates secret from signature

Table 2.2.1: The Cryptographic Design Patterns and Their Purposes.

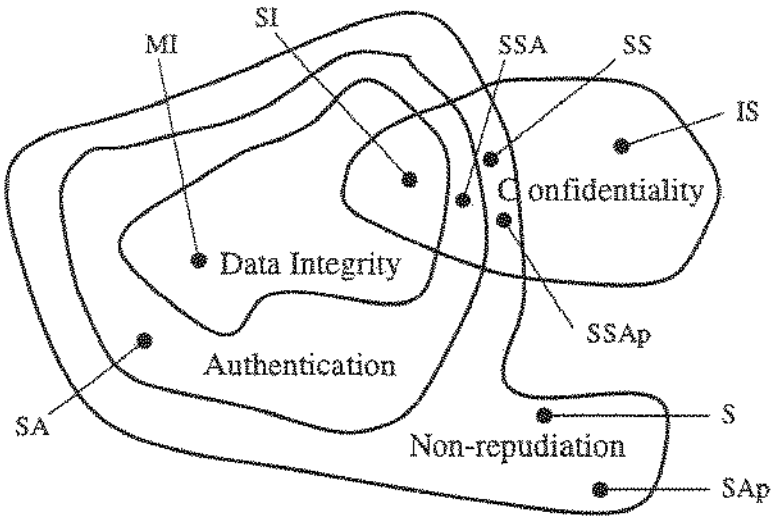

Figura 2.2: Pattern Distribution Over the Regions of Cryptography Services.

Figure 2.2 shows the distribution of cryptographic patterns over the regions of Figure 2.1. There is at least one pattern in each region, and two of them when an alternative, usually faster, variation exists. Because there are no uncovered regions in Figure 2.2, that is regions without a pattern bound to it, all the proper combinations of the four basic cryptographic mechanisms are represented by *Tropyc*. Furthermore, there are no new combinations available. This fact is supported by the matrix of Figure 2.3.

Additionally, *Tropyc* documents both the use and appropriate combination of cryptographic mechanisms in order to accomplish not only the basic cryptographic services, but also the high-level composed ones. In fact, the combined patterns can be viewed as high-level services able to increase the ease of use of cryptographic libraries and APIs. CAPIs

should offer not only the basic four mechanisms, but also their compositions. From a programmer point of view, CAPIs can support the composed cryptographic patterns in a variety of ways, ranging from explicit programmer-made composition of basic mechanisms to transparent composition hidden in high-level, not necessarily programmer-friendly, interfaces.

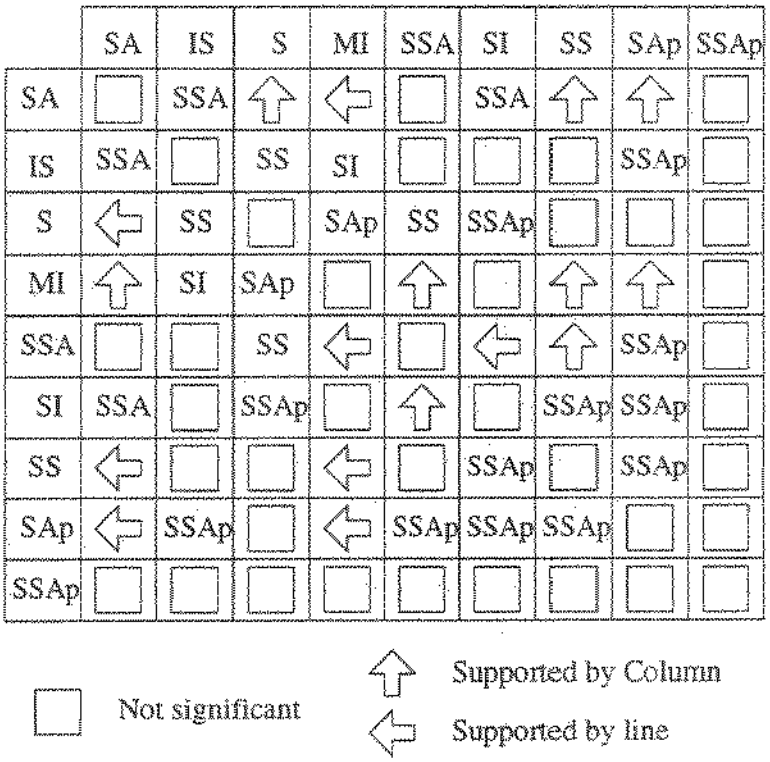

Figura 2.3: Matrix of Cryptographic Services Combinations.

The matrix of Figure 2.3 shows how to obtain valid combinations of mechanisms. The full set of mechanisms in columns are combined with the same set in rows. A square-marked position means that the combination does not add any new cryptographic feature to any of its generators. A left-arrow-marked position means that the resulting combination is already implicitly supported by the row generator. An up-arrow-marked position means the combination is already implicitly supported by the column generator. The valid combinations are marked with the codes naming the patterns. Three Observations emerge from that figure: (*i*) the number of distinct valid positions (combinations) is small; (*ii*) there are alternative ways of reaching an appropriate combination; and (*iii*) it is not necessary to add either columns or rows to the matrix because there are no new valid combinations beyond SSAp.

Since *Tropyc* completely covers not only the cryptographic services, but also their compositions, the evaluation of CAPIs based on it is complete too. Such CAPIs should

not only support the basic patterns, but also the combined ones, in order to be complete. A CAPI supporting only the basic set is considered to be less adequate for modern applications than a complete one. On the other hand, the level of abstraction for providing the combinations is another aspect to be analyzed. These two aspects, support to appropriate combinations and level of abstraction, are different approaches of evaluation, in the sense that the first is quantitative and the second is qualitative.

2.3 Comparative Analysis

Our analysis focuses on six well known CAPIs: IBM's Common Cryptographic Architecture (CCA) [JDK+91, LMJW93], the oldest of the group; RSA's Cryptoki [Kal95]; Microsoft's CryptoAPI [Mic96]; Sun's Java Cryptographic Architecture and its Extension (JCA/JCE) [JBK98, Oak98, MDOY98], the newest of the six; X/Open's Generic Security Service API (GCS-API) [gcs96], presently deprecated; and Intel's Common Security Services Manager API (CSSM-API) [css97].

Most of these CAPIs, except JCA/JCE and CCA, were evaluated in another comparison according to quite different criteria [Tea97]. Section 2.3.1 provides a general evaluation of the six CAPIs according to the cryptographic services they support. The CAPIs' support to patterns is evaluated in Section 2.3.4. A complete description of each CAPI is beyond the scope of this text.

2.3.1 General Evaluation

Object-oriented CAPIs, supported by object libraries, are easier to use, and more difficult to abuse, than those ones based on function libraries. Object libraries hid potentially harmful information and offer higher abstractions than function-based ones. Among the analyzed CAPIs, only Sun's JCA/JCE and RSA's Cryptoki present an object-oriented design, but only the first has widely available object-oriented implementations. In both cases, classes encapsulate families of semantically related functions. The other CAPIs are collections of loose-related functions (usually with large argument lists) and data types. They are usually less friendly and more prone to programming errors than the object-oriented ones.

Despite the kind of transformation performed, both object-oriented and non-object-oriented CAPIs traverse the same state diagram, shown in Figure 2.4, during data transformation. There are three states: (*i*) Initialization, during which the cryptographic engine responsible for data transformation is initialized with keys and other algorithm parameters; (*ii*) Buffer Updating, in which the cryptographic engine's internal buffer is filled and some intermediary transformation is optionally performed; and (*iii*) Finaliza-

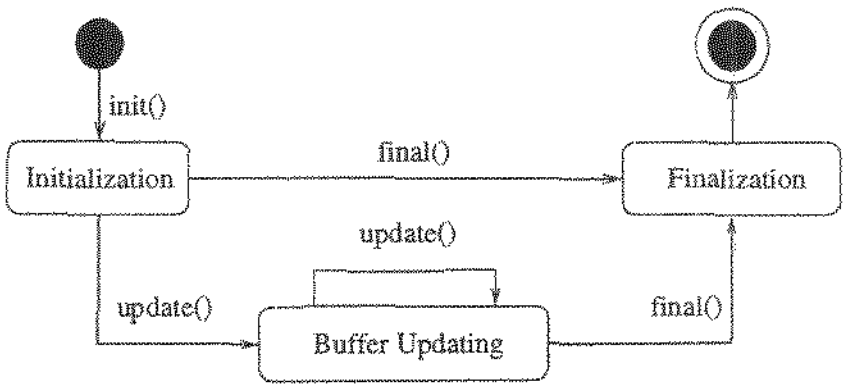

Figura 2.4: State Diagram for Cryptographic Transformations.

tion, in which the last part of input data is accumulated and transformed. Accordingly, there are three main kinds of methods or functions responsible for state transitions: init(), update() and final().

Almost all CAPIs use byte arrays as the data structure for input and output. This requires from the CAPIs' client a great control on both data granularity and length. In fact, all CAPIs, except JCA/JCE, force their clients to worry about both block size and padding. JCA/JCE is able to work not only with byte arrays, but also with serializable objects for both signing and encryption [GS98].

Service \ API	Cryptoki	GCS-API	JCA/JCE	CryptoAPI	CSSM-API	IBM CCA
Encryption	■	■	■	■	■	■
Encryption and Authentication				■		
Encryption and Data Integrity		■		■		
Encryption and Non-repudiation		■		■		
Digital Signature with Recovery	■					
Digital Signature with Appendix	■	■	■	■	■	■
Cryptographic Hashing/MDC	■	■	■	■	■	■
Random Numbers	■	■	■	■	■	■
Key/Key Pair Generation	■	■	■	■	■	■
Key Derivation	■	■			■	
Key Wrapping	■	■		■	■	■
Key Agreement		■	■			
Message Authentication Code				■	■	■

Figura 2.5: Cryptographic APIs and the Mechanisms They Support.

The table in Figure 2.5 summarizes the cryptographic mechanisms the six CAPIs approach. These mechanisms can be divided in three main groups. The basic mechanisms: encryption, hashing/MDC, MAC, and Signature with recovery; their compositions, formed by the Signature with appendix and the combinations of encryption with digital signatures, MDCs, or MACs; and the others, forming the group of management or auxiliary functions. Most of the CAPIs support the basic mechanisms, though a few of them explicitly support mechanism composition. Auxiliary functions are not uniformly supported. Figure 2.5 shows that none of the analyzed CAPIs offers routines for the complete set of mechanisms and their compositions. Furthermore, from a programmer point of view, mechanism composition is usually a difficult task, which requires a lot of knowledge about the topic. Microsoft's CryptoAPI is the most complete, but Sun's JCA/JCE is the most programmer-friendly. Section 2.3.2 offers a comparison of function and method interfaces for Dada signing and Section 2.3.3 analyses the composition of cryptographic mechanisms.

2.3.2 How CAPIs Perform Signing

This Section compares how CAPIs perform data signing and illustrates the key differences among them. These solutions go from unique powerful-but-complex functions to easy-to-use objects which encapsulate both signatures and signed data. We compare Sun's JCA, X-Open's GCS-API, RSA's Cryptoki and Microsoft's CryptoAPI. Each of them uses a different approach for data signing. Two CAPIs are not shown here: CSSM-API approaches signing similarly to Cryptoki and IBM's CCA behaves like GCS-API.

```
minor_status generate_check_value(
        session_context,                                    /* output*/
        input_data,                                         /* input */
        iv,                                              /* input, optional*/
        chain_flag,                                         /* input */
        cc,                                              /* input, output */
        intermediate_result,                             /* input, output */
        check_value                                         /* output */
);
```

X-Open's GCS-API uses a simple function, `generate_check_value()`, for all stages of data transformation (the stages are shown in Figure 2.4 and are initialization, buffering, and signing). Such a powerful function, whose interface is above, requires a large number of arguments, seven in total, to perform its task over arrays of bytes (`input_data`,

intermediate_result, and check_value). A flag, chain_flag, indicates whether the transformation is in first, middle or final stage. Large arrays can be split and treated by successive calls, in such a case, the function should be explicitly fed back with intermediate_results. Bye the time of the final call, check_value contains the signature. cc stands for cryptographic context, which is a structure for encapsulating both sensitive data, such as keys, and the signature engine. iv is an optional initialization vector.

```
BOOL WINAPI CryptCreateHash(
        HCRYPTPROV hProv,                                   /* input */
        ALG_ID AlgId,                                       /* input */
        HCRYPTKEY hKey,                                     /* input */
        DWORD dwFlags,                                      /* input */
        HCRYPTHASH *phHash                                  /* output */
);

BOOL WINAPI CryptHashData(
        HCRYPTHASH hHash,                                   /* input */
        BYTE *pbData,                                       /* input */
        DWORD dwDataLen,                                    /* input */
        DWORD dwFlags                                       /* input */
);

BOOL WINAPI CryptSignHash(
        HCRYPTHASH hHash,                                   /* input */
        DWORD dwKeySpec,                                    /* input */
        LPCTSTR sDescription,                               /* input */
        DWORD dwFlags,                                      /* input */
        BYTE *pbSignature,                                  /* output */
        DWORD pdwSigLen                                     /* input, output */
);
```

CryptoAPI's functions for signing, above, can only produce signatures with appendix and split the signing operation in two tasks: CryptHashData() buffers data in successive calls and generates a hash from them; CryptSignHash() signs the hash stored in phHash and returns an array of bytes containing the signature; a third function, CryptCreateHash(), is used for creating a hash engine, returned in phHash.

```
CK_RV CK_ENTRY SignInit(
        CK_SESSION_HANDLE hSession,                    /* input */
        CK_MECHANISM_PTR pMechanism,                   /* input */
        CK_OBJECT_HANDLE hkey                          /* input */
);

CK_RV CK_ENTRY Sign(
        CK_SESSION_HANDLE hSession,                    /* input */
        CK_BYTE_PTR pData,                             /* input */
        CK_USHORT usDataLen,                           /* input */
        CK_BYTE_PTR pSignature,                        /* output */
        CK_USHORT_PTR pusSignatureLen                  /* output */
);

CK_RV CK_ENTRY SignUpdate(
        CK_SESSION_HANDLE hSession,                    /* input */
        CK_BYTE_PTR pPart,                             /* input */
        CK_USHORT usPartLen                            /* input */
);

CK_RV CK_ENTRY SignFinal(
        CK_SESSION_HANDLE hSession,                    /* input */
        CK_BYTE_PTR pSignature,                        /* output */
        CK_USHORT_PTR pusSignatureLen                  /* output */
);
```

The four interfaces above represent the Cryptoki's family of functions for signing. `InitSign()` initializes the signing operation; then, the function called should be `Sign()`, if data should be processed in a single part; otherwise, successive calls to `SignUpdate()`, followed by a call to `SignFinal()` should take place.

These three APIs follow a procedural programming paradigm, in fact both CryptoAPI and Cryptoki use C programming language in their specifications. Such a binding to procedural languages has direct consequences over the API design. For example, arguments that could be made implicit, such as handles, in object-oriented programming, are explicit and increase the list of arguments. Two aspects are due to C-programming binds: the use of pointers to arrays of bytes and the explicit use of variables to store the length of arrays.

```
public abstract class Signature{
```

```
        public static getInstance(String algorithm);
        public static getInstance(String algorithm,String provider);
        public final void initSign(PrivateKey prvkey);
        public final void initVerify(PublicKey pubkey);
        public final void update(byte[] b);
        public final byte[] sign();
        public final boolean verify(byte[] signature);
}

public final class SignedObject implements Serializable{
        public SignedObject(Serializable o,PrivateKey pk,Signature s);
        public boolean verify(PublicKey pk,Signature s);
        public Object getContents();
}
```

The JCA's `Signature` class, above, groups a Cryptoki-like set of functions and offers the benefits of object-oriented programming. The static method `getInstance()` can be used to instantiate particular algorithm implementations; method `initSign()` initializes the engine for signing with a private key; `update()` is used for data buffering either in single or multiple calls. Method `sign()` signs buffered data and returns the signature as an array of bytes. Since object-oriented paradigm is being used, cryptographic transformations should be applied not only over bytes, but also over objects. Alternatively to the byte-based API, JCA offers the `SignedObject` class, which can be used to sign and verify (serializable) objects. JCA also offers specialized streams to handle encryption over continuous data.

These four approaches are not limited to signing. Each CAPI keeps analogous behavior for encryption, hashing and authentication, as well as for decryption and verification procedures. In summary, the four ways used by CAPIs to perform signing are the following.

1. A unique powerful function signs data in successive calls and flags determine the stage of the transformation. Feedback of intermediary results is usually required.

2. Explicit separation of signing from both buffering and hashing. In this case, buffering can take multiple function class, but feedback is avoided.

3. A family of functions performs initialization, buffering and signing. Hashing is hidden and feedback is also avoided.

4. Signing is performed not only over byte arrays, but also over objects, by using a high level, usually more intuitive, programming interface.

2.3.3 How CAPIs Support Service Composition

Among all CAPIs analyzed in this text, only GCS-API and CryptoAPI explicitly offer cryptographic mechanisms composition procedures. GCS-API still apply its approach and offer a single powerful-but-complex function to all kinds of combinations. Such a function interface, below, requires two cryptographic contexts (one for confidentiality, the other for either integrity or signature) and returns pointers to two outputs containing the encrypted data and the check value. Intermediate results still should be fed back and a flag controls whether transformation should be performed in single or multiple parts, as well as the stage of transformation.

```
minor_status protect_data(
        session_context,                          /* output*/
        input_data,                               /* input */
        iv,                                       /* input,optional*/
        chain_flag,                               /* input */
        confidentiality_cc,                       /* input,output */
        integrity_cc,                             /* input,output */
        intermediate_result,                      /* input,output */
        output_data                               /* output */
        check_value                               /* output */
);
```

Microsoft's CryptoAPI uses the function interface below to combine encryption with finger print generation mechanisms. An authentication engine, which can be either a MDC or MAC generator or even a signature engine, is passed as an argument to the function and is used internally to buffer the data before authentication. Similarly to CGS-API, a flag indicates the last or single function call. The fingerprint is obtained by calling `CryptSignHash()`, see Section 2.3.2, thus only data buffering is, in fact, composed.

```
BOOL WINAPI CryptEncrypt(
        HCRYPTKEY hKey,                           /* input */
        HCRYPTHASH hHash,                         /* input */
        BOOL Final,                               /* input */
        DWORD dwFlags,                            /* input */
        BYTE *pbData,                             /* input,output */
        DWORD *pdwDataLen                         /* input,output */
        DWORD *dwBufLen                           /* input */
```

);

Although not allowing explicit support for cryptographic service composition, Sun's JCA reduces the complexity of this task by using SignedObjects and SealedObjects as specializations of class Serializable. When using objects for securing data, the structuring of the secure classes as specialization of serializable objects allows chaining of objects, since serializable signed objects contain other serializable objects, which can be instances of other secure objects, and so on. This is an application of the *Composite* [GHJV94] design pattern.

2.3.4 Pattern Support

The widespread use of design patterns has reached the design and implementation of object libraries for cryptography. For instance, Sun's JCA/JCE makes extensive use of *Factory Methods* [GHJV94] in order to allow subclasses to specify the objects to be created. There are other patterns related to security aspects of applications [YB97]. In this section, the CAPIs support to *Tropyc*'s cryptographic patterns is analyzed. Compliance to *Tropyc* means that the CAPI has the mechanisms required by an application in order to easily instantiate the patterns.

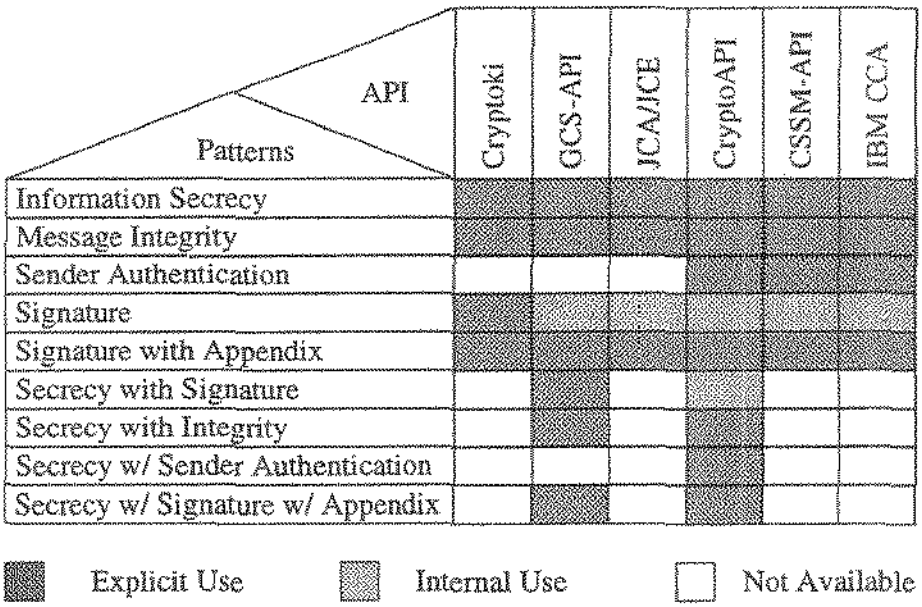

Figura 2.6: Cryptographic APIs and the Patterns they Support.

The table of Figure 2.6 confirms the results obtained from Figure 2.5. Figure 2.6 shows again that Microsoft's CryptoAPI is the most complete CAPI. Although criticized

for its great complexity, GCS-API is the second most complete. In general, the CAPIs support the four simple patterns but not the composed ones. *Signature* is usually only internally supported because it is a building block for *Signature with Appendix*, which is usually supported in a high-level way. Modern applications with cryptography-based security requirements should use either Sun's JCA/JCE or Microsoft's CryptoAPI. The first does not have objects for composed mechanisms, but is so easy to use that mechanism composition becomes a less difficult task. The second has a less friendly interface, but possesses the functions necessary to instantiate the composed cryptographic patterns with relative transparency.

2.4 Conclusions and Future Work

This work uses *Tropyc*, a pattern language for cryptographic software, to evaluate Cryptographic Application Programming Interfaces (CAPIs). In this text we evaluated six widely used CAPIs and argued that, in general, they do not support cryptographic mechanism composition in an easy-to-use way. Furthermore, none of them provides transparent cryptographic mechanism composition. Modern applications, such as electronic commerce systems, possess strong requirements on composing cryptographic mechanisms, in a way that there is a practical need for cryptographic object libraries supporting this composition. This deficiency can be overcome by either developing (proposing) a new CAPI or extending an existing one. In such a case the JCA/JCE is the best choice for an extensible CAPI because of its modern and flexible programming interface. We are using *Tropyc* to guide the extension of a household implementation of JCA/JCE, in order to explicitly address easy composition of cryptographic mechanisms, as well as easy instantiation of cryptographic patterns.

Capítulo 3

Tropyc: A Pattern Language for Cryptographic Object-Oriented Software

3.1 Introduction

Historically associated to encryption, modern cryptography is a broader subject, encompassing the study and use of mathematical techniques to address information security problems. Cryptographic mechanisms are used in a wide variety of applications, such as electronic mail, database protection, and electronic commerce. The present interest in software architectures and patterns, and the existence of well-known cryptographic solutions to recurring security problems, motivate the development of cryptographic software architectures and cryptographic patterns.

In this work, we present *Tropyc*, a pattern language addressing four fundamental security-related services [iso98]: data confidentiality, data integrity, sender authentication, and sender non-repudiation, organized as a pattern language for cryptographic software. Data confidentiality is the ability to keep information secret except from authorized users. Data integrity guarantees that information has not been modified without permission, which includes the ability to detect unauthorized manipulation. Sender authentication corresponds to the assurance, by the communicating parties, of the origin of 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.

Tropyc is composed of ten patterns. The foundation pattern is *Secure-Channel Communication*, which expands into four basic patterns that correspond to the services just described: *Information Secrecy*, *Message Integrity*, *Sender Authentication*, and *Signature*. These, in turn, generate the five remaining patterns in the language: *Secrecy with*

Integrity, Secrecy with Sender Authentication, Secrecy with Signature, Signature with Appendix, and *Secrecy with Signature with Appendix*.

The intended of this paper is to help software developers not familiar with cryptographic techniques address the security requirements of applications. Developers looking for general software architectures for their cryptographic software or components may also be interested.

The remainder of the text is organized as follows. Section 3.2 motivates the need for cryptography-based security through an electronic payment application, *PayPerClick*, which is also used throughout the paper to illustrate how our patterns can be employed. Section 3.3 describes *Tropyc* and its ten patterns. Section 3.4 describes a design of *PayPerClick* using *Tropyc*. Conclusions and future work are shown in Section 3.5.

3.2 *PayPerClick* - An Electronic Payment System

Electronic commerce applications are very good examples of systems that require cryptographic services. A simple scenario of a commercial transaction using a credit card follows:

1. Alice, a custom, asks Bob, a bookseller, for a nice book about cryptography.
2. Bob replies that he has only one copy in stock and offers it to Alice for a reasonable price.
3. Alice accepts Bob's offer.
4. Alice sends her credit card number and some further information to Bob upon his request.
5. Bob contacts the credit card's issuer to validate the card information. In case of positive validation, he accepts the payment.
6. Bob sends the book to Alice along with a receipt.

There are four security issues implicit in this scenario, which are very important when one uses an open network, such as the Internet, to implement the system. First, Alice requires confidentiality when sensitive information, such as credit card numbers, is sent to a seller. Second, this information should arrive uncorrupted at its destiny; that is, its integrity should be preserved, so it can be correctly validated. Third, Alice and Bob need mutual assurance of each other's identity during the transaction; that is, they should authenticate the sender of the exchanged messages. The same holds for Bob and the

credit card's issuer. Fourth, the payment receipt should be signed by Bob. This simple scenario becomes more complex if electronic money is used instead of credit cards. New requirements, such as anonymity and prevention of double spending, may arise.

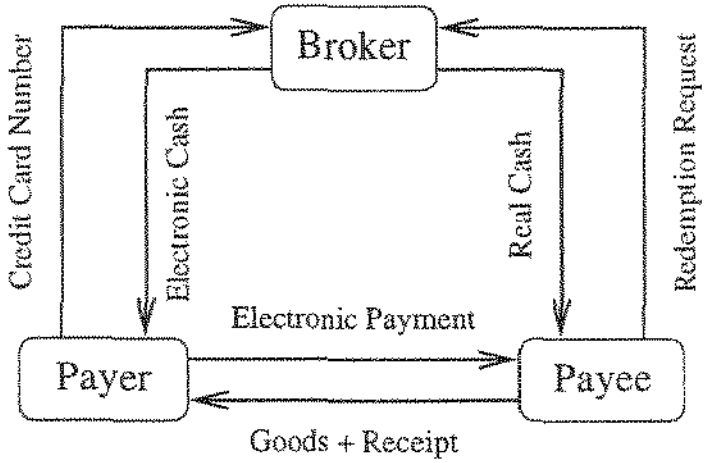

Figura 3.1: Participants in an Electronic Payment Transaction

Figure 3.1 shows the three main participants of an electronic payment transaction and the flow of money and sensitive data between them [AJSW97]. The Broker is usually a bank or any other kind of financial institution. The Payee can be an Internet access provider and the Payer is the customer. The Payer makes a request to the Broker for electronic cash that can be used to buy (electronic) goods over the Internet. The Payer can request a receipt issued by the Payee. The Payee request redemption of electronic cash to the Broker. Usually real money flows from the Broker to the Payee.

Electronic payment systems are good examples of the end-to-end argument [SRC84] applied to cryptography. In fact, cryptography-based security is such an important feature of electronic payment systems, that the application itself should deal with it avoiding delegation to underlying communication subsystems. Additional information about electronic payment systems can be found in [FD98, HY97].

PayPerClick [BDR98] is a tool for electronic purchase and on-line distribution of hypertext documents based on the model of Figure 3.1. Hypertext documents are accessible through links to HTML pages and are visible through Web browsers. *PayPerClick* can be used, for instance, to sell on-line books through links in their table of contents. Therefore, customers can buy specific parts of the book by clicking on a hyperlink. The hyperdocuments are structured according to the Composite [GHJV94, 163] pattern, which makes the computation of the cost and fingerprint of a hypertext document an easy task. The fingerprint of a hyperdocument is a Modification Detection Code (MDC) (see Appendix A) computed by traversing its tree in some order. The Payer can be implemented

as Java applets, which communicate with Web servers, instances of either Payee or Broker. Payers usually have an electronic wallet. Brokers issue to Payers electronic cash, a multiple of fixed-value electronic coins.

3.3 A Pattern Language for Cryptographic Software

This Section presents *Tropyc*, a pattern language for cryptographic software, which focuses on three goals: (*i*) the definition of a software architecture for cryptography-based secure communication, (*ii*) the description of cryptographic services as patterns, and (*iii*) the organization of these cryptographic patterns as a pattern language.

#	Pattern	Scope	Purpose
1	Secure-Channel Communication	Generic	Provides a generic software architecture for cryptographic systems
2	Information Secrecy	Confidentiality	Provides secrecy of information
3	Message Integrity	Integrity	Detects corruption of a message
4	Sender Authentication	Authentication	Authenticats the origin of a message
5	Signature	Non-repudiation	Provides the authorship of a message
6	Secrecy with Integrity	Confidentiality and Integrity	Detects corruption of a secret
7	Secrecy with Sender Authentication	Confidentiality and Authentication	Authenticates the origin of a secret
8	Secrecy with Signature	Confidentiality and Non-repudiation	Proves the authorship of a secret
9	Signature with Appendix	Non-repudiation	Separates message from signature
10	Secrecy with Signature with Appendix	Confidentiality and Non-repudiation	Separates secret from signature

Table 3.3: The Cryptographic Patterns, with Their Scopes and Purposes.

Table 3.3 summarizes the ten patterns that compose *Tropyc*. *Secure-Channel Communication* abstracts common aspects of both structure and behavior relative to secure communications, regardless of the kind of cryptographic transformation performed. *Information Secrecy* provides confidentiality of data in transit. However, information secrecy alone does not prevent modification or replacement of data. Particularly in online communication, granting *Message Integrity* and *Sender Authentication* is also important. In other situations, it is necessary to prevent entities from denying their actions or commitments. Thus, some form of *Signature* is necessary. These four basic cryptographic services, in suitable combinations, generate three more patterns: *Secrecy with Integrity*, *Secrecy with Sender Authentication*, and *Secrecy with Signature*. For implementation efficiency purposes, two additional patterns are provided: *Signature with Appendix* and *Secrecy with Signature with Appendix*.

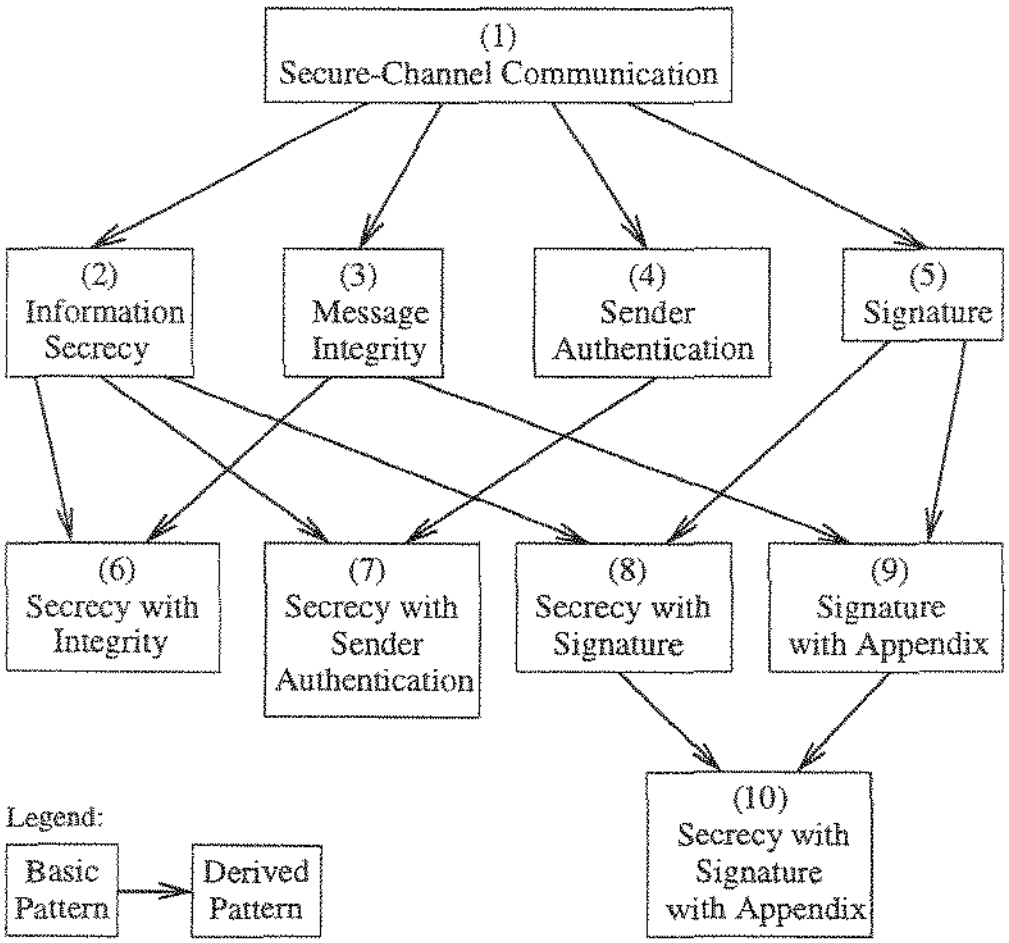

Figura 3.2: Cryptographic Design Patterns and Their Relationships.

Figure 3.2 shows a directed acyclic graph of dependencies between patterns. An edge from pattern A to pattern B means pattern A generates pattern B. Secure-Channel Communication generates the four basic patterns. The remaining patterns are derived from combinations of these. A walk on the graph is directed by two questions. First, how should the cryptographic software be structured to obtain both easy reuse and flexibility? Second, what cryptographic services should be used to address application requirements and user needs? In other words, what cryptographic services should be added to the current instantiation of Secure-Channel Communication in order to overcome its present deficiencies?

In the following patterns Alice and Bob represent two communicating participants, while Eve is an adversary eavesdropping and possibly modifying information exchanged by Alice and Bob.

3.3.1 Pattern 1: Secure-Channel Communication

Context Alice and Bob exchange data through messages, on which they need to perform cryptographic transformations. Moreover, a flexible and reusable cryptographic software architecture is required to make cryptographic service composition easier and to separate concerns between application functionality and security requirements.

Problem How should one structure flexible and reusable cryptographic software for secure communication?

Applicability

- When separation of concerns between functional requirements and non-functional security requirements should be promoted.
- When the incorporation of security in software systems should be done in a structured and disciplined manner to avoid an increase in the software's complexity.

Forces

- The dependencies between cryptographic features and the application's functionality should be minimized to facilitate reuse of the cryptographic components.
- Software with cryptographic code should be easy to understand, modify, and adapt.
- The increase of the systems' complexity due to the inclusion of security services should be kept under control.

- The performance of cryptographic algorithms should not be affected by application's design.

Solution Alice performs a cryptographic transformation $x = f(m)$ on data m before sending it to Bob. Bob receives x and performs transformation $y = g(x)$. Alice and Bob must previously agree on transformations f and g and either share or distribute keys, if necessary. Figure 3.3 shows a class diagram that models the cryptographic transformations. The diagram defines two template classes, *Alice* and *Bob*, and two hook classes, *Codifier* and *Decodifier*. The *Codifier* class has a hook method *f( )*, which performs a cryptographic transformation on m. The *Decodifier* class has a hook method *g( )*, which performs the transformation, $y = g(f(m))$. Figure 3.4 shows the interaction diagram between *Alice* and *Bob*. In these diagrams, *a*, *b*, *c* and *d* are the roles performed by instantiations of classes *Alice*, *Bob*, *Codifier* and *Decodifier*, respectively. The *Secure-Channel Communication* pattern is a high-level abstraction which is inherited by the remaining patterns of the language.

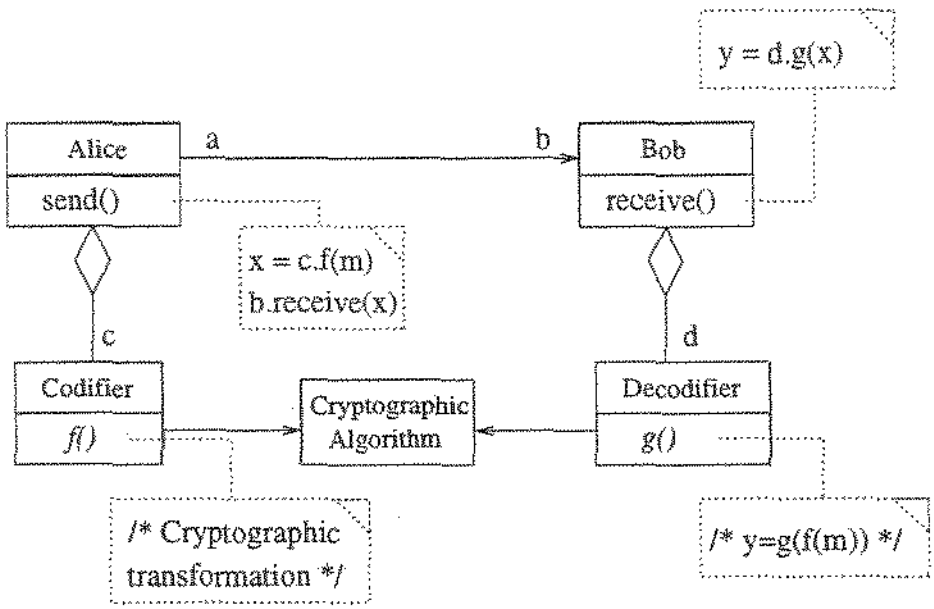

Figura 3.3: Class Diagram for the Secure-Channel Communication Pattern.

Consequences The use of the Secure-Channel Communication pattern has the following consequences:

- It separates the concerns related to the application domain from those related to the provision of cryptographic mechanisms in a structured and disciplined fashion for application developers.

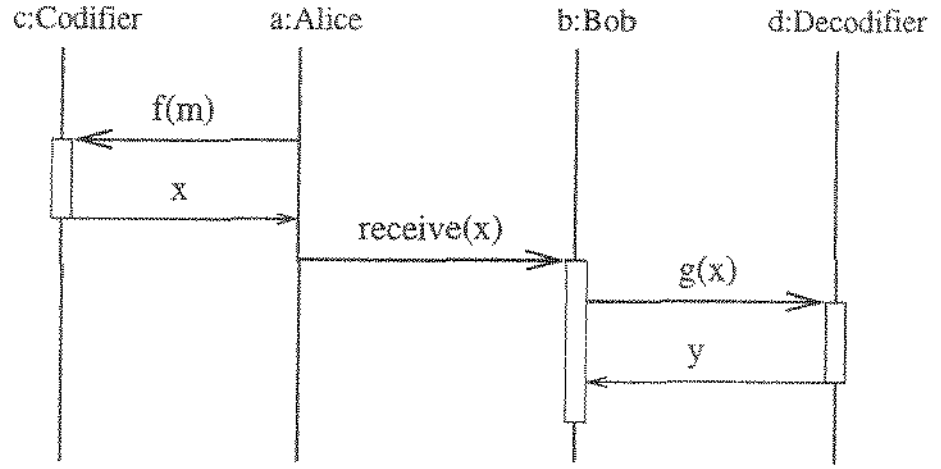

Figura 3.4: Sequence Diagram for the Secure-Channel Communication Pattern.

- It promotes reusability of cryptographic mechanisms.
- It allows application developers to choose the most adequate security strategy for the system's implementation.
- It provides a system design that is easier to maintain, adapt, change, and extend than traditional approaches for developing cryptographic software.
- It may introduce inefficiencies in the cryptographic protocols, due to the references, implicit and otherwise, made in the object-oriented design.

Resulting Context Alice and Bob can instantiate the overall architecture of cryptographic systems. However, they should choose the cryptographic patterns that are the most adequate for their application's requirements. Concrete implementations of this pattern should be based on the four basic patterns, *Information Secrecy*, *Sender Authentication*, *Message Integrity*, and *Signature*, and suitable combinations of them.

Implementation

- This pattern can be adapted to deal with file storage and recovery. In these situations, messages for storing and recovering a file replace those used for sending and receiving information.
- The reflection pattern can be used to implement the security requirements in a transparent and non-intrusive manner for application programmers, that is, without interfering in the application's original structure. The reflection mechanism encourages modular descriptions of software systems by introducing a new dimension of

modularity: the separation of base-level and meta-level computation. Reflection allows the creation of meta-objects that implement the cryptographic mechanisms and that control and extend the behavior of application objects located at the base level.

- Before secure communication begins, a negotiation step is necessary in order for the participants not only to agree on which transformation to perform, but also to exchange information, such as keys and algorithm parameters.
- Eve's role depends on the concrete implementations. She can replace or modify a message in transit in the channel, or insert her own messages.

Example Electronic payment systems have high security requirements. They can be better structured when the *Secure-Channel Communication* pattern is used. Sections 3.2 and 3.4 discuss some important aspects of our case study, *PayPerClick*.

Known Uses In the literature we can find various systems that use the Secure-Channel Communication pattern, such as [HY97, Her97, CGHK98, HN98, Lin93, Zim95, BDR98].

Related Patterns Several well-known patterns can be used when instantiating *Secure-Channel Communication*. The *Strategy* [GHJV94, 315] pattern can be used to obtain algorithm independence. The *Bridge* [GHJV94, 151] pattern can be applied to promote implementation independence. The *Abstract Factory* [GHJV94, 87] pattern can be employed in the negotiation step to choose which cryptographic 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 can be combined with the cryptographic patterns in order to offer secure and transparent inter-process communication, so that Alice becomes part of the Forwarder and Bob is incorporated into the Receiver. The *State* [GHJV94, 305] pattern can be used to provide state dependent behavior, such as turning the security of the channel on and off. The *Null Object* [MRBV97, 5] pattern can be used to design a null transformation. The *Reflection* [BMR+96, 193] pattern can be used to separate application functionality from the security requirements. The cryptographic infrastructure, as encryption/decryption algorithms, pseudo-random generators, hashing algorithms, etc., can be provided by a *Security Access Layer* [YB97, 16].

3.3.2 Pattern 2: Information Secrecy

Context Alice wants to send sensitive messages to Bob. Moreover, she wants to keep these messages secret from Eve, whom Alice suspects may be trying to read its contents.

Problem How can Alice send a message to Bob so that Eve cannot possibly read its contents?

Applicability

- When two participants need to share confidential information.
- When it is necessary to decouple encryption/decryption activity from either data communication or storage.

Forces

- Eve cannot, in any situation, gain access to the message contents.
- The cost of encryption/decryption should not be greater than the intrinsic value of the message being encrypted.
- The cost of cryptoanalysis by Eve should be much higher than that of the message itself.

Solution This pattern supports encryption and decryption of data. Alice and Bob have previously agreed on an (assumed public) encryption function and a shared secret key (in public-key cryptography, Bob must first obtain Alice's public key). Bob encrypts the message and sends it to Alice. Alice decrypts it and recovers the original message.

Consequences

- Encryption is, in general, a slow task. The security of encrypted information relies on the secrecy of the encryption key and on the strength of the encryption algorithm. Clearly a key must be long enough to prevent exhaustive search of the key space.
- Eve cannot read the message contents, but she still can replace or modify encrypted messages.
- Transmission or storage errors can potentially render the recovery of the original message difficult.

Resulting Context Alice and Bob use an encrypted channel for their communication. However, this channel does not provide data integrity, sender authentication, or non-repudiation. To achieve integrity without loss of secrecy, Alice and Bob should instantiate *Secrecy with Integrity*. If they want to add *Sender Authentication* to *Information Secrecy*, they should instantiate *Secrecy with Sender Authentication*. Furthermore, if sender non-repudiation of secrets is desired, one should instantiate *Secrecy with Signature*.

Implementation

- Both private (in public-key, or asymmetric, systems) and secret (in secret-key, or symmetric, systems) keys must be kept protected from unauthorized access.
- An infrastructure to distribute public keys is required.

Example In *PayPerClick*, when a Payer makes a cash request, he sends the Broker his credit card number in encrypted form. The Broker decrypts the card number with her decryption key and charges the Payer's credit card with the requested cash amount.

Known Uses Common uses of this pattern can be found, for example, in electronic mail systems [Zim95, Lin93, Her97], automatic banking machines, and voice encryption.

3.3.3 Pattern 3: Message Integrity

Context Alice sends long messages to Bob, who wants to verify the integrity of the received messages. He suspects that they may have been corrupted accidentally, due to transmission errors. Alice and Bob do not share cryptographic keys.

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

Applicability

- In the detection of occurrence of errors in either transmission or storage of data.
- In the detection of unauthorized modification of data.
- When it is necessary to generate "fingerprints"for either messages or data records.

Forces

- The mechanism should be robust against unauthorized, accidental or not, modification of data.
- The mechanism should be cost-effective.

Solution Alice and Bob agree to use a MDC (Appendix A). 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 to the one received from Alice. If they match, the message has not been altered.

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 message and the corresponding MDC.
- MDCs by themselves do not guarantee the authorship of a message.

Resulting Context Alice and Bob use MDCs to detect corruption of data. However, this technique guarantees neither sender authentication nor non-repudiation. If these are required, other patterns, such as *Sender Authentication and Signature*, should be used instead. Moreover, *Information Secrecy* can be combined with *Message Integrity* in the *Secrecy with Integrity* pattern, thus providing integrity and secrecy at the same time in the channel.

Implementation

- A message must be bound to its corresponding MDC to avoid mismatch of messages and MDCs.
- Measures should be taken to recover the original contents of corrupted data. MDCs can detect corruption, but not correct it. One standard solution is message replaying.
- When feasible, sending two or more copies of a message, with the additional feature of allowing weak correction capability, is an alternative to employing MDCs for detecting corruption.
- MDCs are often implemented using cryptographic hashing algorithms [MvOV96].

Example In *PayPerClick*, electronic payments must have their integrity preserved in order for the Payee verification to succeed. Thus, the payment should be sent by the Payer along with its MDC. The Payee recomputes the MDC of the received payment and checks it against the received MDC.

Known Uses Two common uses of MDCs are the detection of file modification caused by viruses and the generation of passphrases to produce cryptographic keys. Privacy-Enhanced Mail [Lin93] is one of the systems that provides *Message Integrity*. MDCs can also be used as unique identifiers of electronic coins in electronic commerce applications [FD98].

3.3.4 Pattern 4: Sender Authentication

Context Alice and Bob want to exchange messages, but they cannot distinguish their own messages from spurious ones, perhaps inserted by Eve, in the communication channel. Moreover, we assume that they have previously established a secret key using some secure channel.

Problem How can genuine messages be distinguished from spurious ones?

Applicability

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

Forces

- The authenticated messages should be hard to forge..
- The authentication mechanism should detect accidental data modification, as well as those supposedly done by Eve.
- The authentication procedure should be cost-effective; that is, it should not imply a cost higher than the intrinsic value of the data being authenticated.

Solution Alice and Bob agree beforehand on a shared secret key and a cryptographic algorithm for generation of Message Authentication Codes (MACs) (Appendix A). Alice computes the MAC of the pair (message, key) and sends both message and MAC to Bob. Bob computes the MAC of the received message and the shared key and compares it with the MAC he 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 for a given message.

Consequences

- A message is correctly authenticated if and only if the shared key is kept secret from third parties.
- The authorship of a message produced by Alice or Bob cannot be proved to a third party, since both sides can compute valid MACs.
- Eve may insert a previously seen message along with its (correct) MAC into the communication channel, thus fooling Alice and Bob. In this situation, some guarantee of the message freshness should be provided. A common solution for this problem is the inclusion of timestamps or sequence numbers as part of the message contents.

Resulting Context Alice and Bob can authenticate the origin of messages they exchange as well as detect their corruption. However, if they want to prove the authorship of messages, the *Signature* pattern should be used instead. If desired, encryption facilities can be added to the communication channel to apply the *Secrecy with Sender Authentication* pattern.

Implementation Factors

- A secure means for exchanging and maintaining a secret key is necessary.
- Similarly to MDCs, a message must be correctly bound to its corresponding MAC.
- MAC generators can be implemented in many ways. Two common possibilities are symmetric cryptosystems and cryptographic hash functions.
- As with MDCs, additional measures should be taken if error correction is desired.

Example In *PayPerClick*, Eve may try to substitute her coins for someone else's. This substitution can be prevented if MACs of the payments are computed by the Payer whenever they are sent to the Payee. Such a solution ensures that the Payee will always receive valid payments. However, a Payee can generate a fake payment and still request redemption to the Broker. Analogously, a Payer can repudiate old legitimate payments claiming that they were generated by the Payee. If the value of payments is relatively large and coin losses are frequent, the use of the *Signature* pattern is a better solution to this problem.

Known Uses MACs have been used, among other applications, to authenticate IP packages over the Internet [CGHK98].

3.3.5 Pattern 5: Signature

Context Alice sends messages to Bob, but he cannot distinguish Alice's messages from the ones Eve may insert in the communication channel. Furthermore, Alice can later dispute the authorship of a message actually sent by her, denying Bob the ability to prove to a third party that only Alice could have sent that particular message. We assume that Alice has a public/private key pair and that her public key is widely available.

Problem How can one correctly attribute the authorship of a message in such a way that this authorship cannot be later disputed? In other words, how can the receiver of a particular message convince himself and a third party of the identity of the sender of that message?

Applicability

- In contexts where 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. Thus, signatures implicitly guarantee data integrity and sender authentication.
- Signatures must be hard to forge or alter.
- The cost of signing must be substantially lower than the cost of the data being signed.
- It must be possible to verify the authenticity of a signature without its author's cooperation.

Solution Alice and Bob agree on the use of a public-key digital-signature protocol (Appendix A). In most such systems, Alice applies the decryption algorithm to a message using her private key and sends the result (her signature) to Bob. He then encrypts the signed message with Alice's public verification key. If the result makes sense to Bob, that is, if Bob recognizes in the resulting data what he expected to be the original message, then it must be true that Alice is the sender of that message. This is the case, since only the knowledge of the key used by Alice in the signature generation procedure could have produced that signature.

Consequences

- The verification of a message's signature is based on the secrecy of the author's key and the strength of the signing algorithm. Thus, Alice could presumably deny the authorship of an old message by claiming loss or theft of her private key.
- Signatures are usually as large as the data being signed, sometimes producing an intolerable overhead.

Resulting Context Bob can now prove to a third party that a message he has received came indeed from Alice. Data integrity and sender authentication are implicit in the use of digital signatures. However, signatures are as large as the data being signed and often even larger. A more efficient approach would be to sign a much smaller fingerprint (the hash value) of a message, instead of the message itself and send the signed fingerprint along with the message. This is exactly what is provided by the *Signature with Appendix* pattern. Finally, encryption can be added to the signing process giving rise to the *Secrecy with Signature* pattern.

Implementation

- 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 make public keys for signature verification broadly available is necessary.
- For efficiency purposes, it is often preferred to sign the hash value rather than the message itself.

Example This pattern is used in *PayPerClick* in two situations of sender non-repudiation: cash issuing by a Broker and receipt issuing by a Payee. In the first case, a Broker produces a cash amount, signs it and sends the signed cash to a Payer, which verifies the Broker's signature. In the second case, a Payee verifies the Broker's signature in coins received from the Payer before issuing the receipt.

Known Uses Electronic commerce applications use digital signatures to authenticate both customers and merchants [FD98]. Digital signatures can also be used to guarantee authenticity and non-repudiation of information obtained over the Internet [HN98]. Both Privacy-Enhanced Mail [Lin93] and Pretty Good Privacy [Zim95] provide non-repudiation of electronic mail based on digital signatures.

Pattern 6: Secrecy with Integrity

Context Alice and Bob exchange encrypted messages, and they want to verify the integrity of the exchanged messages. Alice and Bob do not share cryptographic keys for purposes other than encryption.

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

Applicability

- In the detection of occurrence of errors in either transmission or storage of secret data.
- In the detection of unauthorized modification of secret data.
- When it is necessary to generate "fingerprints"for either secret messages or secret data records.

Forces

- It is desirable that the integrity of secret information can be verified without disclosure of the information.
- Granting secrecy and data integrity at the same time should not happen at the expense of one or the other. For instance, it should not be any easier to decipher a message in the presence of its MDC than it is without it.

Solution Two basic patterns are combined to solve this problem: *Information Secrecy* and *Message Integrity*. The MDC is computed over the original non-encrypted message which is then encrypted and sent, along with the MDC, to Bob. This pattern requires only one public/private key pair (or a shared secret key) used for encryption purposes.

Consequences

- Malicious replacements of messages can still garble valid data, thus rendering it useless after decryption.
- The computation and verification of MDCs may cause a noticeable decrease of performance.

Resulting Context Alice and Bob combine MDC generation functions and encryption in such a way that they preserve the integrity of an encrypted message without loss of secrecy.

Implementation

- This pattern can be implemented by computing the message's MDC either before or after encryption. In the first case, transmission errors can be detected before decryption. In the second, when the message structure is unknown, small transmission errors can only be detected after both decryption and MDC verification.

Example If a Payer's encrypted card number arrives corrupted at the Broker, it will not be decrypted successfully. Thus, the Broker should have the ability to detect the corruption of an encrypted message, to prevent the acceptance of a wrong but perhaps valid card number. So, during a *PayPerClick* cash request, the Payer should compute the MDC of the card number, then encrypt the card number, and send both the MDC and encrypted number to Broker.

Known Uses Privacy-Enhanced Mail [Lin93] protocols provide both encryption and message integrity for electronic mail.

3.3.6 Pattern 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 may 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 sender of an encrypted message without loss of secrecy?

Applicability

- When the occurrence of errors during transmission or storage of a secret must be detected.
- When detection of corruption or unauthorized modification of secret data is necessary.
- When it is necessary for both Alice and Bob to certify the origin of exchanged messages that were encrypted using public-key algorithms.

Forces Similar to the Secrecy with Integrity pattern.

Solution We combine two basic cryptographic patterns to solve this problem: *Information Secrecy* and *Sender Authentication*. The MAC should be computed over the original non-encrypted message. Both the encrypted message and the corresponding MAC are sent to Bob. The secret key used to compute the MAC must, of course, be different from the public key 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 the decryption processes in order to compute and verify a MAC, which can affect the system's performance.

Resulting Context Alice and Bob combine MAC generation functions and encryption in such a way that they not only preserve the integrity, but also guarantee sender authentication of an encrypted message without loss of secrecy.

Implementation

- If Alice and Bob use secret-key cryptography for encryption, then Sender Authentication is redundant and useless, except for granting an extra degree of security.
- As with *Secrecy with Integrity*, this pattern can be implemented by computing MAC before or after message encryption.

Example In a *PayPerClick* cash request, if the public key is used for card number encryption, then the Payer can use this pattern to ensure sender authentication vis-à-vis the Broker.

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

3.3.7 Pattern 8: Secrecy with Signature

Context Alice and Bob exchange encrypted messages, but they cannot prove the authorship of such messages. Moreover, Eve can modify, replace, or insert messages in the communication channel in such a way that Alice and Bob cannot detect these spurious messages. We assume that Alice and Bob already share keys for secrecy purposes.

Problem How can Bob prove to a third party the authorship of Alice's encrypted messages without loss of secrecy?

Applicability

- When non-repudiation of a secret is desired.

Forces Similar to *Secrecy with Sender Authentication* and *Secrecy with Integrity* patterns.

Solution We combine two basic cryptographic patterns to address 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

- Signatures provide a proof of authorship of encrypted messages. However, the cost of signing long messages may become intolerably high.

Resulting Context Alice and Bob combine mechanisms of digital signatures and encryption achieving non-repudiation of secret messages and, implicitly, sender authentication and corruption detection, of such messages. However, the resulting signatures are at least as large as the data being signed. When possible, the *Secrecy with Signature with Appendix* pattern should be used, providing a more efficient solution, since the signing procedure is applied on the "fingerprint" of the encrypted message.

Implementation

- Different keys should be used in encryption and signing purposes.

- As before, this pattern can be implemented in two different ways, according to the order in which encryption and signature are computed on the message. When encryption is applied first, verification of the signature can only be done after decryption, since, in principle, signatures have no apparent structure. This apparent difficulty can be easily overcome by attaching to the encrypted message a known header before signing. If the signature is applied first to the non-encrypted data, then signature verification must expose the encrypted content. This may be unacceptable when different parties are responsible for decryption and signature verification. Usually, a better strategy is to use Secrecy with Signature with Appendix.

Example When sending credit card numbers over the Internet, a user wishes it to remain secret. At the vendor's side there is the need for that number to be tied to the correct user, in a non-repudiable fashion.

Known Uses Both Privacy-Enhanced Mail [Lin93] and Pretty Good Privacy [Zim95] combine encryption and digital signatures for electronic mail.

3.3.8 Pattern 9: Signature with Appendix

Context Alice and Bob sign exchange signed messages. However, they not only have limited resources for both storage and processing, but also the messages they exchange are very large and produce large signatures.

Problem How can memory requirements for signatures be reduced while increasing the performance of the digital signature protocol?

Applicability

- When a message can be kept separate from its signature.
- When space and time requirements for the digital signature protocol are tight.

Forces Similar to those of the Signature pattern.

Solution Two patterns are combined to solve this problem: *Signature* and *Message Integrity*. The resulting 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 at least 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.
- The combination of weak MDCs and signatures can potentially decrease the security of digital signature protocols.

Resulting Context Alice and Bob reduce their time and memory requirements by reducing the size of the data to be signed. Encryption mechanisms can be included in the signing process to instantiate the *Secrecy with Signature with Appendix* pattern.

Example In *PayPerClick*, non-repudiation of a receipt could be achieved by signing it. However, using the *Signature* pattern by itself in each node of the hyper-document's tree is not practical. Even if *Signature with Appendix* is computed for each tree node, this computation may produce a large receipt. However, signing a single fingerprint of a hyper-document's tree, as in *Signature with Appendix*, is a much faster procedure. The resulting receipt is attached to the corresponding purchased hyper-documents.

Known Uses When the user of an Internet application must digitally sign information, small signatures should be favored [CGHK98]. An example of this are signed applets: Java Development Kit uses *Signature with Appendix* to produce small signatures for large amounts of code [JBK98].

3.3.9 Pattern 10: Secrecy with Signature with Appendix

Context Alice and Bob exchange encrypted signed messages in order to achieve secrecy and non-repudiation. They possess limited storage and processing resources, and the messages they exchange are large.

Problem How can one reduce the amount of memory necessary to store a message's signature, while increasing system performance, without loss of secrecy?

Applicability

- When secret data may be separated from its signature.
- When the digital signature protocol operates in limited resource environments.

Forces Similar to previous pattern combinations.

Solution Two patterns are combined to address 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 its signed hash value are sent to Bob. He deciphers the encrypted message with his decryption key and verifies the signature of the hash value using Alice's verification key. Bob then computes a new hash value of the message and compares it with that received from Alice. If they match, the original message is correctly signed.

Resulting Context Alice and Bob not only achieve both secrecy and non-repudiation in their communication, but also reduce the amount of time and memory required for signatures.

Consequences

- The inclusion of a hash computation in a procedure that already has two processing phases may seem a difficult decision to make. However, hash computations are among the fastest in cryptographic software. Moreover, the reduction in signing/verification time and space certainly compensate for the hashing overhead.

Implementation

- This pattern can be implemented either by signing the message's MDC before message encryption or by signing an encrypted message's MDC directly.

Example Electronic forms usually contain some sensitive information that requires both confidentiality 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, and so forth. The use of digital signatures to guarantee non-repudiation of such data can potentially result in large signatures. *Secrecy with Signature with Appendix* solves this problem with a substantial gain in performance.

Known Uses Digital signatures for electronic mail, alone or in combination with encryption, are provided by Privacy-Enhanced Mail [Lin93] and Pretty Good Privacy [Zim95] using signatures with appendix.

3.4 Deploying the Cryptographic Pattern Language

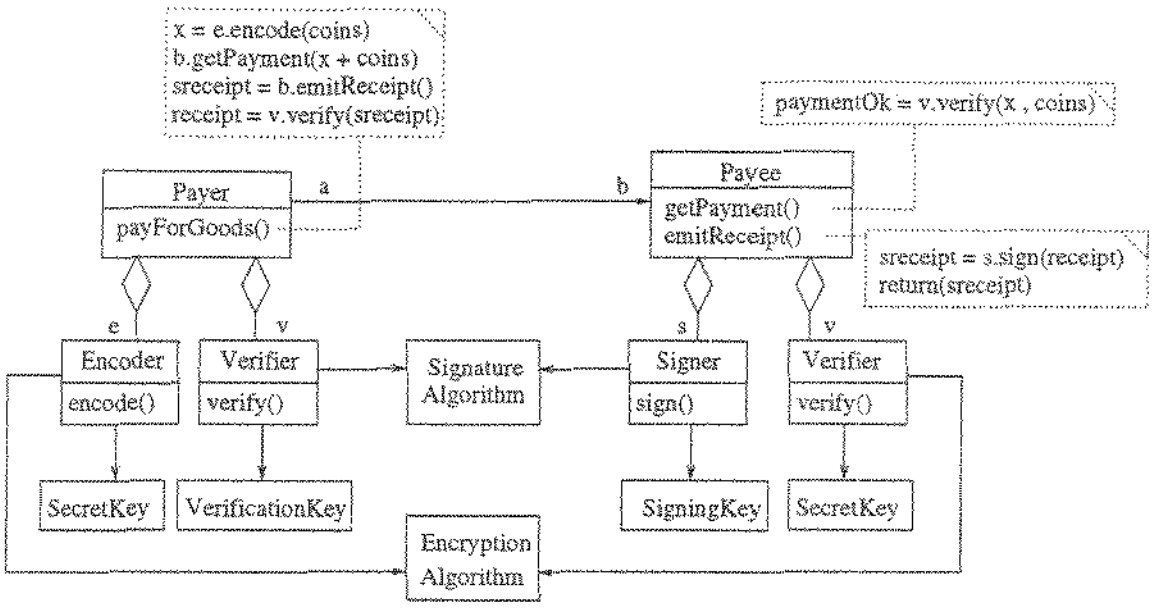

Figura 3.5: Class Diagram for the Payment Transaction.

Figures 3.5 shows the class diagram for a *PayPerClick* payment transaction, using *Sender Authentication* and *Signature*. The first pattern authenticates the sender of the payment; the second signs the receipt. Consider a payment scenario, which illustrates the use of these patterns:

Let a and b be two objects, instances of the `Payer` and `Payee` classes, respectively. The encoder of a has been initialized with a secret key shared with b, and its verifier has been initialized with b's public key. Likewise, b's verifier and signer have been previously initialized with the shared secret key and with b's private key. The following sequence of events complete the scenario (Figure 3.6):

1. a uses the Encoder e to compute a MAC x of his/her coins.

2. a sends the coins along with x as payment to b, who uses its Verifier v to check the validity of x, thus authenticating the sender of the coins, namely a.

3. a requests a receipt from b, who uses the Signer s to generate a signed receipt, sreceipt, and returns it to a.

4. a verifies sreceipt using the signature verifier.

The Java code in this Section corresponds to a *PayPerClick* transaction and uses Java Cryptographic Architecture (JCA). Classes `SecureRandom`, `Signature`, `SignedObject`,

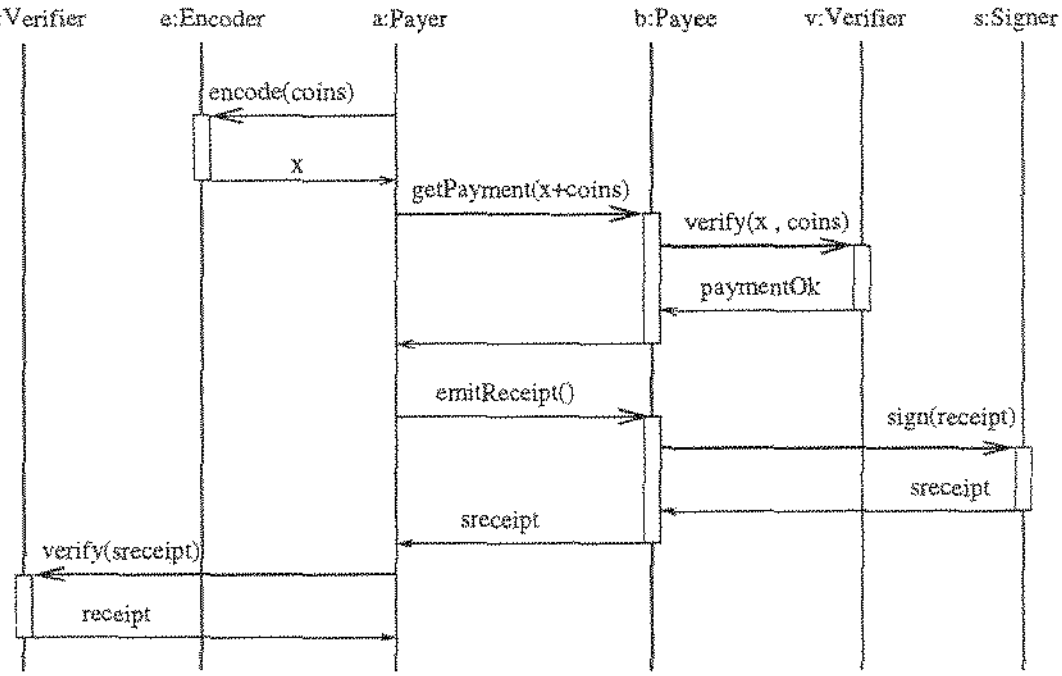

Figura 3.6: Sequence Diagram for the Payment Transaction.

`PublicKey`, and `PrivateKey` are provided by JCA. Classes `Cipher`, `SealedObject`, and `SecretKey` are supported by Java Cryptographic Extension (JCE). Information about the Java cryptographic API can be found in [Knu98].

Classes `Signer` and `Verifier` (Sample Code 3.4.1) perform signing and verification, respectively. A `Signer` should be initialized with a `PrivateKey` and a `Signature` engine. A `Verifier` uses a `Signature` engine and a `PublicKey`. The method `sign()` in the `Signer` class returns a `SignedObject` containing the object being signed and its digital signature. The method `verify()` of the `Verifier` class takes as input a `SignedObject` and returns true if the verification succeeds.

The `Payer` class (Sample Code 3.4.2) contains a `Signer s`, which signs payments, an `Encipher e`, which encrypts credit card numbers, and two instances of `Verifier`: one, `vPayee`, verifies payment receipts issued by a `Payee`; the other, `vBroker`, verifies cash issued by a `Broker`. An instance of `Vector` is a simple electronic wallet. `Payer` has two methods, `payForGoods()`, which performs a payment to `Payee` and requests a signed receipt, and `cashRequest()`, which asks the `Broker` for money. Method `payForGoods()` returns a `SignedObject` containing a payment of amount coins to a `Payee`. The required amount of coins is removed from the `wallet`, the payment is signed and sent to the `Payee`, from whom a receipt is requested. A payment transaction succeeds if the payment succeeds and the receipt is authentic. Method `cashRequest()` asks the `Broker` for an amount of electronic money, which should be charged to the `Payer`'s credit card. The

Class 3.4.1 Classes Signer and Verifier

```
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));}
}
```

number of the Payer's card is sent to the Broker in a SealedObject. A SignedObject containing cash is received, verified, and credited to the Payer's wallet.

Class Payee (Sample Code 3.4.3) contains a Signer s, used to sign receipts and two instances of Verifier. One, vPayer, is intended for verification of payments signed by the Payer; the other, vBroker, for verification of single coins issued and signed by the Broker. Payee has two methods, issueReceipt(), which issues a signed receipt, and getPayment(), used to verify and check payments and coins. Method issueReceipt() returns a SignedObject, which contains the number of valid coins received since the issuing of the last receipt. This implementation does not consider the purchased goods for which this receipt is being issued. A better solution should contain the fingerprint of the purchased document. Method getPayment() takes a SignedObject and verifies whether it is a valid payment with valid coins in it. The Payer verifier checks payments; the Broker verifier checks coins. The method returns true if all these verifications succeed.

The Broker class (Sample Code 3.4.4) contains a Decipher, which decrypts the Payer's card number, and a Signer s, which authenticates cash issued by Broker. It has two methods: getCreditCard(), which receives a SealedObejct containing the Payer's encrypted card number, and issueCash(), used to generate an amount of coins. In method issueCash() an amount of cash is a Vector in which each coin is a SignedObject

Class 3.4.2 Class Payer

```
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 b, 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 &= b.getPayment(s.sign((Serializable) payment));
        receipt = b.issueReceipt();
        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.issueCash(amount);
        ok &= vBroker.verify(o);
        if(ok) wallet = (Vector) o.getObject();
        return(ok);
    }
}
```

Class 3.4.3 Class Payee

```
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 issueReceipt(){
        String str = (coinCounter.intValue() ≠ 1?"s":);
        String receipt = "I received " +
            coinCounter.toString() + "coin"+str+
            "from You.  Since last receipt was issued.";
        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);
    }
}
```

Class 3.4.4 Class Broker

```
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 issueCash(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));
    }
}
```

containing a random value. Another `SignedObject` contains the whole cash amount.

3.5 Conclusions

Cryptography-supported security facilities are becoming a standard feature in many modern applications. To facilitate the design, implementation, and reuse of cryptographic software, the architectural aspects of cryptographic software 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. First, the patterns represent not only the overall architecture of typical cryptosystems, but also all the valid combinations of the four basic cryptographic mechanisms. Second, the cryptographic patterns are widely used in many applications [HY97, Her97, CGHK98, HN98, Lin93, Zim95] and are supported by many cryptographic APIs [JBK98, Kal95, css97]. However, other auxiliary patterns and pattern languages, supporting infrastructure 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.

Capítulo 4

A Reflective Variation for the Secure-Channel Communication Pattern

4.1 Introduction

In many applications, cryptography-based security is a non-functional requirement, those requirements related to *how well* an application accomplishes its purpose [SW96]. Security, distribution and fault tolerance are other examples of non-functional requirements which are usually independent of application functionality. The widespread use of cryptographic techniques and the present interest and research on flexible/extensible software architectures led us to a reflective object-oriented approach for the design of cryptographic components. This approach allows the explicit separation of functional and (non-functional) cryptographic requirements of object-oriented applications.

The use of computational reflection in object-oriented programming is not new [Mae87], neither is the use of meta-object protocols in the implementation of non-functional requirements of object-oriented applications [SW96]. The encapsulation of authentication facilities and their composition to fault tolerance and distribution, in client-server applications using a meta-object protocol, were proposed by Fabre and Pérennou [FP96]. Meta-Object protocols for cryptographically secure communication are a recurring solution for the insecure communication problem in reflective software architectures, and can be abstracted and formally specified as architectural connectors [WS98a]. Software architectural styles are composed of connectors and components [SG96].

This work presents *Reflective Secure-Channel Communication*, a refinement of the generic object-oriented cryptographic architecture proposed in [BRD98b, BRD98a], in order to decouple objects responsible for cryptographic services from the application objects.

The contribution of this work is the proposal of a design pattern obtained by the combination of *Secure-Channel Communication* [BRD98a] pattern and *Reflection* [BMR+96, 193] architectural pattern. *Reflective Secure-Channel Communication* is useful in two situations: (*i*) during design of general purpose application with non-functional cryptography-based security requirements; (*ii*) in addition of cryptography-based security to third-party commercial-off-the-shelf components and applications. In fact, the pattern proposed here was used to design a reflective object-oriented framework based on a meta-object library for cryptography [BDR99a]. The diagrams in this paper are presented using Gamma *et al.'s* notation [GHJV94]. Those readers interested in cryptography techniques should take a look at [Sch96, MvOV96, Sti95].

4.2 Reflective Secure-Channel Communication Pattern

Context We have proposed a pattern language for cryptographic software which is composed by a set of ten design patterns [BRD98a]: *Secure-Channel Communication*, *Information Secrecy*, *Sender Authentication*, *Message Integrity*, *Signature*, *Secrecy with Sender Authentication*, *Secrecy with Signature*, *Secrecy with Integrity*, *Signature with Appendix*, and *Secrecy with Signature with Appendix*. These patterns document the experience and the expertise of practitioners in designing cryptographic services, such as secrecy, integrity, authentication and non-repudiation, for secure communication and storage applications. These patterns share the same structure and dynamic behavior. This aspects can be abstracted in a generic cryptographic architecture, which is stabilished by the foundation pattern, *Secure-Channel Communication*. However, these patterns do not explicitly capture the design of cryptographic services as non-functional requirements.

Figure 4.1 shows this generic structure defining two template classes, Alice and Bob, which are application classes, and two hook classes, Codifier and Decodifier, which are cryptography-aware classes. Class Codifier has a hook method $f()$, which performs cryptographic transformations. The class Decodifier defines 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. The objects' interaction diagram is shown in Figure 4.2.

A limitation of this design is that it forces functional objects (instances of Alice and Bob) to explicitly take care of non-functional (cryptography-aware) objects. That is, Alice and Bob reference cryptographic objects and decide when a cryptographic transformation should take place. This highly coupled design has the following disadvantages:

- It limits the reuse of Alice and Bob.

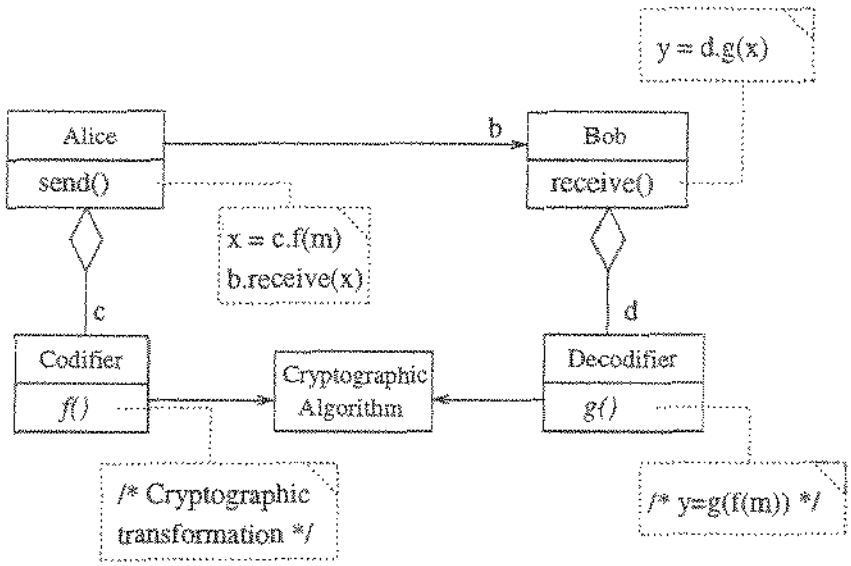

Figura 4.1: Secure-Channel Communication Structure.

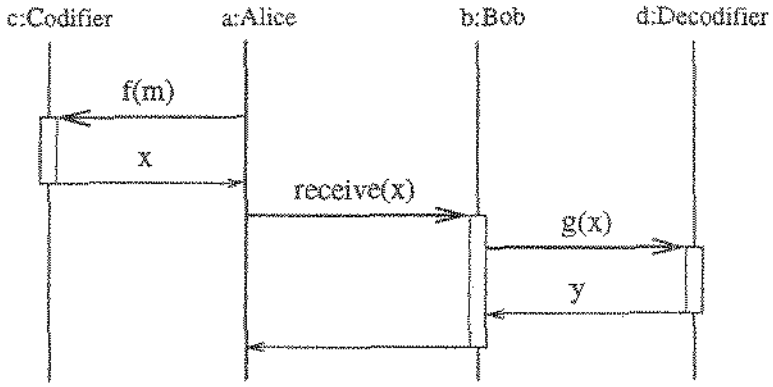

Figura 4.2: Secure-Channel Communication Dynamics.

- It pollutes application objects with explicit references and method invocations of non-functional cryptography-aware objects, reducing readability.
- It requires some background on cryptography from application programmers.

Applicability

- When cryptography-aware objects address non-functional application requirements.
- When reuse of functional objects should be facilitated.
- When the separation of concerns between functional and non-functional aspects should be made explicit.

Problem How could the separation of concerns between application functional objects and cryptography-aware objects be explicitly represented in a way that reuse and readability can be improved? In other words, can cryptography-based security be added transparently to third-party applications or components, even if source code is not available?

Forces

- Cryptographic services are usually non-functional requirements related to communication and persistence requirements, but orthogonal to these. Leaving application responsibilities decoupled from security services facilitates reuse and security policy changes, and frees application programmers from having to acquire cryptographic knowledge.
- The explicit separation of concerns can lead application designers to: (*i*) procrastination of important security policy decisions in cryptography-aware applications; (*ii*) lack of control over cryptographic features, from the application programmers' point of view.
- Delegation of cryptography-aware decisions has the advantage of encouraging the utilization of largely tested (cryptanalyzed) components. However, it can also expose application functions and sensitive data to third party's Trojan horses.

Solution In order to overcome the limitation stated in the Section "Context", a restructuring of the interaction mechanism among objects can be used. Meta-object protocols with message interception mechanisms can potentially invert the dependencies among non-functional objects and functional ones, in a way that non-functional requirements are

transparently accomplished by non-functional objects, which may not be known by the application functional objects.

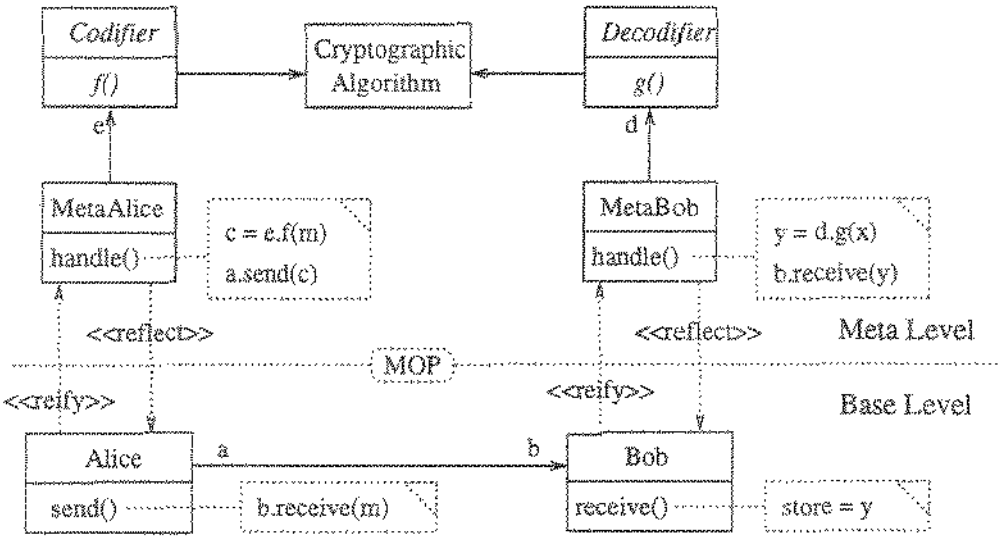

Figura 4.3: Reflective Secure-Channel Communication Structure.

The use of a meta-object protocol explicitly separates cryptographic requirements from application functionalities. Figure 4.3 addresses *Secure-channel Communication* in a reflective way. Classes MetaAlice and MetaBob are responsible for cryptographic method calls and for the re-sending of base-level methods, which were previously intercepted. Figure 4.4 shows the interaction diagram. For instance, method send() is intercepted by MOP's reflective kernel and materialized in a send-operation object. This operation object and its argument, m, are treated by the meta-object, ma, which requests the cryptographic transformation accordingly. The intercepted method is, then, re-sent (containing now the encrypted argument, c) by MOP's kernel to its original target. The same happens with method receive().

Consequences This design has the following advantages:

- Decoupling of functional objects from non-functional ones in a way that application objects do not need to know either what kind of (cryptographic) transformation is taking place or what kind of security requirements are being accomplished (confidentiality, integrity, authentication, non-repudiation, or some apropriate combination of these).

- Separation of cryptographic objects from application objects so that it is potentially possible to understand application code without cryptographic background.

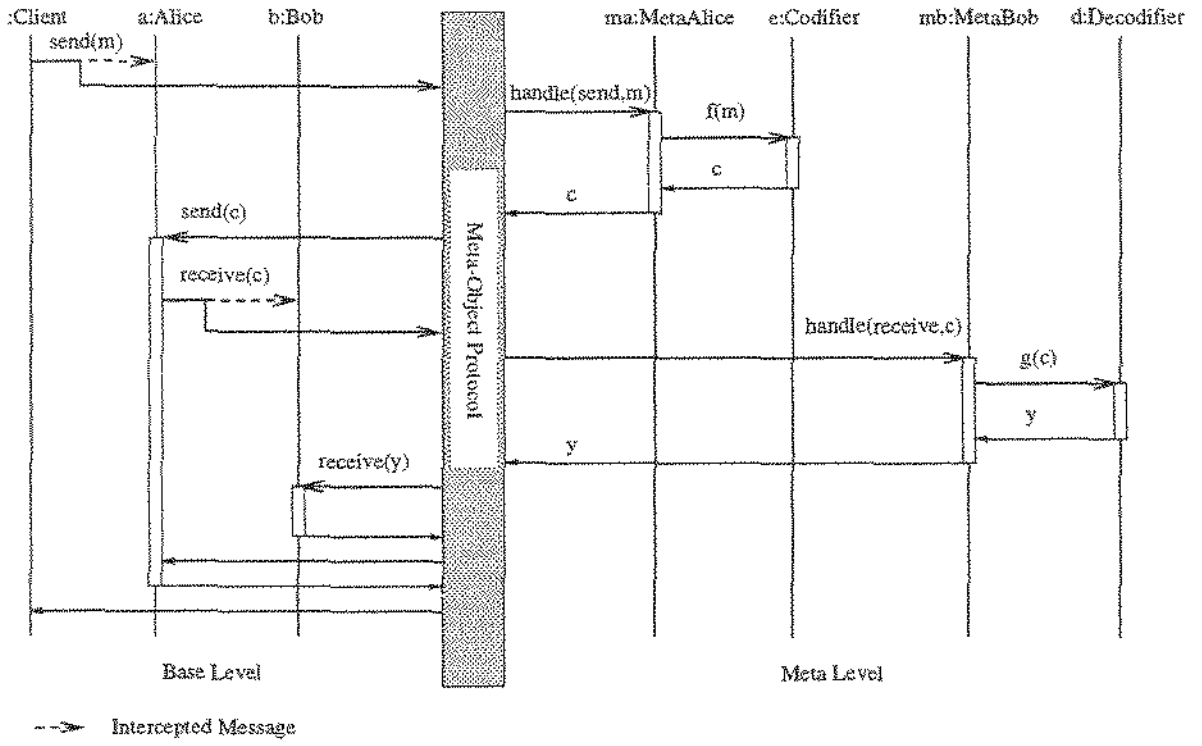

Figura 4.4: Reflective Secure-Channel Communication Dynamics.

- Development and testing of cryptographic components can be done only once, separately, for highly reused components.

Its main disadvantage is a potential decrease of performance, for two reasons:

- A relatively large number of method calls, due to a larger number of indirections in code.
- A time delay due to the method interception mechanism.

A minor disadvantage is the larger number of objects within the whole application. Cryptographic algorithms are usually implemented within methods. If cryptographic transformations are performed faster enough, small losses of performance, due to method invocation and interception, can be negligible.

Implementation Factors

- The *a priori* negotiation, concerning the usage and agreement of cryptographic services and the generation, exchange and storage of keys, may or may not be handled at the meta-level. This decision depends on the degree of control over the

cryptographic services the application programmer intends to have. For instance, application programmers may be interested on what kind of service is being used at a given moment, maintaining the ability of turning the security aspects of the channel on and off.

- The tower of meta-objects [Mae87] can be as high as the number of non-functional requirements. The decision concerning which position cryptography will occupy in this tower is not simple. Aspects such as requirement composition or chaining must be considered carefully. For instance, since cryptography is orthogonal to persistence and communication, which can, in turn, stay at the meta-level, cryptography should be accomplished at the meta level of these, that is, at a meta-meta level. However, if fault tolerance is another requirement, it can be accomplished either above or below encryption [FP96].

- The number of cryptographic meta-objects may vary among three main possibilities: (*i*) a single meta-object responsible for encryption and decryption: this solution works if Alice and Bob share the same address space; (*ii*) two meta-objects, one instance of MetaAlice, associated to a method Alice.send(), and one instance of MetaBob, which treats method Bob.receive(), recommended for secure communication; (*iii*) at least as many meta-objects as the number of Alice and Bob instances. The MOP's ability to manage the need for distinct simultaneous instances of Encoders and Decoders, potentially initialized with keys used for different purposes, in order to simultaneously keep track of channels with different degrees of security, determines the final number of meta-objects. How easily this task can be accomplished depends on the MOPs ability for meta-object composition.

- There are situations in which the result of an (intercepted) operation should be encrypted, authenticated or verified for non-repudiation. How easily this can be accomplished depends on the flexibility of the meta-object protocol. MOPs offering features for both result interception and modification facilitates transparent secrecy as well as authentication of results.

- There are two approaches for adding cryptography-based security to third-party components: (*i*) performing behavioral changes dynamically, on-the-fly, over executable (byte) code; in this case, a hook (well known) interface [Wel97] must be available; (*ii*) working over the source code, potentially performing some preprocessing.

- The meta-level application can work as an object-oriented framework [Lew96, Pre95] and the inversion of control, which characterizes frameworks, takes place [BDR99a].

Example Modern software systems are being modeled according to architectural styles [SG96], which consist of groups of components glued together by connectors, according to some criteria. Commercial-off-the-shelf (COTS) applications and components usually present legal and practical obstacles in accessing their source code, these obstacles restrict component flexibility. However, in component-based applications, it is often necessary either add features to or modify the behavior of COTS. For instance, cryptography-based security can be added to a COTS communication component in order to transparently modify its behavior and provide confidentiality, integrity, authentication and non-repudiation, without COTS modification. In this situation, a MOP can be used as the architectural connector that glues a cryptographic component to the COTS communication component.

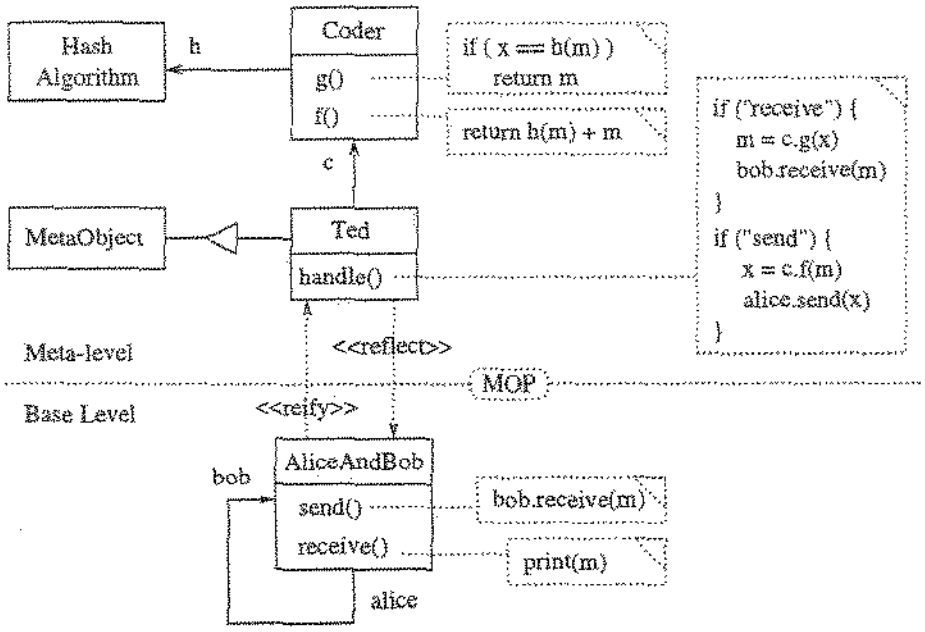

Figura 4.5: Ted Structure.

The diagrams on Figures 4.5 and 4.6 show the structural and dynamic models for a simple program implemented in Section "Sample Code". The meta-level application, Transparent error detection (Ted), is used to modify the behavior of the base level application, AliceAndBob. The MOP transparently add cryptography-based integrity to data, exchanged through method calls, in the base level. Ted does not need to access AliceAndBob source code. However, in this case, it requires at least a known (hook) interface to be accessed by the MOP. In this example, this interface is based on (static) class methods.

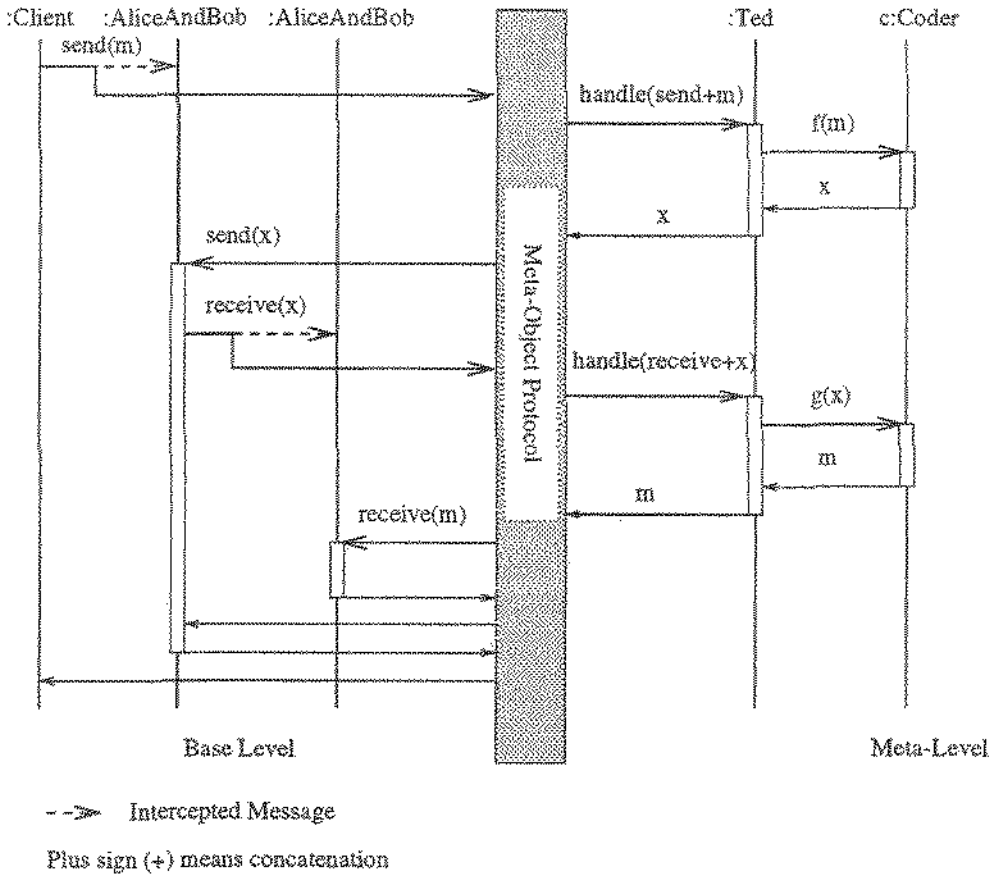

Figura 4.6: Ted Dynamics.

Class 4.2.1 UncorruptedObject

```
class UncorruptedObject implements Serializable {
    public UncorruptedObject(Serializable object, MessageDigest hashEngine);
    public final Object getObject();
    public final boolean verify(MessageDigest hashEngine);
}

class Coder {
    public UncorruptedObject encode(Serializable o);
    public Object decode(Serializable o);
}
```

Sample Code The following Java code corresponds to a simple program, Ted, that adds cryptography-based modification detection facilities to another program, AliceAndBob. Ted works over AliceAndBob bytecode. It is based on a hook interface of AliceAndBob's static methods, though. Ted uses *Guaraná* [OGB98], a meta-object protocol for Java. Ted is activated by typing, in a unix shell, the command line: `% guarana Ted AliceAndBob`. *Guaraná* interprets Ted which takes AliceAndBob as an argument. Ted was written with *Guaraná* MOP in mind and executed by the *Guaraná*. On the other hand, AliceAndBob is a common Java class file which is used without any modification.

Classes `UncorruptedObject`(Class 4.2.1), `Coder` and `Ted` belong to the meta-level application. The public interfaces for those classes are shown below. `UncorruptedObject` encapsulates a serializable object and its fingerprint, computed by a `MessageDigest` engine from the package `java.security`. Method `verify()` checks the object's fingerprint, and method `getObject()` returns the original object. `UncorruptedObject` is analog to class `SignedObject` from `java.security`. Class `Coder` emcapsulates the creation of `UncorruptedObjects` in method `encode()`; fingerprint's verification and object recovery are in method `Coder.decode()`.

Class `Ted`(Class 4.2.2) extends *Guaraná*'s `MetaObject` and has two interesting methods: a `handle()` for (reified) intercepted `Operations` and `main()`, responsible for base-level class loading and association to meta-objects. Method `handle()` obtains the method name from the reified `Operation`, taken as parameter, and tests it in order to determine if it is either a `receive()` or a `send()` call. In the first case, the argument is encoded; in the second, it is decoded. After that, the `Operation`, now containing the modified argument, replaces the old one and is invoked.

Method `Ted.main()`(Class 4.2.3) loads a class, which name was passed as an argument, looks for its `main()` method and uses `Guarana.reconfigure()`, a call to the reflective kernel, to turn a Ted's instance into the primary meta-object of the loaded class. Since

Class 4.2.2 Ted

```
public class Ted extends MetaObject {
    public Result handle(final Operation op, final Object ob) {
        Operation replace = null;
        switch (op.getOpType()) {
        case Operation.methodInvocation:
            try {
                Object[] args = op.getArguments();
                Serializable arg0 = (Serializable) args[0];
                String s = op.getMethod().getName();
                System.out.println("Method "+s+"interceped.");
                if (s.equals("receive")) arg0 = (Serializable) c.decode(arg0);
                if (s.equals("send")) arg0 = (Serializable) c.encode(arg0);
                args[0] = (Object) arg0;
                replace = opfact.invoke (op.getMethod(), args, op);
                replace.validate();
            }
            catch(Exception e) {e.printStackTrace();System.exit(0);}
            Result res = Result.operation(replace,Result.noResultMode);
            return res;
        }
        return Result.noResult;
    }
```

Sample Code 4.2.3 Method Ted.main()

```
    public static void main(String[] argv){
        java.lang.Class c = Class.forName(argv[0]);
        java.lang.reflect.Method m = c.getMethod("main", new Class[] { String[].class });
        Guarana.reconfigure(c, null, new Ted());
        m.invoke(null, new Object[]{argv});
    }
```

Class 4.2.4 AliceAndBob

```
public class AliceAndBob{
    public static void send(Serializable o, AliceAndBob bob)
    { System.out.println(bob.receive(o));}

    public static Serializable receive(Serializable o)
    { return("I received:  "+o+", is it ok?");}

    public static void main(String[] argv){
        send("This string must not be corrupted",new AliceAndBob());
    }
}
```

Ted's instance is associated to a class, not to instances, only static (class) method calls can be intercepted. Finally, the `main()` method of the loaded class is invoked.

Class `AliceAndBob`(Class 4.2.4) is the base-level applicaction. It has three `static` methods: `main()`, `send()` and `receive()`, which are not cryptographically secure. Thus, the target of a `receive()` message cannot determine whether the object received was corrupted or eavesdropped.

An interesting feature of this example is the inversion of control over the main execution flow; that is, Ted is the main program which loads AliceAndBob. In fact, Ted works as a small object-oriented framework [Lew96, Pre95] for adding cryptography-based error detection to AliceAndBob-like applications. This framework can be extented in order to not only offer other cryptographic services, but also cover a broad range of object-oriented applications [BDR99a].

Known Uses *Friends* [FP96] is a reflective software architecture for implementing fault tolerance and authentication to object-oriented applications, which uses a meta-object protocol for authentic communication. Transparent addition of security features to third-party (off-the-shelf) Java componentes, based on a reflective architecture, is another interesting application of this pattern [WS98b, Wel97].

The ideas present in the sections "Example" and "Sample Code" can be extended to other cryptographic services. We have used this pattern during the realization of the basic design features of a reflective object-oriented framework based on a meta-object library for cryptography-based security [BDR99a], which focuses on three points: easy reuse of cryptography-aware code, easy composition of cryptographic services and transparent addition of cryptography-based security to third-party code. The framework is applicable to not only third-party commercial-off-the-shelf applications, but also legacy systems. In

this framework, instances of MetaAlice and MetaBob are especialization of an abstract MetaLevelApp class, which offer hooks as in the *Template Method* [GHJV94, 325] pattern.

The main goal of this meta-object library is to provide base-level applications with reusable cryptography-based security features in which addition and composition of cryptographic services are transparent, from the point of view of the base-level programmer. This powerful approach allows the addition of cryptography-based security to (Java) applications even when source code is not available. This meta-object library acts in the realm of object communication in such a way that data exchanged among communicating (potentially distributed) objects through method calls are transparently secured. The reflective object-oriented framework provides the proper inversion of control in order to assure that cryptographic code is not known by base-level applications.

This pattern can also be used recursively. For example, secure meta proxies can be used to protect objects for cryptographic keys, handled in meta level, against corruption and unauthorized copy [BDR99a].

Related Patterns *Reflective Secure-Channel Communication* is a refinement of Tropyc's *Secure-Channel Communication* [BRD98a] obtained by combining the later and the *Reflection* [BMR+96, 193] architectural pattern.

4.3 Conclusions and Future Work

Although our cryptographic design patterns were proposed with the intent of facilitating design reuse, practice has shown that the high coupling, due to the use of explicit references, among cryptography-aware objects and functional objects leads to both reduction of application objects reuse and decrease of design understandability. Specific applications, specially those in which cryptography plays a non-functional role, could benefit from a combination with computational reflection mechanisms in a way that both readability of application code and components reuse are increased. The composability of cryptographic mechanisms, such as confidentiality, integrity, authentication and non-repudiation, is also facilitated by a meta-object protocol in which meta objects could be easily composed. The reflective variations of our cryptographic design patterns can be used to document not only the usage [Joh92], but also the design (for example, when self-securing cryptographic keys) of a reflective cryptographic framework for secure object-oriented applications.

Capítulo 5

A Meta-Object Library for Cryptography

5.1 Introduction

Fields such as computer networking, distributed systems, electronic messaging and browsing have strong security concerns in granting integrity, authentication, non-repudiation and confidentiality. Modern cryptography is used in applications such as electronic commerce systems, legacy systems, not originally developed with security features, and software systems in which cryptography-based security plays a non-functional role. In order to facilitate the reuse of flexible and adaptable cryptographic software in such an heterogeneous environment, the architectural aspects of cryptographic components, the design patterns that emerge from them and the gluing techniques for the combination of security-aware components with commercial-off-the-shelf ones should be considered.

This work presents a Meta-Object Library for Cryptography (MOLC for short). The main goal of this library is to provide base-level applications with reusable cryptography-based security features in which addition and composition of cryptographic services are transparent, from the point of view of the base-level programmer. This powerful approach allows the addition of cryptography-based security to (Java) applications even when source code is not available. MOLC acts in the realm of object communication in such a way that data exchanged among communicating (potentially distributed) objects through method calls are transparently secured. MOLC was implemented in *Guaraná*, a meta-object protocol for Java, which is fully documented in a series of technical reports [OGB98, OB98a, OB98b, OB98c].

This text is organized as follows. Section 5.2 reviews the main cryptographic services and the role of cryptographic patterns. Section 5.3 approaches the main aspects of MOLC's design. The design issues in adding security to third-party applications are in

Section 5.4. The meta-level reconfigration policy is treated in Section 5.5. Section 5.6 outlines implementation issues. Conclusion and future work are in Section 5.7.

5.2 Cryptographic Services and Patterns

Modern cryptography addresses four security goals [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 ones and are the building blocks for security protocols. Confidentiality is the ability to keep information secret except from authorized users. Data integrity is used to guarantee that 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 his actions or commitments in the future.

Some combination of the basic cryptographic services are required in order to accomplish the security requirements of applications. We have proposed a pattern language for cryptographic software [BRD98a] which addresses the valid combinations of cryptographic services in the context of secure communication, when security aspects are so important that they cannot be delegated to the communication or persistence subsystems and are treated by the application itself. The cryptographic design patterns corresponding to the basic services and their valid compositions are summarized in Table 5.2. The *Secure-Channel Communication* pattern is an abstraction for the others' common aspects of behavior and structure.

Computational reflection techniques allow the explicit separation of concerns between functional and non-functional requirements of object-oriented applications. Software systems, specially those in which cryptography plays a non-functional role, could benefit from a combination with computational reflection mechanisms in such a way that both readability of application code and reuse of software components are increased. We have proposed in [BRD99a] a refinement for the cryptographic patterns in order to decouple objects responsible for cryptographic services from the application's objects. This approach is useful: (*i*) during the design of general purpose application with non-functional cryptography-based security requirements; (*ii*) in addition of cryptography-based security to either legacy systems or third-party commercial-off-the-shelf components. MOLC provides a set of meta objects whose main goals are the instantiation of the reflective

cryptographic patterns and the composition of cryptographic services.

#	Pattern	Purpose
1	Secure-Channel Communication	Provides a generic software architecture for cryptographic systems
2	Information Secrecy	Provides secrecy of information
3	Message Integrity	Detects corruption of a message
4	Sender Authentication	Authenticats the origin of a message
5	Signature	Provides the authorship of a message
6	Secrecy with Integrity	Detects corruption of a secret
7	Secrecy with Sender Authentication	Authenticates the origin of a secret
8	Secrecy with Signature	Proves the authorship of a secret
9	Signature with Appendix	Separates message from signature
10	Secrecy with Signature with Appendix	Separates secret from signature

Table 5.2: The Cryptographic Design Patterns and Their Purposes.

5.3 Meta-Object Library

The communication among objects can take place through either method calls or buffers. In the first case, method calls can be local or remote and references to objects can be direct or through proxies. In the second, buffers can be either persistent or transient. In these situations, cryptography-based security can be applied to both arguments and results of methods. A meta-object protocol can be used to provide transparent security to data exchange. For example, method calls and returned results are intercepted by the meta level and the operation's arguments or results are converted to some secure format according to the communication security requirements. The target object of the method call or returned result have to restore data to their original insecure format. Of course, meta objects must agree on cryptographic features, such as keys and algorithms, before the communication begins. A cryptography subsystem is supposed to be responsible for such an agreement.

The general software architecture for this meta-object library is shown in Figure 5.1. MOLC contains the flow of program's execution, which in turn contains the base-level application. Such a feature characterizes MOLC as an object-oriented framework [Pre95]. The cryptographic routines are obtained from a Java Cryptographic Service Provider [JBK98]. Because the cryptographic provider's routines are in a low level of abstraction, an adapter layer between MOLC and the Java Cryptographic Provider is necessary in

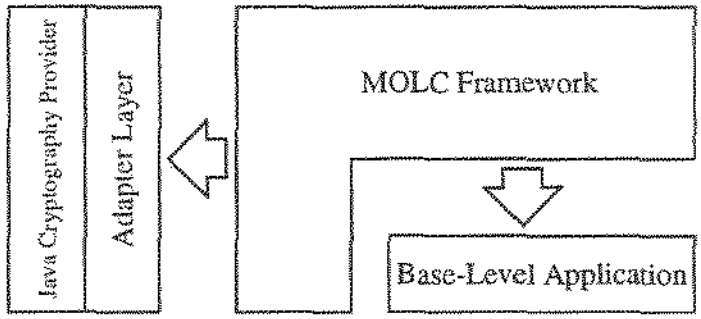

Figura 5.1: Architecture of MOLC.

order to reduce both the complexity of meta objects and the dependencies from particular implementations of cryptographic services. Besides offering meta objects for cryptographic transformations over base-level data, MOLC can also reflect about itself in order to secure its own data. An interesting example of such a recursive use of the reflective cryptographic pattern is the implementation of secure proxies for cryptographic keys as meta objects. Another possibility is self authentication of either compiled classes or distinct algorithm implementations.

5.3.1 Securing Keys with Meta Proxies

Keeping cryptographic keys securely stored in computer memories is always a problem. Protecting keys from unauthorized copy or modification is a difficult task because they are usually ordinary objects kept insecurely in computer memories. A step toward making key manipulation in memory less insecure is to reduce the time keys stay in memory as active objects. If keys only stay in memory as local variables of methods, the chances for unauthorized copy are greatly reduced because objects local to methods are usually deallocated at the end of method execution (or garbage-collected when unreachable) and the memory freed so that it has a greater chance to be used by another method's local data in a relatively short period of time. Particularly, Java objects are stored in the heap in an implementation specific format and Java local variables are kept into the method's stack, whose memory is released after the method's execution. Such features greatly reduce the chance of memory scans looking for sentivive data, but the risk still persists.

Protection proxies [GHJV94, 207] can be used to control access to cryptographic keys. We have implemented a meta object, called MetaKey, for proxing cryptographic keys, which are kept encrypted in persistent storage and whose contents are supposed to be decrypted only for use in the innermost methods as a local variable. We used *Guaraná*'s facilities for creating proxies and associating meta objects to them. The proxy is a Key's instance created by *Guaraná*'s makeProxy() method and which has a meta object of class

MetaKey associated to it. Any attempt to access the proxy contents is intercepted by the meta level and redirected to the real key object kept secure in persistent storage.

5.3.2 Reflecting Over Transformations

By extending the basic *Guaraná*'s MetaObject class, we have implemented a class hierarchy responsible for transformations over arguments and results of intercepted operations. These classes are shown in the diagram of Figure 5.2. Class MetaCryptoEngine works as a specialized message handler useful for communication among meta objects or between base level and meta level. MetaCryptoEngine recognizes three subtypes of *Guaraná*'s Message interface: MethodToReflectAbout is used to add a method, whose arguments will be secured, to the meta object's list of methods; TurnOn and TurnOff are used to turn the security of the channel on and off, respectively. The broadcasted messages TurnOn and TurnOff are associated to the abstract methods turnon() and turnoff() that are supposed to be overloaded by subclasses implementing specific transformations.

As shown in Figure 5.2, MetaCryptoEngine has four direct subclasses, divided in two pairs. MetaTransformationParams and MetaReverseTransformationParams are responsible for performing transformations and their reverses over parameters. MetaTransformationResult and MetaReverseTransformationResult act over returned results of operations. The first pair overloads the MetaObject's handle for operation in order to perform the transformation and their reverses over intercepted methods' arguments. The second pair overloads the handle for result in order to transform the returned results of intercepted operations. All these meta transformations have abstract methods (transformParam(), revertParam(), transformResult(), and revertResult()) which are supposed to be implemented by subclasses for specific transformations.

Specific Cryptographic Transformations

There are four subclasses for each meta transformation of Figure 5.2. Each of them corresponds to one of the four categories of cryptographic services. For example, class MetaTransformationParams has the subclasses MetaEncryptionParams, MetaMdcGeneratorParams, MetaMacGeneratorParams, and MetaSignatureParams. The corresponding reverse transformation class, MetaReverseTransformationParams, has the following four subclasses: MetaDecryptionParams, MetaMdcVerificationParams, MetaMacVerificationParams, and MetaSignatureVerificationParams. The complete hierarchy, shown in Figure 5.3, contains 16 concrete classes for the basic cryptographic services. These classes do not cope with cryptographic service composition, and, although these meta objects can be used separately, the power of MOLC lies in the composition of them. We have developed special meta objects for this purpose.

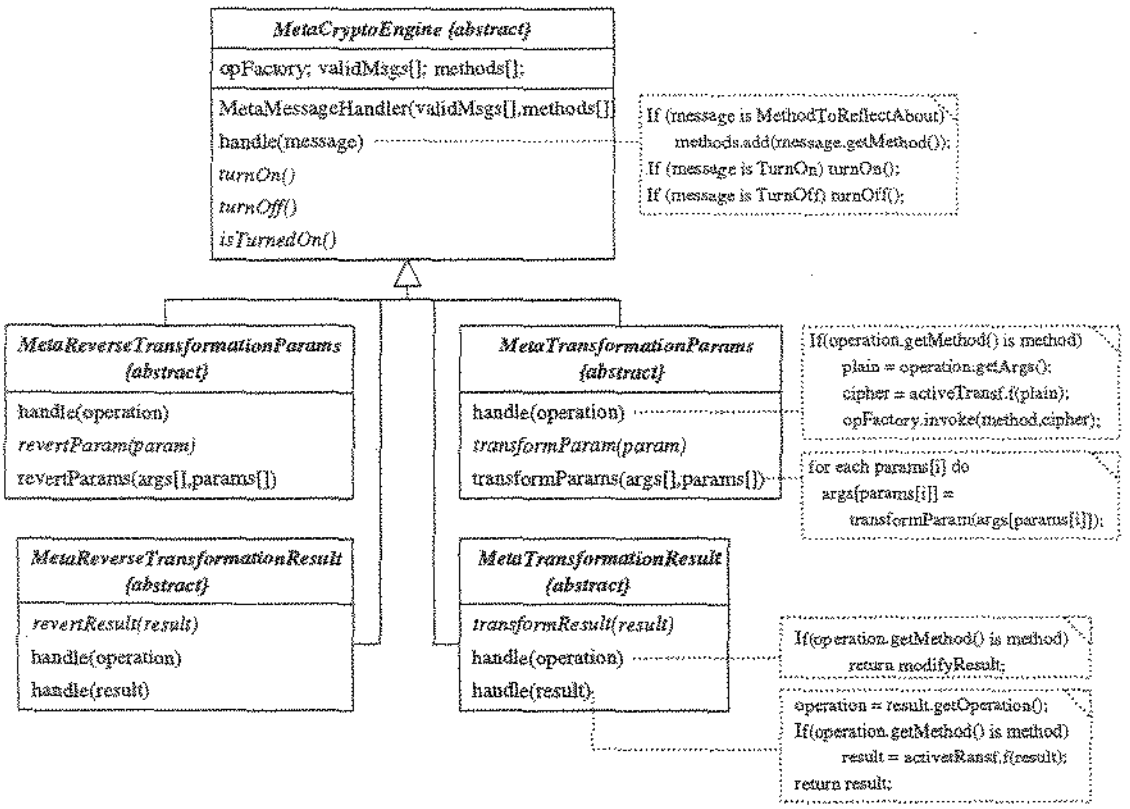

Figura 5.2: Abstract Meta-Objects for Generic Transformations.

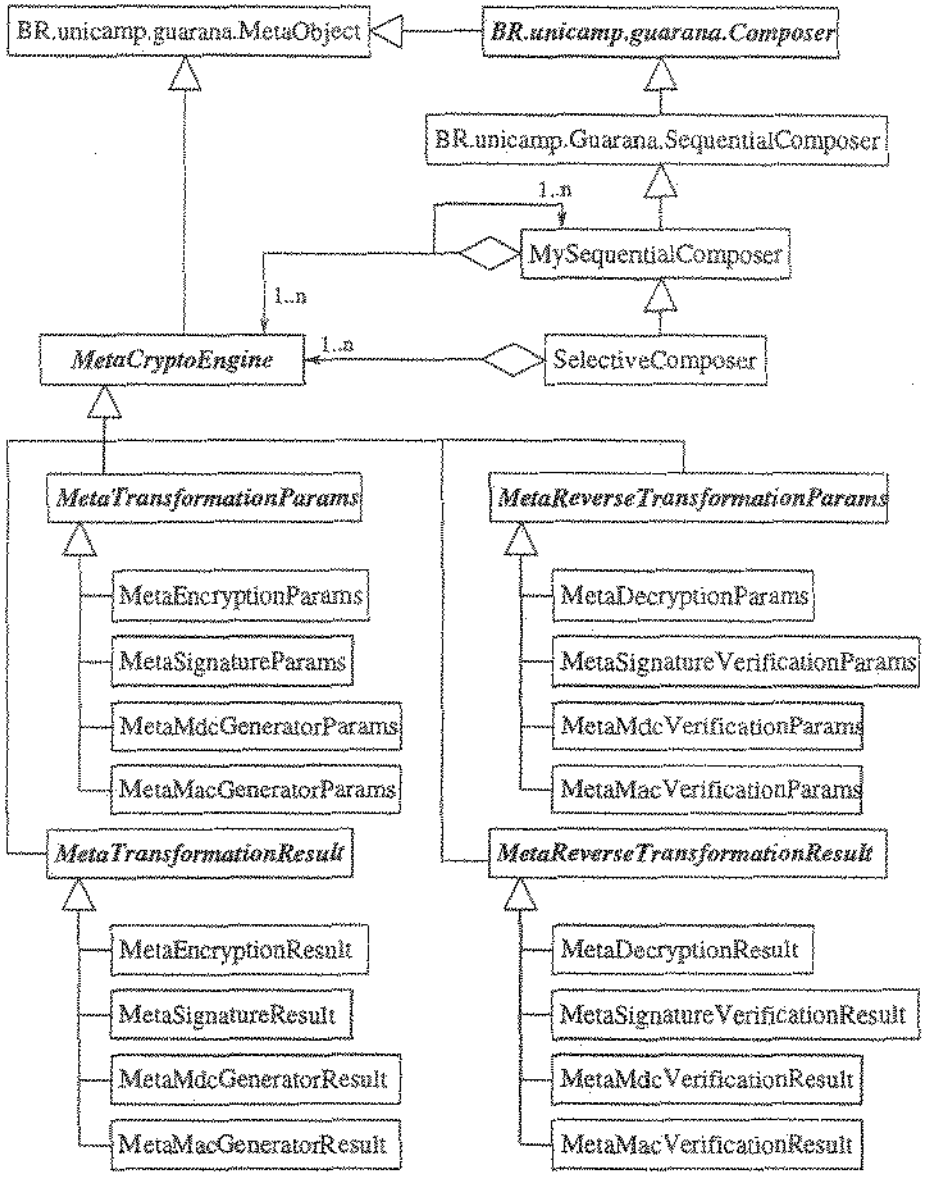

Figura 5.3: Meta-Objects for Cryptographic Services and Their Compositions.

5.3.3 Composing Cryptographic Services

The cryptographic services for MDCs, MACs and digital signatures are mutually exclusive and relate to each other as follows: MDCs support data integrity only, MACs support sender authentication and data integrity, digital signatures support non-repudiation and both sender authentication and data integrity. Encryption is orthogonal to the other cryptographic services and can be combined to each of them. Our pattern language [BRD98a] documents the constraints over cryptographic services combination by limiting the number of valid patterns. The ways meta objects for cryptographic transformations are composed are limited by the number of cryptographic patterns.

The reflective architecture of *Guaraná* provides an easy way for meta-object composition [OB98a]. An abstract subclass of MetaObject called Composer can be used to define arbitrary policies for delegation of control to other meta objects, separating the functionality of the meta level from its organization and management aspects. Particularly, *Guaraná*'s SequentialComposer delegates control to its aggregated meta objects sequentially and recovers the results of them in reverse order.

We subclassed the Composer meta object and obtained a ConfigurableComposer, which has the ability of turning its meta objects on and off according to a list of valid Message's subclasses, which are received and used as filters or function selectors. Another useful subclass of Composer we have implemented is the SelectiveComposer, which implements mutually exclusive selection of meta objects. That is, at any time, there is at most one meta object active, the others are kept off. The selection of active meta objects, similarly to the ConfigurableComposer, is based on subclasses of *Guaraná*'s Message which work as function selectors. The messages that can be understood by ConfigurableComposers, SelectiveComposers and MetaCryptoEngines are shown in Figure 5.4. The static relations among these three meta objects are shown in Figure 5.3. ConfigurableComposers can contain instances of SelectiveComposers, MetaCryptoEngine and other ConfigurableComposers. SelectiveComposers can contain only instances of MetaCryptoEngine.

SelectiveComposers can be used to implement the mutually exclusive aspect for the composition of meta objects responsible for MDCs, MACs, and signatures. In such a situation, the messages used to select functions are the subclasses of FingerprintOn and FingerprintOff. ConfigurableComposers can be used to compose orthogonal meta objects, preserving the ability of arbitrarily turning them on and off. A common configuration is to use a ConfigurableComposer to combine the behaviors of an encryption meta object and a SelectiveComposer, already initialized with meta objects for signature, MACs, and MDCs. In this case, the messages used for function selection are the subclasses of EncryptionOn and EncryptionOff and the messages addressed to the instance of SelectiveComposer. Another useful configuration is the use of a SelectiveComposer's instance to select among alternative encryption algorithms. Any meta configuration, using cryp-

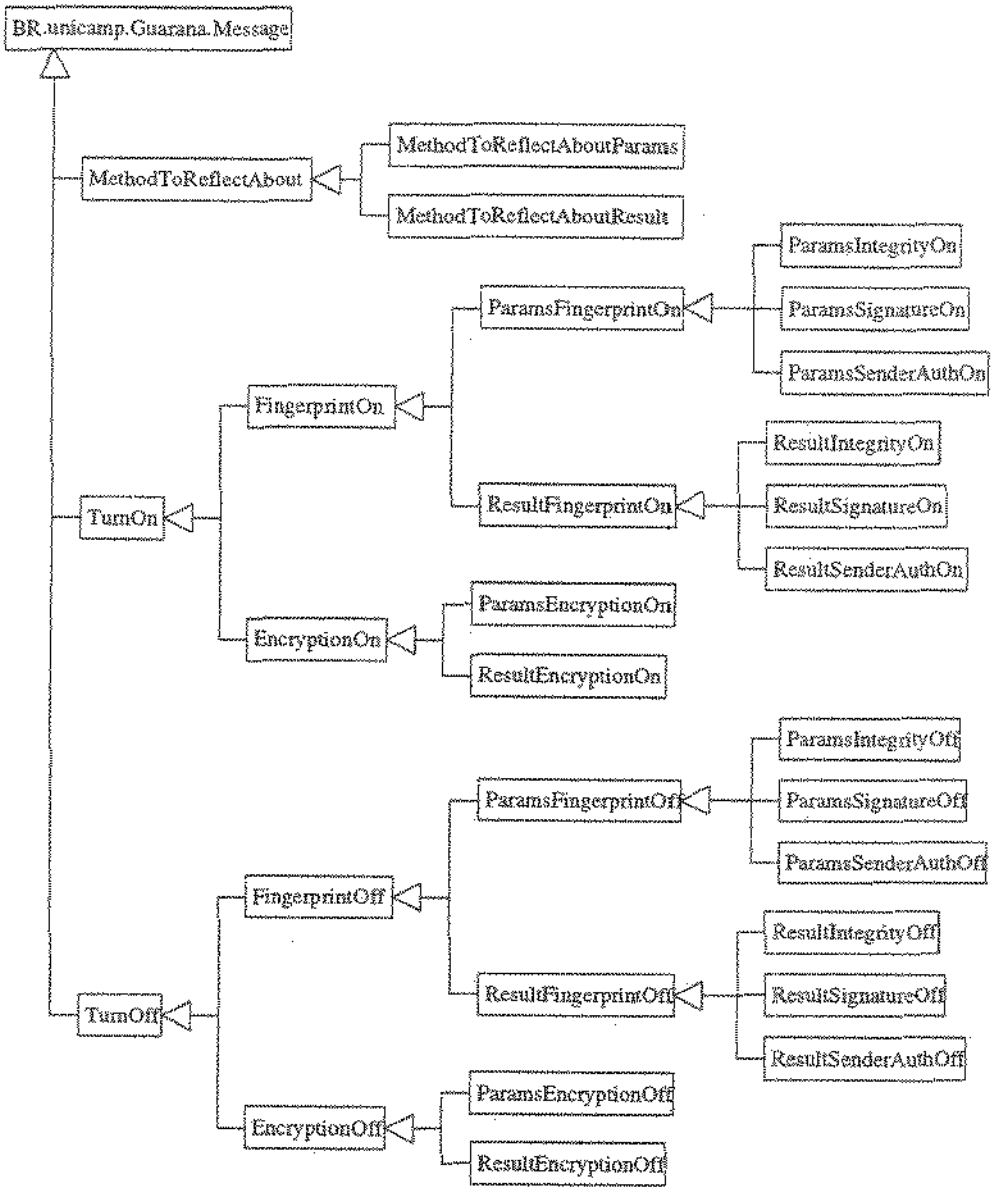

Figura 5.4: Message Hierarchy for Cryptographic Service Selection.

tographic meta objects either individually or in compositions, is an instantiation of the reflective cryptographic pattern [BRD99a]. An interesting property of our implementation is that cryptography-aware meta objects can be composed in any order.

5.3.4 The Underlying Cryptographic Service Library

Since version 1.1.2 of the Java Development Kit, Java has offered an object library for low-level cryptographic services such as digital signatures and hash functions [JBK98, Oak98, MDOY98]. Encryption facilities, due to export restrictions, are not available from the basic library issued by Sun. An extension to Sun's basic library is available only in United Stated and Canada, though free implementations can also be found. Such a library, known as the Java Cryptographic Architecture (JCA), is so flexible that its API can be used as a hook to either JCA's third-party implementations or to other proprietary implementations. In both cases, services are accessible through the JCA's API. In order to overcome the export restrictions, we have developed our own cryptographic library based on Java 1.1's JCA. This approach has also brought greater control over implementation details, which are usually not available from third-party code. The description of our household JCA implementation will be available as a technical report.

The Java cryptographic API, though quite complete, offers only low-level control over cryptographic routines and secured data [BDR99b]. Similarly to old cryptographic APIs, byte arrays are the data structure used to input and output. There are few facilities to encipher and sign objects [JBK98, GS98]. Another potential disadvantage is the amount of knowledge that a client object should have about the API. Such a client object should look after cryptographic objects' initialization with keys and block splitting for input and output. A cryptography-aware meta object which takes care of such an old-style and perhaps unfriendly API is also complex enough to make its reuse very difficult.

In order to simplify the design and implementation of the cryptography-aware meta objects, we have developed a set of adapters, in the sense of the *Adapter* [GHJV94, 139] design pattern, which deals with the low-level cryptographic API and provides meta objects with easier abstractions to deal with. Some of these adapters and the static relationship between them and meta objects are shown in Figure 5.5. Each cryptographic meta object contains a reference to an adapter. Such a reference can be easily switched between a real adapter and a null one. This approach implements the null state variation of the *NullObject* pattern [MRBV97, 5] making the modification of the meta-level behavior easy without having to change meta objects. For example, the meta object responsible for signing methods' parameters, MetaSignatureParams, contains a reference to a SignerInterface which can be either a Signer or a NullSigner. The corresponding meta object for verification of signatures, MetaSignatureVerification, contains references to an

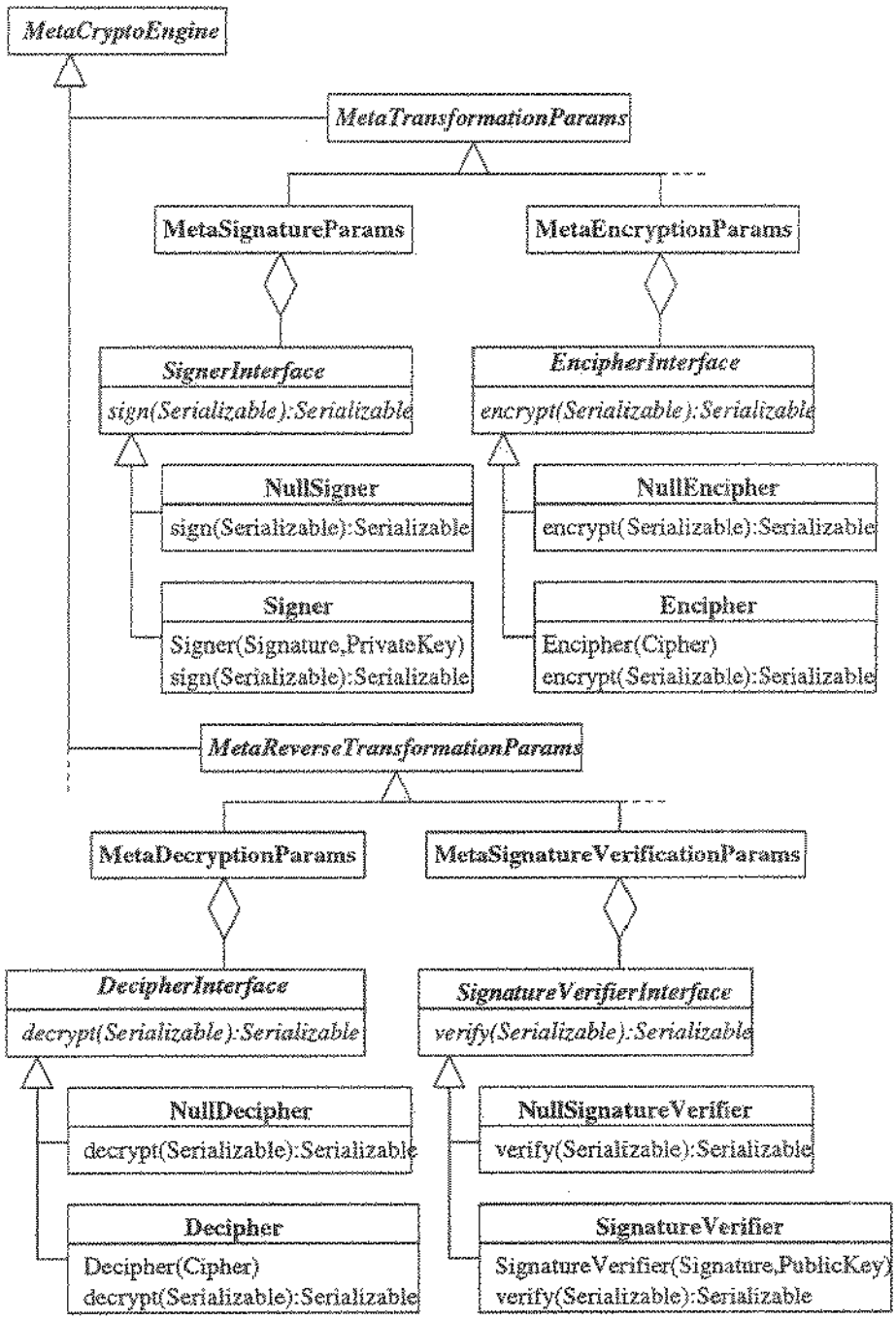

Figura 5.5: Relationships among Meta-Objects and Adapters.

adapter which can be an instance of either SignatureVerifier or a NullSignatureVerifier. The other cryptography-aware meta objects work in the same way.

Adapters use serializable objects in both input and output. Such a feature eliminates the disadvantage mentioned above, which was the use of a lower-level abstraction for input and output. We have extended JCA's set of classes that handle objects, that is SignedObject and SealedObject, in order to cover also authentication with MACs and integrity checking with MDCs. This set of secure objects are shown in the class hierarchy of Figure 5.6. This implementation of the Serialization [MRBV97] pattern is also a *Composite* [GHJV94, 163] in the sense that the composition of cryptographic features such as signing and encryption is facilitated. Implementation details, such as object serialization and block splitting, are no longer a problem and are treated by such objects in an implementation dependent way, which is hidden from the user of MOLC.

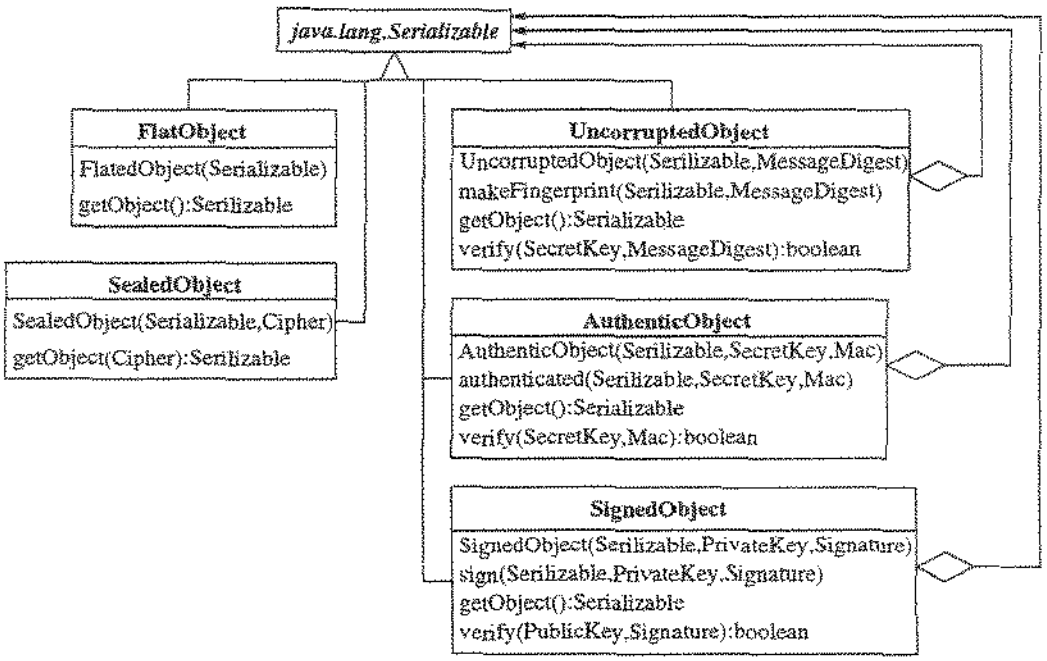

Figura 5.6: Secure Objects Hiearachy.

The use of adapters and serializable secure objects simplifies the design of cryptography-aware meta objects in such a way that meta objects do not have to worry about JCA's API specifics. Meta objects are free from such low-level responsibilities and are concerned only with whether a transformation should be performed or not. The composition of cryptographic services can be easily accomplished using SequentialComposer's delegation facilities. The sets of adapters and secure objects and the cryptographic service library constitute the lower layer of the MOLC framework.

Adapters sign errors during fingerprint verification or decryption using exceptions from

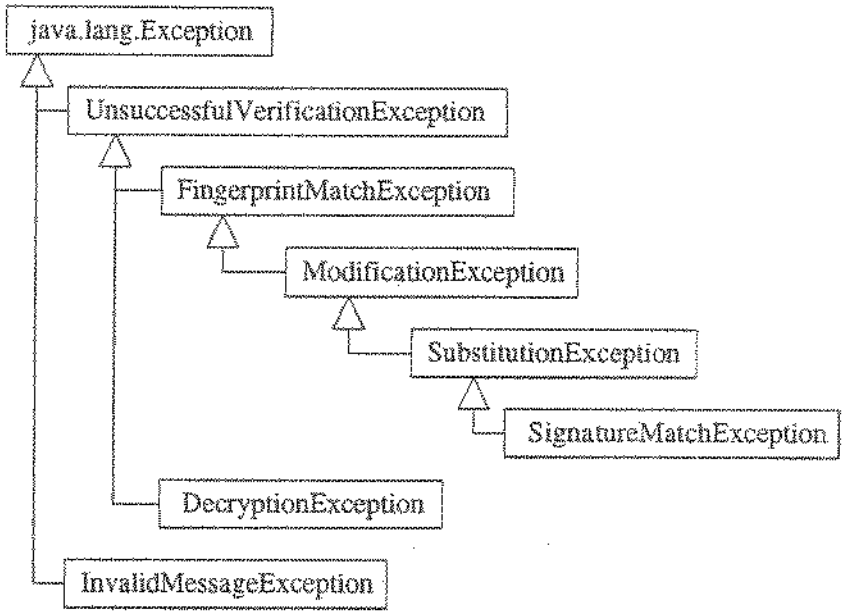

Figura 5.7: Cryptographic Service Verification Exceptions.

the class hierarchy of Figure 5.7. This hierarchy captures the containment relation among different kinds of fingerprints. For instance, a SignatureMatchException can be caused by either substitution or modification during verification of a digital signature. Errors of MAC verification can cause a SubstitutionException to be thrown. Similarly, MDCs errors throw ModificationExceptions and decryption ones throw DecryptionExceptions. ConfigurableComposers and MetaCryptoEngines throw an InvalidMessageException upon receiving an unknown Message's instance. All the exceptions of Figure 5.7 can be encapsulated in a *Guaraná*'s MetaException.

5.4 Reflective Framework for Cryptography

The MOLC's procedure for adding cryptography-based security has the following steps:

1. Load base-level classes. That is, load the classes for which a meta configuration is required.

2. Reflect about base-level classes, which means to create the meta configuration required by the base-level application.

3. Start up the meta objects from a secure initial state.

4. Load the class of the base-level application.

5. Execute the base-level application from the meta level.

Steps 1, 3 and 4, are the same for any application, having a few, parameterizable, differences. Steps 2 and 5 are what can vary among applications. In step 2, a meta configuration is created based on the base-level application's security requirements and, although a limited number of cryptographic services is available, the requirements can vary a lot and produce strongly different meta configurations. Step 5 has at least two main variations: execute a static method, probably a main() one, or create an instance of the application class and then execute some ordinary method.

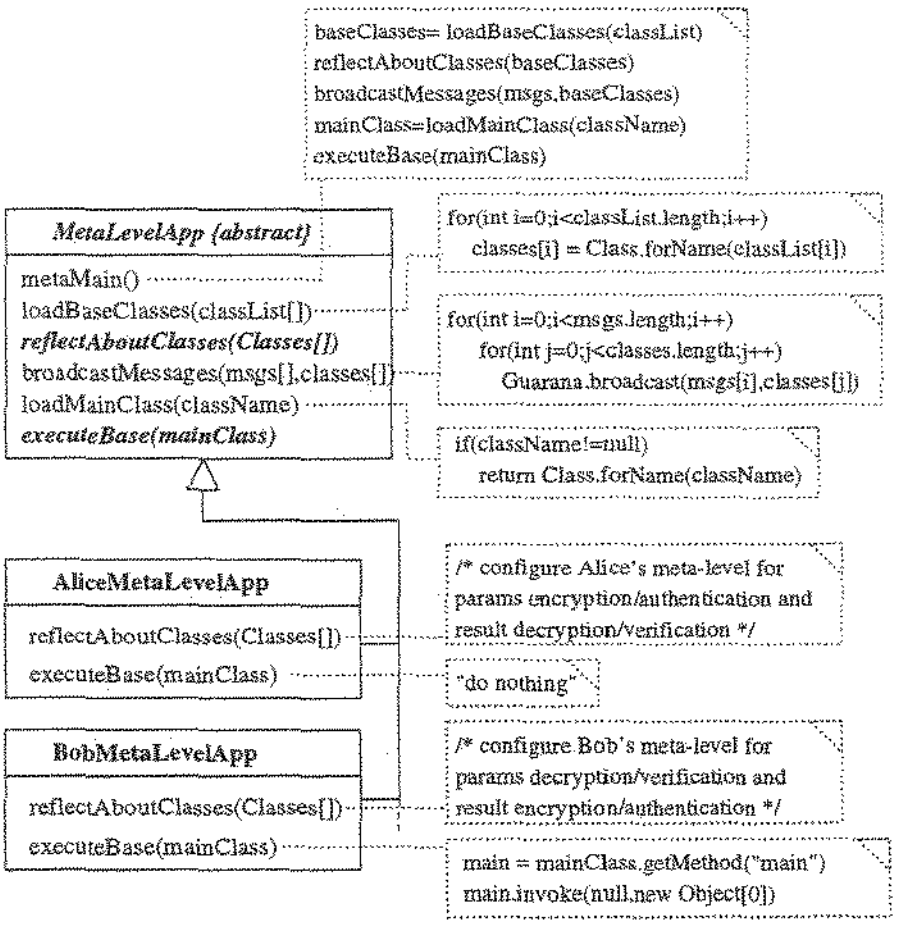

Figura 5.8: A Reflective Object-Oriented Framework for cryptography-based Security.

This small algorithm can constitute a simple object-oriented framework [Pre95] for adding cryptography-based security to any object-oriented application and can be implemented as the *Template Method* [GHJV94, 325] design pattern. Figure 5.8 shows the design of the framework. The abstract class MetaLevelApp implements the algorithm's

invariant parts, leaving hooks for subclasses. Method metaMain() performs the algorithm. The methods loadBaseClasses(), loadMainClass() and BroadcastMessages() are the invariant parts. The abstract methods reflectAboutClasses() and executeMain() are the hooks.

In order to test the framework, two subclasses of MetaLevelApp were implemented, AliceMetaLevelApp and BobMetaLevelApp. These classes implement the meta configuration shown in Figure 5.9, which also presents the runtime relationships among objects for such an application. It is important to notice that, as the number of MetaLevelApp subclasses increases, the use of MOLC becomes more and more like a black box. The application level contains Alice and Bob instances and a base-level application's instance, shown in Figure 5.9. The framework classes work as a glue layer between the base-level application and the meta-object library. In such a glue level there is a MainProgram which contains instances of the framework's classes AliceMetaLevelApp and BobMetaLevelApp. These classes are responsible for reflecting about Alice and Bob classes and instances. They also start up the base-level application. AliceMetaLevelApp and BobMetaLevelApp create two symmetric meta configurations. Symmetric meta configurations means complementary functions in each end of the communication channel. That is, when Alice's data should be encrypted, Bob's data should be decrypted and so on. Composer meta objects distinguish among mutual exclusive services and services compositions.

The meta configurations are associated to classes, that is Alice and Bob, and when new instances of such classes are created, the meta configurations are propagated. We decided to copy the class meta configuration to each new instance, instead of sharing a single one among several instances, in order to simplify the management tasks, particularly, the ones concerning key management.

5.5 MOLC's Reconfiguration Policy

Guaraná's meta-object protocol allows dynamic meta-level reconfiguration through replacement of meta objects during program execution. Although this feature makes the design of *Guaraná* extremely flexible, it is potentially harmful for security-aware meta objects. A secure policy for meta-level reconfiguration should be taken in order to avoid naive replacements of cryptography-aware meta objects.

When a cryptographic meta object is asked for reconfiguration, it can follow either a conservative or a non-conservative approach. In the conservative one, weakening the cryptographic features of an object's meta configuration is not allowed. In this approach, the meta configuration can either remain the same or allow self-strengthening. In the non-conservative approach, the weakening of meta configurations is also allowed. We adopted a conservative approach for meta-level reconfiguration. In our approach, a meta object for

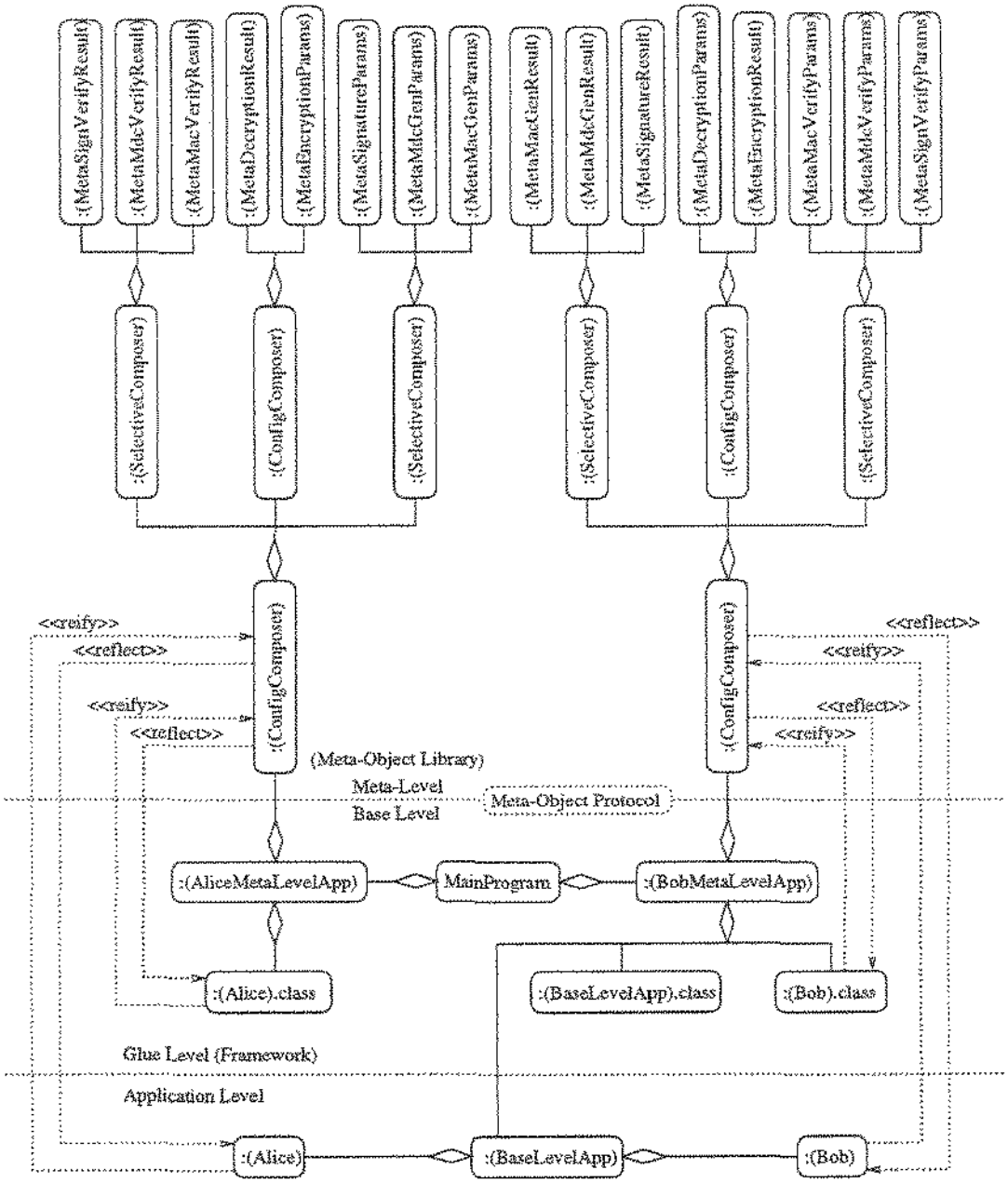

Figura 5.9: Runtime configuration for the example application.

signature cannot be replaced by neither a MAC meta object nor a MDC one and a MAC meta object cannot be replaced by a MDC one. Meta objects of the same type cannot replace each other either. A single encryption meta object can be composed, through a ConfigurableComposer, with any meta object for Signature, MAC or MDC. Figure 5.10 summarizes the contexts in which the following rules for reconfiguration are applicable. This minimum set can be easily extended to support more interesting policies. Starting from the most conservative, the rules we have implemented are:

Current \ New	Encryption	MAC	MDC	Signature	SelectiveComposer	ConfigComposer
Encryption	1	3	3	3	1	1
MAC	3	1	1	2	1	1
MDC	3	2	1	2	1	1
Signature	3	1	1	1	1	1
SelectiveComposer	1	1	1	1	1	1
ConfigComposer	1	1	1	1	1	1

Figura 5.10: Summary of the Reconfiguration Policy Applicability.

1. The current meta configuration is not replaced.
2. A selective composition of both current and new meta configurations replaces the current one.
3. A configurable composition of both current and new meta configurations replaces the current one.
4. The new configuration replaces the current one.

It is important to notice that such reconfiguration policy is only applicable when base level and meta level are co-designed and base level has a small control over which transformation should be active. On the other hand, when cryptography-based security is added to third-party components, such components do not have access to the meta level. Thus, there is no possibility of changing the meta configuration. Furthermore, the meta configuration of a key instance, that is, a MetaKey instance, cannot be modified in any case.

5.6 MOLC Programming Overview

The goal of this Section is to provide programmers with some feeling on integrating MOLC to third-party Java applications. We approach the instantiation of cryptographic meta objects, the composition of them and the integration of programs and meta programs. The following sample code creates part of the meta configuration shown in Figure 5.9 and outlines the implementation of the hooks of Figure 5.8.

`MetaLevelApp`'s subclasses implement the abstract method `reflectAboutClasses()`, which is responsible for creating meta configurations. Sample Code 5.6.1 is a fragmented implementation of `AliceMetaLevelApp`'s `reflectAboutClasses()` method. All cryptographic meta objects are instantiated similarly to the `MetaEncryptionParans`'s instance. It receives a set of Message subclasses, to which it is supposed answer, an initialized adapter and a list of Alice's methods, on which the cryptographic operations work. It is important to state that cryptographic operations work on an Alice's methods subset whose result or arguments are serializable.

A `ConfigurableComposer`, (Sample Code 5.6.1) `a_cc`, contains instances of both `MetaEncryptionParams` and `MetaDecryptionResult`. A `SelectiveComposer`'s instance, `a_sc1`, has an array of `MetaEncryptionEngine`'s subclasses (`MetaMacGenerationParams`, `MetaMdcGenerationParams`, and `MetaSignatureParams`). `a_sc2`, another instance of `MetaEncryptionEngine`, contains meta objects for Alice's verification of Bob's signatures, MACs, or MDCs, on his methods. Another `ConfigurableComposer`, `a_c` , aggregates all the other Composers and acts as Alice's primary meta object. *Guaraná*'s reconfigure method performs the task of setting an object's primary meta object.

`MetaLevelApp`'s `execute()` (Sample Code 5.6.3) method executes the `main()` method of `BaseLevelApp`. Both `AliceMetaLevelApp` and `BobMetaLevelApp` have to implement the `execute()` method. However, Alice and Bob belongs to the same program and there is no need for executing it twice. Thus, `AliceMetaLevelApp`'s `execute()` is null, while `BobMetaLevelApp`'s, below, performs the real work.

Meta proxies for cryptographic keys can be created in the following way (Sample Code 5.6.4). A `SecretKey` object, as well as a pass phrase used to encrypt the key and a file name, is passed to a `MetaKey` constructor, during `MetaKey` creation. Both the `Metakey` object and the `SecretKey` class are used by *Guaraná*'s `makeProxy()` method to create a `SecretKey` proxy, which can be attributed to a `SecretKey` variable. A `MetaKey` created with only a pass phrase and a file name is used to recover an already securely stored key from a file.

There should be a main program to launch the base-level application and settle its meta configuration. The main method of such a main program, Sample Code 5.6.5, is responsible for initializing the meta configuration from a secure state, creating adapters and

Sample Code 5.6.1 Creating Cryptographic Meta Objects

```
void reflectAboutClasses(Class[] classes){

                                        :

    MetaCryptoEngine a_mep =
        new MetaEncryptionParams(
                                    new Class[]{MethodToReflectAboutParams.class,
                                                ParamsEncryptionOn.class,
                                                ParamsEncryptionOff.class},
                                    encipher,aliceMethods);

                                        :

    Composer a_cc = new ConfigurableComposer(new MetaObject[]{a_mep,a_mdr});

                                        :

    Composer a_sc1 =
        new SelectiveComposer(
                                new MetaCryptoEngine[]{a_ap,a_hp, a_sp},
                                new Class[]{MethodToReflectAboutParams.class,
                                            ParamsFingerprintOn.class,
                                            ParamsFingerprintOff.class});

                                        :

    Composer a_sc2 =
        new SelectiveComposer(
                                new MetaCryptoEngine[]{a_vap,a_vhr,a_vsr},
                                new Class[]{MethodToReflectAboutResult.class,
                                            ResultFingerprintOn.class,
                                            ResultFingerprintOff.class});

                                        :

    Composer a_c = new ConfigurableComposer(new MetaObject[]{a_cc,a_sc1,a_sc2});
    Guarana.reconfigure(classes[i],null,a_c);
}
```

Sample Code 5.6.2 Composing Cryptographic Meta Objects

```
                                        ⋮
    Composer a_cc = new ConfigurableComposer(new MetaObject[]{a_mep,a_mdr});

                                        ⋮

    Composer a_sc1 =
        new SelectiveComposer(
                                new MetaCryptoEngine[]{a_ap,a_hp, a_sp},
                                new Class[]{MethodToReflectAboutParams.class,
                                        ParamsFingerprintOn.class,
                                        ParamsFingerprintOff.class});

                                        ⋮

    Composer a_sc2 =
        new SelectiveComposer(
                                new MetaCryptoEngine[]{a_vap,a_vhr,a_vsr},
                                new Class[]{MethodToReflectAboutResult.class,
                                        ResultFingerprintOn.class,
                                        ResultFingerprintOff.class});

                                        ⋮

    Composer a_c = new ConfigurableComposer(new MetaObject[]{a_cc,a_sc1,a_sc2});
    Guarana.reconfigure(classes[i],null,a_c);
}
```

Sample Code 5.6.3 Executing The Base-Level Application

```
void execute(Class c)
{ (c.getMethod("main", new Class[]{String[].class}))
        .invoke(null, new Object[]{new String[0]});}
```

Sample Code 5.6.4 Protecting Keys in Meta Objects

```
        ⋮
    SecretKey k0 = new SecretKey(),
        k1 = (SecretKey) Guarana.makeProxy(SecretKey.class,
                                           new MetaKey(k0,"passphrase","Key.ser")),
        k2 = (SecretKey) Guarana.makeProxy(SecretKey.class,
                                           new MetaKey("passphrase","Key.ser"));
        ⋮
```

Sample Code 5.6.5 The Glue Program

```
public static void main(String[] argv) {
    ⋮
    Message intialState[] = new Message[]{new ParamsIntegrityOn(),
                                          new ParamsEncryptionOn(),
                                          new ResultSenderAuthOn(),
                                          new ResultEncryptionOn()};
    ⋮
    Object[] adapters = new Object[]{ new MdcGenerator(messagedigest),
                                      new MdcVerifier(messagedigest),
                                      new MacGenerator(mac,k1),
                                      new MacVerifier(mac,k1),
                                      new Encipher(cipher1),
                                      new Decipher(cipher2)};
    ⋮
    AliceMetaLevelApp aliceApp =
        new AliceMetaLevelApp(new String[]{"Alice"},null,initialState);
    BobMetaLevelApp bobApp =
        new BobMetaLevelApp(new String[]{"Bob"},"BaseLevelApp",initialState);
    ⋮
    aliceApp.cryptoInit(adapters);
    bobApp.cryptoInit(adapters);
    aliceApp.metaMain();
    bobApp.metaMain();
  }
}
```

launching meta Alice and meta Bob, whose `execute()` method launches `BaseLevelApp`.

A list of Message's instances represents the meta configuration's initial state, which is check integrity of, and perform encryption on the arguments of Alice's methods, and perform sender authentication and encryption on results of Bob's methods. A list of adapters supplies the cryptographic transformations for such an initial state. Lacking features, such as non-repudiation, are internally filled by null adapters in order to be kept turned off. Alice's and Bob's `MetaLevelApps` receive the initial state, the name of the base-level application (in which case Alice's `MetaLevelApp` receives null), and finally the names of the classes to reflect about, that is, Alice and Bob. Before calling the `metaMain()` methods of meta Alice and meta Bob, both of them receive the list of adapters.

5.7 Conclusions and Future Work

In this work, the main aspects of a meta-object library for cryptography were presented. This meta-object library is a reflective extension of a well known cryptographic object library, Sun's Java Cryptography Architecture. In addition to being an object-oriented framework for transparent addition of cryptography-based security to third-party components, this reflective extension is able to easily compose cryptographic services, a lacking feature of many cryptographic libraries [BDR99b], according to a set of cryptographic patterns. An interesting feature of MOLC is the ability to use the reflective cryptographic pattern recursively, for example, when securing keys with meta proxies. Future improvements to this meta-object library can focus on such a self-securing ability. Particularly, efforts can be directed to self-authentication of classes in order to prevent unauthorized substitution of implementations and illegal reading or corruption of internal data.

Capítulo 6

The Role of Patterns in an Object-Oriented Framework for Cryptography

6.1 Introduction

Cryptography software design is an important topic these days. There is an urgent necessity for cryptography-based security in applications ranging from electronic commerce to word processing. It is not economically feasible still developing cryptographic software as it used to be in World War II, from scratch. Cryptography is a complicate subject, it is not a good idea to suppose everyone is able to (or has time and money to) learn cryptographic techniques. We are living in the object-oriented component-based era. Thus, what programmers, who have to deal with cryptography-based security requirements, really want is an easy-to-use highly reusable cryptographic component as well as a direct way to find out the right security feature. In order to help programmers in finding the right cryptographic services for their requirements, we have proposed a pattern language for cryptographic object-oriented software [BRD98a]. In order to provide programmers with easy-to-use cryptographic mechanisms, we have developed a reflective object-oriented framework as well as a meta-object library for cryptography [BDR99a].

It has been extensively argued that patterns generate architectures [BJ94], as well as pattern languages document frameworks [Joh92, BMA97]. In this work we address the role of patterns in using as well as designing an object-oriented framework for cryptography. Our case study is a reflective cryptographic pattern [BRD99a], which is a variation of the ones present in the pattern language for cryptographic software [BRD98a], applied to document the usage of a reflective object-oriented framework for cryptography [BDR99a]. In order to illustrate how patterns can document design, we present the overall design of

the cryptographic framework as a set of integrated patterns instantiations.

This paper is organized as follows: Section 6.2 is an overview of cryptographic design patterns. Section 6.3 is a discussion about the intrinsic relation among pattern languages and frameworks; Section 6.4 introduces the framework's design issues and examples of usage. Section 6.5 presents some performance measurements on using the framework. Conclusion and future work are in Section 6.6. This text uses a UML-like notation for diagrams, otherwise a legend is available when necessary. The Java programming language is used for code samples.

6.2 Cryptographic Patterns Overview

Modern cryptography addresses many security services [iso98]. Four of them are considered main security goals [MvOV96]: confidentiality, integrity, authentication, and non-repudiation. Accordingly, there are four basic cryptographic mechanisms: (*i*) encryption/ decryption, (*ii*) MDC (Modification Detection Code) generation/verification, (*iii*) MAC (Message Authentication Code) generation/verification, and (*iv*) digital signing/verification. These four mechanisms can be combined in specific and limited ways to produce more high-level ones and are the building blocks for security services as well as security protocols.

Confidentiality is the ability to keep information secret except from authorized users. Data integrity is used to guarantee that 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. The basic cryptographic services can be invoked in appropriate combinations with other services and mechanisms. Particular cryptographic mechanisms can be used to implement the basic services. Practical realizations of systems may implement particular combinations of the basic cryptographic services for direct invocation.

The present interest in software architectures [SG96, BMR+96] and patterns [GHJV94, CS95, MRBV97, BRD99c], and the existence of well-known cryptographic solutions to recurring security problems [iso89, iso98, MvOV96, Sch96] motivate the development of cryptographic software architectures and cryptographic patterns. Our pattern language 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. Securing a communication channel can be such an important task that it should be accomplished by the application itself, without compromising its main functionality. *Secure-Channel Communication*, the foundation pattern, documents general aspects of

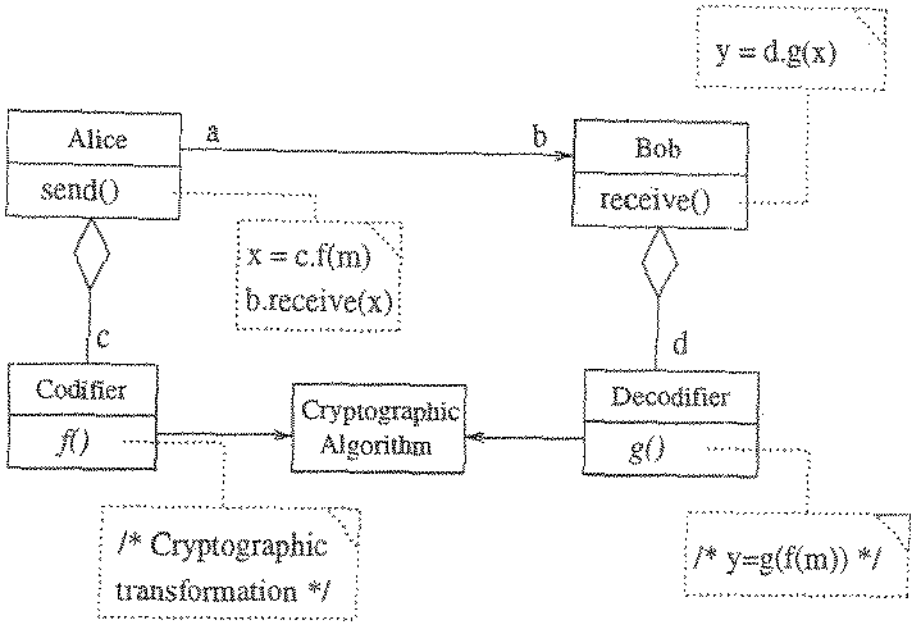

Figura 6.1: Secure-Channel Communication Structure.

both structure and behavior common to secure communication, independent from the kind of cryptographic transformation performed. Figure 6.1 shows this generic structure defining two template classes, Alice and Bob, which are application classes, and two hook classes, Codifier and Decodifier, which are cryptography-aware classes. The class Codifier has a hook method $f()$, which performs a cryptographic transformations. The class Decodifier defines a hook method $g()$, which performs the reverse transformation, $y = g(f(x))$. The transformation and its reverse are based on the same cryptographic algorithm.

#	Pattern Name	Purpose
1	Secure-Channel Communication	A General Software Architecture for cryptographic software
2	Information Secrecy	provide secrecy of information
3	Message Integrity	detect corruption of a message
4	Sender Authentication	authenticate the origin of a message
5	Signature	provide the authorship of a message
6 2	Secrecy with Integrity	detect corruption of a secret
7	Secrecy with Sender Authentication	authenticate the origin of a secret
8	Secrecy with Signature	prove the authorship of a secret
9	Signature with Appendix	separate message from signature
10	Secrecy with Signature with Appendix	separate secret from signature

Table 6.2: The Cryptographic Design Patterns and Their Purposes.

Table 6.2 summarizes the patterns corresponding to the basic cryptographic mechanisms and their valid compositions. They are applicable as follows. When either on-line communication or exchange of information through files takes place, sometimes, due to great sensitiveness of data, (*Information*) *Secrecy* should be guaranteed. However, secrecy alone does not prevent either modification or replacement of data. Particularly in on-line communication, granting *Message Integrity* and (*Sender*) *Authentication* is also important. Sometimes, 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. The cryptographic services, in appropriate combinations, lead to *Secrecy with Integrity*, *Secrecy with Sender Authentication* or *Secrecy with Signature*. Cryptography can be so time consuming that algorithm performance is always important. *Signature* can be speeded up by a *Signature with Appendix*. Similarly, *Secrecy with Signature* can be speeded up by *Secrecy with Signature with Appendix*.

Figure 6.2 is a directed acyclic graph of dependences among patterns. An edge from pattern A to pattern B means pattern B derives from pattern A. *Secure-Channel Communication* generates the micro-architecture for the four basic patterns. All other patterns are combinations of these. Thus, all the lower-level cryptographic patterns instantiate *Secure-Channel Communication*. A walk on the graph is directed by two questions. First, how should the cryptographic software be structured to obtain both easy reuse and flexibility? Second, what cryptographic services should be added to the current instantiation of *Secure-Channel Communication* in order to address application requirements and user needs?

The cryptographic mechanisms corresponding to the services for data integrity, sender

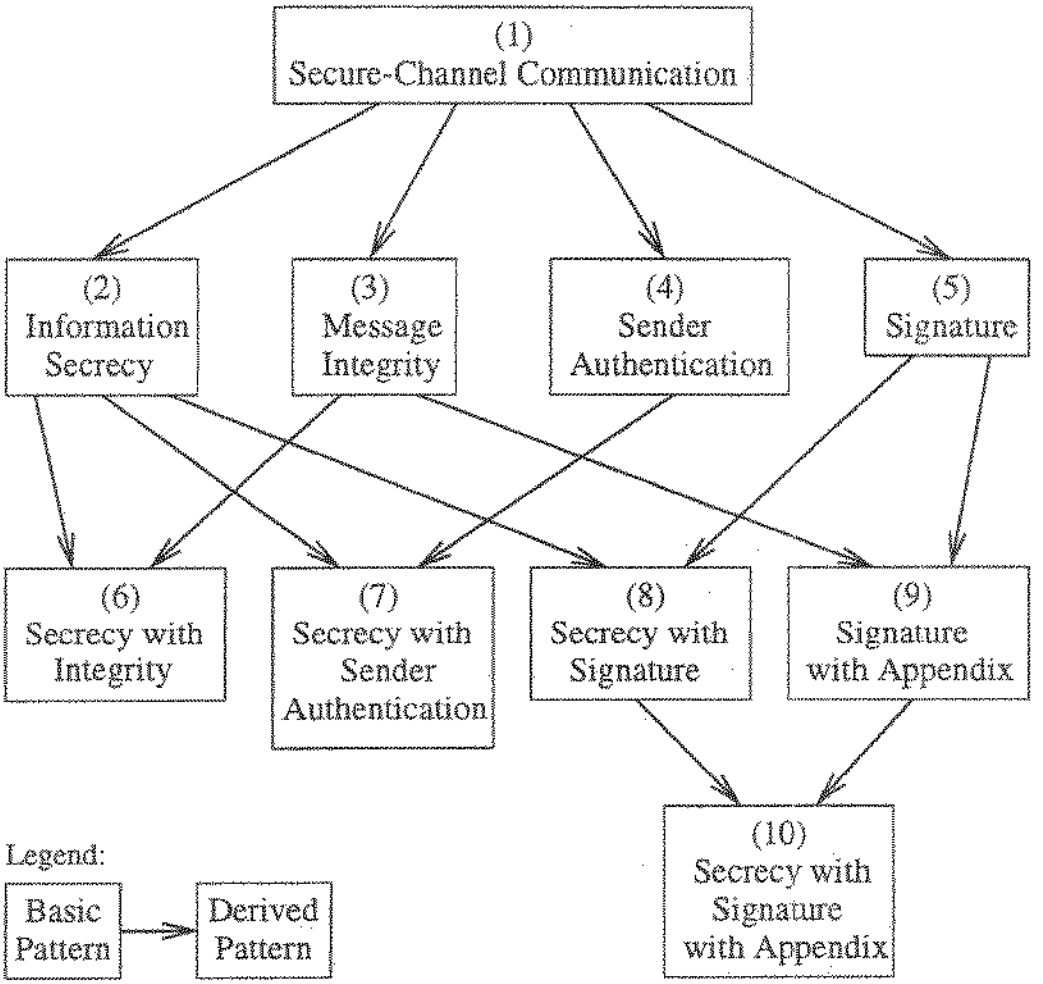

Figura 6.2: Cryptographic Design Patterns and Their Relationships.

authentication and (digital) signatures relate to each other as follows: MACs support data integrity, signatures support both sender authentication and data integrity as well as non-repudiation. Encryption, which supports confidentiality, is orthogonal to the other cryptographic mechanisms and can be combined with each of them.

Our pattern language documents both the use and appropriate combination of cryptographic mechanisms in order to accomplish not only the basic cryptographic services, but also the high-level composed ones, in secure communication. In fact, the combined patterns can be viewed as high-level services able to increase the cryptographic unawareness of cryptographic libraries, which should offer not only the basic four mechanisms, but also their compositions. From a programmer point of view, such libraries can support the composed cryptographic patterns in a variety of ways, ranging from explicit programmer-made composition of basic mechanisms to transparent composition hidden in high-level, not necessarily programmer-friendly, interfaces.

Object-oriented applications with non-functional cryptography-based security requirements can benefit from a flexible design in which cryptographic objects and application functional objects are weakly coupled. We argue that the combination of computational reflection and cryptographic design patterns can improve reuse of both design and code (while decreasing coupling and increasing flexibility) of cryptographic components, this combination can also be treated as a design pattern. Following, the *Reflective Secure-Channel Communication* Pattern is presented in a simple format. The complete pattern description can be found in [BRD99a]. The use of computational reflection and object-oriented programming is not new [Mae87], neither is the use of meta-object protocols in the implementation of non-functional requirements of object-oriented applications [SW96]. Meta-object protocols have also been used to encapsulate authentication facilities and compose them with fault tolerance and distribution [FP96].

6.2.1 Reflective Secure-Channel Communication

Context *Secure-Channel Communication* forces functional objects (Alice and Bob) to explicitly take care of non-functional objects. That is, Alice and Bob reference cryptographic objects and decide when a cryptographic transformation should take place. This highly coupled design has the following disadvantages: (*i*) it limits the reuse of Alice and Bob; (*ii*) it pollutes application objects with explicit references and method invocations of non-functional cryptography-aware objects, reducing readability; (*iii*) it requires some background on cryptography from application programmers.

Problem How could the separation of concerns between application functional objects and cryptography-aware objects be explicitly represented in a way that reuse and rea-

dability can be improved? In other words, can cryptography-based security be added (transparently) to third-party applications or components, even if source code is not available?

Forces

- Cryptographic services are usually non-functional requirements of general purpose applications related to communication and persistence requirements, but are orthogonal to these. Leaving application responsibilities decoupled from security services facilitates reuse and security policy changes, and frees application programmers from having to acquire (too much) cryptographic knowledge.
- The explicit separation of concerns can lead designers to: (*i*) procrastination of important security policy decisions in cryptography-aware designs applications; (*ii*) lack of control over cryptographic features (for instance, key management), from the application programmers' point of view.
- Delegation of cryptography-aware decisions has the advantage of encouraging the utilization of largely tested (cryptanalyzed) components. However, it can also expose application functions and sensitive data to third party's Trojan horses.

Solution In order to overcome the limitation stated in the Section "Context", a restructuring of the interaction mechanism among objects can be used. Meta-object protocols (MOPs) with message interception mechanisms can potentially invert the dependencies among non-functional objects and functional ones, in such a way that non-functional requirements are transparently accomplished by non-functional objects, which may not be known by the application functional objects.

The use of a meta-object protocol explicitly separates cryptographic requirements from application functionalities. Figure 6.3 is the reflective version of *Secure-Channel Communication*. Classes MetaAlice and MetaBob are responsible for cryptographic method calls and for the re-sending of base-level methods, which were previously intercepted. For instance, the send() method is intercepted by the MOP's reflective kernel and materialized in a send-operation object. This operation object and its argument (m) are treated by the meta object ma, which requests the cryptographic transformation accordingly. The intercepted method is, then, re-sent (containing now the encrypted argument x) by the MOP's kernel to its original target. The interception of method receive() presents an analogous behavior.

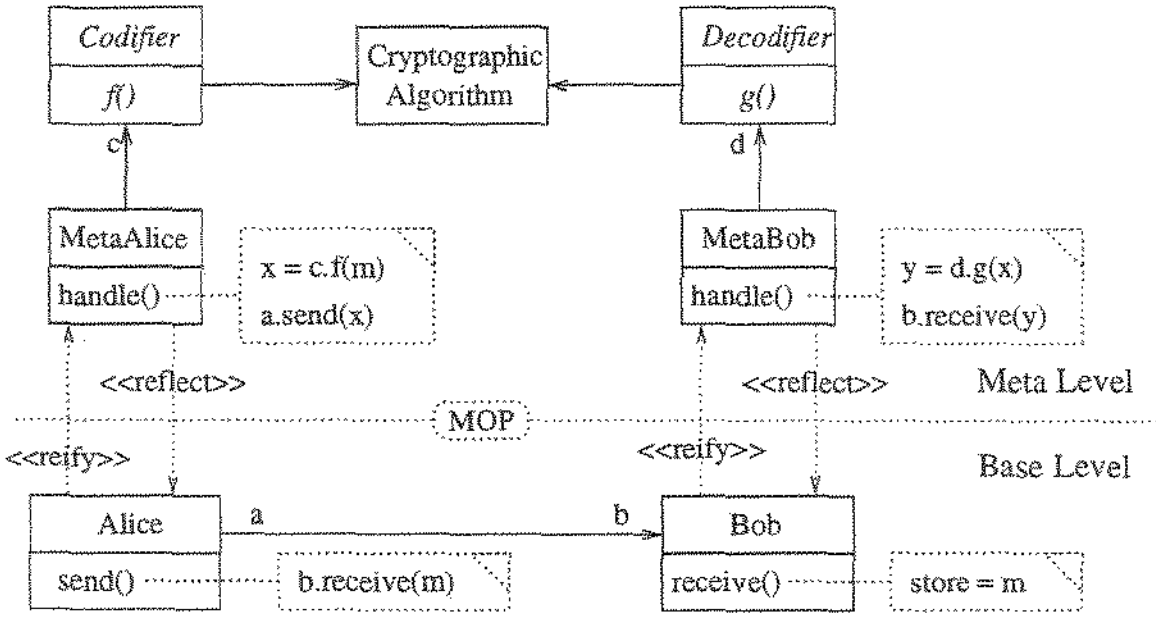

Figura 6.3: Reflective Secure-Channel Communication Structure.

6.3 Pattern Languages Generate Frameworks

Pattern languages generate frameworks when a framework offers the building blocks (public interfaces, hook methods and abstract classes) which can be used to solve problems targeted by a pattern language. Object-oriented frameworks are not off-the-oven black-box components, instead they have to evolve in time. This evolution is called the framework life span [BMA97]. The pattern language which accompanies the framework also evolves in time. Such an evolutional relation is stronger than parallel evolution. (Evolution in response to the same environmental stimulus, but not necessarily influencing each other.) It can be called co-evolution because modifications in one partner causes modifications in others. A framework's evolution ranges from white-box raw frameworks to black-box ones.

In the beginning of a framework's life span, the firstly implemented parts are architectural elements, which reflect the most general scenarios in the pattern language. As new applications (based on the framework) appear, the framework itself evolves by incorporating more specialized black-box components provided by these applications. The solutions a framework offers can be in one of three levels of abstractions [BMA97]: (*i*) elementary components responsible for architectural aspects and more general scenarios; (*ii*) basic design specializations, application independent components, conceived for specific domains; (*iii*) domain specific components, specializations of the ones in the previous category, obtained from framework's applications and added as black boxes. In matured

frameworks, most components are domain-specific black boxes, which reflect more specialized scenarios of framework's usage and generate variations for the patterns in the evolving pattern language.

We have developed a reflective object-oriented framework based on a meta-object library for cryptography [BDR99a]. This framework offers the cryptographic services stated in the cryptographic pattern language [BRD99c] using a reflective variation of those patterns [BRD99a]. We classify this framework, according to scope [FS97], as a system infrastructure framework because it helps the development of software systems' cryptography-based security infrastructures. Also, it can be called a young white-box framework since its mainly functionality is available through class inheritance and hook methods overloading (*Template Method* [GHJV94] pattern). However, it has a great potential to becoming a black-box framework since it is based on a closed, relatively small, set of patterns of usage for cryptographic services. Furthermore, the framework for cryptography already has specializations for reflective cryptographic patterns. These domain specific patterns are applicable when cryptography-based security is a non-functional requirement of applications and should be delegated to a meta-level in order to achieve separation of concerns between application's functionality and non-functional requirements.

6.4 Documenting the Framework

In this Section we show, by an example and structured documentation, how to both use and design the cryptographic framework. There are two kinds of documentation for object-oriented frameworks [MCK97]: (*i*) user documentation, intended to the final user and consisting of scenarios, tutorials and examples, and (*ii*) design documentation, information concerning the design of the framework; in both cases, patterns play an import role. These two types of documentation should target three kinds of users [MCK97]: (*i*) users deciding which framework to use, (*ii*) users wanting to build a typical application and (*iii*) users wanting to extend the framework. User documentation targets both users choosing a framework and users building typical applications with it. The third user type, users extending the framework, is mainly targeted by design documentation. This section tries to target both documentation types.

6.4.1 Using the Framework

We argue that the cryptographic pattern language can be used to choose a cryptographic service. Particularly, a walk in the graph of Figure 6.2 can lead to specific pattern instantiations. Once this pattern is known, tutorials and examples should be used to show how

Class 6.4.1 SaveObject.java — Saves Serialized Objects

```
public class SaveObject {
    public static void main(String[] args) {
        if ((args.length>2)&&(args.length<1)) {
            System.err.println("java SaveObject [-r] <classname>");
            System.exit(-1);}
        try {
            if (args[0].equals(-r")) System.out.println(readObject(args[1]+".ser"));
            else writeObject(Class.forName(args[0]).newInstance(), args[0]+".ser");
        } catch (Exception e) { e.printStackTrace();}
    }

    public static void writeObject(Object o, String n) throws IOException {
        ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream(n));
        out.writeObject(o);
        out.flush(); out.close();
    }

    public static Serializable readObject(String n)
        throws IOException,ClassNotFoundException{
        ObjectInputStream in = new ObjectInputStream(new FileInputStream(n));
        Object o = in.readObject();
        in.close();
        return((Serializable) o);
    }
}
```

to instantiate the pattern from framework's classes. A framework's usage documentation should not only provides scenarios of applicability for each pattern in the corresponding pattern language, but also show how to instantiate the application from the classes of the framework. A pattern format can be used for this task [OQC97]. Since the intent of this text is exemplify the framework's usage, we target both features for one pattern in free prose format.

We extend a simple command-line program for saving objects in order to encrypt and test the integrity of the persistent instance. The command to save objects has the following syntax: `SaveObject [-r] <ClassName>`, in which `-r` is an optional switch used to read an object from persistent storage. The command for coping, encrypting and testing integrity of objects, `SecureObject [-r] <ClassName>`, generates an encrypted copy of `<ClassName>` instances, still made by SaveObject, in persistent storage upon receiving a password. The `-r` switch loads, decrypts and tests the integrity of `<ClassName>`'s persistent instances. A scenario of usage for SecureObject can be the following, regarding that both Alice and Bob know the password: (*i*) Alice uses `SecureObject Valuable` to protect her sensitive data, that is, the instances of class Valuable; (*ii*) Bob uses `SecureObject -r Valuable` to recover the sensitive data.

The most important requirement of SecureObject, besides being secure for both encryption and data integrity, is do not change SaveObject code, since we want to keep the ability to simply save objects by typing `SaveObject Valuable`. When looking at the cryptographic pattern language for a pattern that fits this requirements, one can find that basic cryptographic services should be combined to achieve both confidentiality and data integrity. This lead to the *Secrecy with Integrity* pattern. However, a non-intrusive approach is also required to avoid code modification. Thus, a reflective cryptographic pattern should be used. The key idea is extend SaveObject without intrusion.

The next step is to determine what should be known about SaveObject. In other words, what should be the SaveObject's hook interface to SecureObject. This information can be obtained from either design documentation or code listings. Class 6.4.1 show that SaveObject uses FileInputStreams to read from files and FileOutputStreams to write to files. Fortunately, the code for saving and recovering is hidden in two methods, writeObject() and readObject(). A simple solution can intercept SaveObject's read/write operations and perform the necessary cryptographic transformations over those parameters or result.

Class 6.4.2 shows SecureObject, the main program responsible for launching the application and giving control to the framework. In order to use the framework, SecureObject should instantiate one of MetaLevelApp's specializations responsible for file encryption and integrity checking, called SecureFile. This class can either encrypt or decrypt objects, according to a crypt flag, upon receiving a password; integrity checking is performed du-

Class 6.4.2 SecureObject.java — Application's Main Program

```
public class SecureObject {
    public static void main(String[] argv){
        boolean crypt = !argv[0].equals("-r");
        String password = new String("Easy password!");
        String className = "SaveObject";
        Method toIntercept;
        MethodToReflectAbout message;
        Message m[];
        try{
            if (crypt){
                toIntercept = Class.forName(className).getMethod("writeObject",
                    new Class[]{Object.class,String.class});
                message = new MethodToReflectAboutParams(toIntercept);
                m = new Message[]{
                    new ParamsIntegrityOn(),
                    new ParamsEncryptionOn(),message};
            } else {
                toIntercept = Class.forName(className)
                    .getMethod("readObject",new Class[]{String.class});
                message = new MethodToReflectAboutResult(toIntercept);
                m = new Message[]
                    {new ResultIntegrityOn(),new ResultEncryptionOn(),message};
            }
            SecureFile sf = new SecureFile(crypt,password,
                    new String[]{className},"SaveObject",m);
            sf.setArgv(argv);
            sf.metaMain();
        } catch(Exception e){System.out.println(e);}
    }
}
```

ring decryption.

Cryptographic transformations are activated by events. Events can not only turn both encryption and integrity on and off, but also add methods to the list of secured methods. When the flag is on, the events ParamsEncryptionOn and ParamsIntegrityOn are used for SecureFile initialization; otherwise, events ResultDecryptionOn and ResultIntegrityOn take place. Method setArgv() transmits the application's argument list to SaveObject. Method metaMain() catches application's main loop.

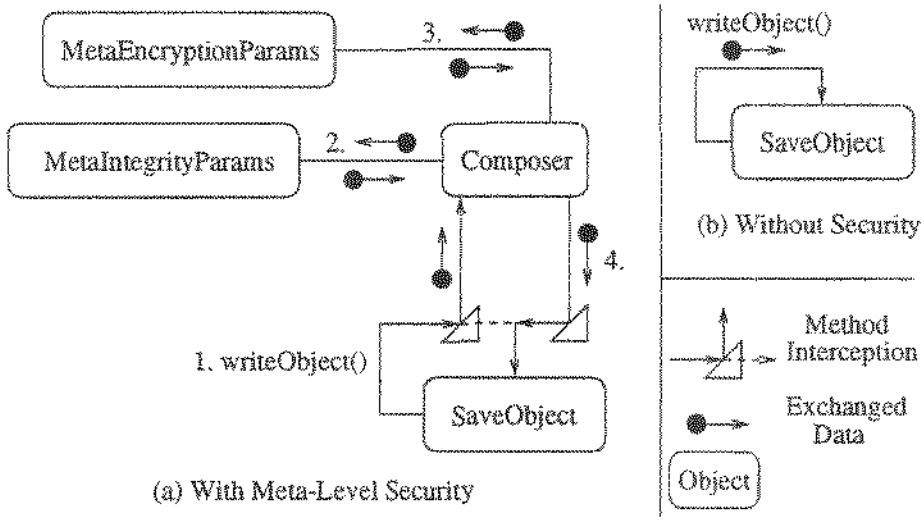

Figura 6.4: Sequencing of Actions During Method Interception.

Figure 6.4 show the runtime behavior for SecureObject. The important objects are the three meta-level objects and the base-level object. Classes SecureObject and SecureFile are not shown in the figure, since they are not important at runtime; the first is a glue class for framework launching; the second instantiated the framework behavior. Figure 6.4a shows the sequence of actions concerning method SaveFile.writeObject() interception, the data handling at meta level and the method re-sending to base level. Figure 6.4b shows the usual behavior of SaveObject.

The cryptographic patterns instantiation, as well as the framework use, imposes discipline in programming. Por example, its easier to find out the hook interface for SaveObject because the input/output operations are encapsulated in two methods, which can be mapped to Alice.send() and Bob.receive(). Also, users intending to develop common applications from the framework do not need to know that computational reflection is being used to invert the dependencies between objects, because this is a design information. Summarizing, the process of using the cryptographic framework comprises the following general steps, which should be supported by meta-object library catalogs, tutorials, examples, and pattern instantiation's scenarios:

1. Determine the adequate cryptographic pattern for instantiation.

2. Find out the base-level application's hook interface. Cryptographic transformation are usually performed over input/output operations for either communication or storage.

3. Define the event flow, if any, from base level to meta level. This step is usually necessary when the principle of separation of concerns should be neglected.

4. Look for an specialization of MetaLevelApp that addresses both the pattern from step 1 and the hook interface from step 2. In this step, the specialization's security specification, such as cryptographic algorithm implementation and key length, should be evaluated according to security requirements.

5. Implement an adequate MetaLevelApp subclass if none could be found in step 4. This specialization should be added to the framework. This step can be skipped more often as the framework matures.

6. Implement a glue program (in this text, SecureObject) for framework initialization and launching.

6.4.2 Designing the Framework

The documentation of a Framework's design can be structured as a four-level-depth tree [OQC97], in which each level addresses a deeper aspect of the software, as shown by Figure 6.5. Level 1 targets the system architecture. Doing so, it describes the purpose of the system as well as the properties of the approach. Level 2 documents the properties exposed by Level 1 as a set of interrelated pattern instantiations. This level can also contains pattern instantiation details; for example, modifications in standard structure in order to address particular aspects of the present instantiation. Level 3 documents the classes in detail, but relegating the documentation of source code to Level 4. Hypertext-based tools can be used to materialize this tree-structured documentation [OQC97, MCK97].

Level 1: System Architecture

A key design issue in this framework is to support ease of use and flexibility for reuse of cryptography services, that is, the possibility of using cryptographic features without (too much) knowledge about cryptography. The following three properties enforce this issue.

1. Implicit invocation of cryptographic services. Cryptographic services do not need to be explicitly invoked by function calls. Instead, a hierarchy of messages (or events)

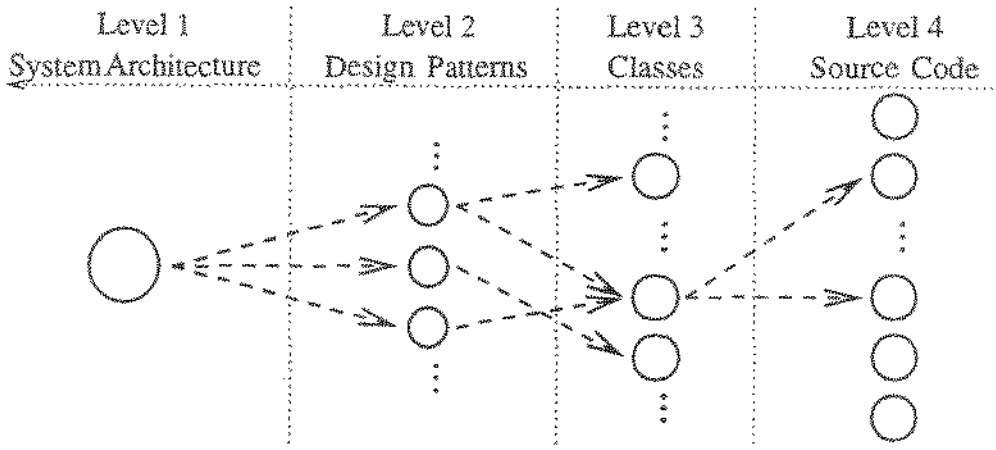

Figura 6.5: Levels for Framework's Design Documentation.

can be broadcasted by the application and target framework (meta) objects as in the *Implicit Invocation* [SG96] architectural style.

2. Transparent composition of services. Cryptographic services can be applied not only individually, but also in combinations to application's data in a way that application's objects cannot determine the order of cryptographic transformations' execution.

3. Potentially transparent addition of security features. Cryptography-based security can potentially be added to third-party software without code recompilation. Even when source code is not available. This kind of program extension is obtained by using meta objects as in the *Reflection* [BMR+96] architectural pattern. A strong constraint of this approach is that the framework should be informed about what methods and objects should be made secure. Events from the implicit invocation mechanism can be used to broadcast this information.

A number of cryptographic application programming interfaces and cryptographic libraries are available today [JDK+91, LMJW93, Kal95, Mic96, JBK98, Oak98, gcs96, css97]. However, none of them targets ease of use and flexibility for reuse, since they use procedure calls for explicit service invocation and, in general, do not provide service composition in an easy-to-use way [BDR99b].

Figure 6.6 shows the main components of this architecture. The framework contains the flow of program's execution, which in turn contains the base-level application. This feature characterizes it as an object-oriented framework [Pre95, Lew96]. The cryptographic routines are obtained from a Java Cryptographic Service Provider [JBK98, Oak98]. Because the cryptographic provider's routines are in a low level of abstraction, an adapter

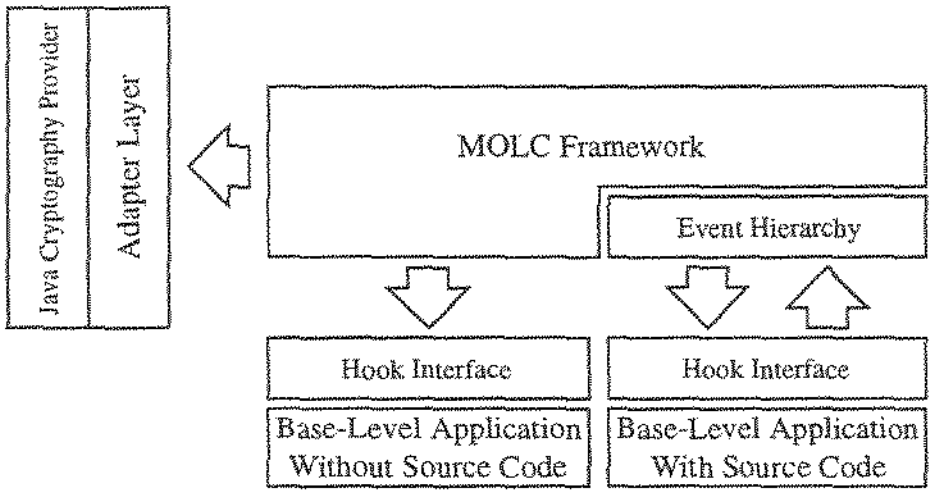

Figura 6.6: Framework High-Level Organization.

layer between meta-object library and the Java Cryptographic Provider is necessary in order to reduce both the complexity of meta objects and the dependencies from particular implementations of the Java cryptographic library. Besides offering meta objects for cryptographic transformations over base-level data, the framework can also reflect about itself in order to secure its own data, for example, cryptographic keys.

We distinguish between base-level applications with and without source code available. In the first, both meta level and base level are, probably, being developed in parallel and it can be desirable for the base level to control some aspects of security services. For example, turn either encryption or authentication of a communication channel on and off or even ask for security in a new aspect of its computation. In this situation, base-level applications should be supplied with mechanisms for communicate with meta level, but without polluting their code with explicit references to cryptographic services. The implicit invocation mechanism of the framework targets this issue directly.

It is important to notice that the idea behind base-level objects communicating, even implicitly, with meta-level objects is not conceptually correct in computational reflection, since such a feature breaks the separation of concerns principle. However, it represents an attempt of using meta-object protocols as architectural connectors for component communication, in which two-way communication between components is usually necessary.

In the second case, when either source code is not available or it is desirable to preserve the explicit separation of concerns, base-level applications cannot change any aspect of security features determined in the meta level, mainly due to two reasons. The first is the constraint that modifications should not be allowed. The second is an implementation constraint that cryptographic features are not known by the base-level code. In both cases, with and without access to source code available, the meta level should be notified

about which aspects of base level's computation should be made secure. These aspects constitute a hook, not necessarily public, base-level interface known by meta-level objects. We have implemented this hook interface as a set of methods in which either parameters or results can be serializable. That is, methods with the property of using serializable data in their signatures. Such methods are the Template methods Alice.send() and Bob.Receive() from the *Secure-channel Communication* pattern and its reflective variation.

Level 2: Design Patterns

In order to accomplish the three issues from Section 6.4.2, we adopted a design pattern-based approach in which patterns instantiations are grouped in clusters. A cluster corresponds to the blend of its patterns in such a way that all these patterns are applied together to solve a problem. In a cluster, it is difficult to separate a member pattern's implementation from the others'. This design contains six main clusters, shown in Figure 6.7 and listed below, which are put together by the reflection [BMR+96] pattern instantiated by *Guaraná* [OGB98], a meta-object protocol for Java based on method interception and meta-object composition. The pattern clasters are the following:

1. *Meta-Key Proxy.* Recursive use of the reflective cryptographic pattern to implement secure proxies for cryptographic keys as meta objects.

2. *Selective Broadcast (Implicit Invocation).* Two patterns instantiated in *Guaraná* (*Reactor* [CS95, Sch95] and *Composite* [GHJV94]) are combined to implement an implicit invocation mechanism.

3. *Secure Objects.* Secure objects are electronic envelops for (encrypted) data and their fingerprint. Each kind of secure object targets one cryptographic mechanism, say, encryption, data integrity, sender authentication and digital signing. This cluster uses *Composite* [GHJV94] and *Serialization* [MRBV97] to obtain easy cryptographic service composition.

4. *Security Service Turning.* This cluster uses the *NullObject* [MRBV97] pattern, a specialization of the *State* [GHJV94] pattern, to implement the ability of turning a cryptographic mechanism on and off. Null objects are instantiated over adapters, producing null adapters. Real adapters offer cryptographic services based on secure objects, instead of streams or arrays of bytes. Thus, adapters simplify the instantiation of the reflective cryptographic patterns.

5. *Framework Behavior.* The inversion of control is provided by a *Template Method* [GHJV94] instantiation, which provides some hooks for creating the abstract factory responsible for cryptographic objects instantiation.

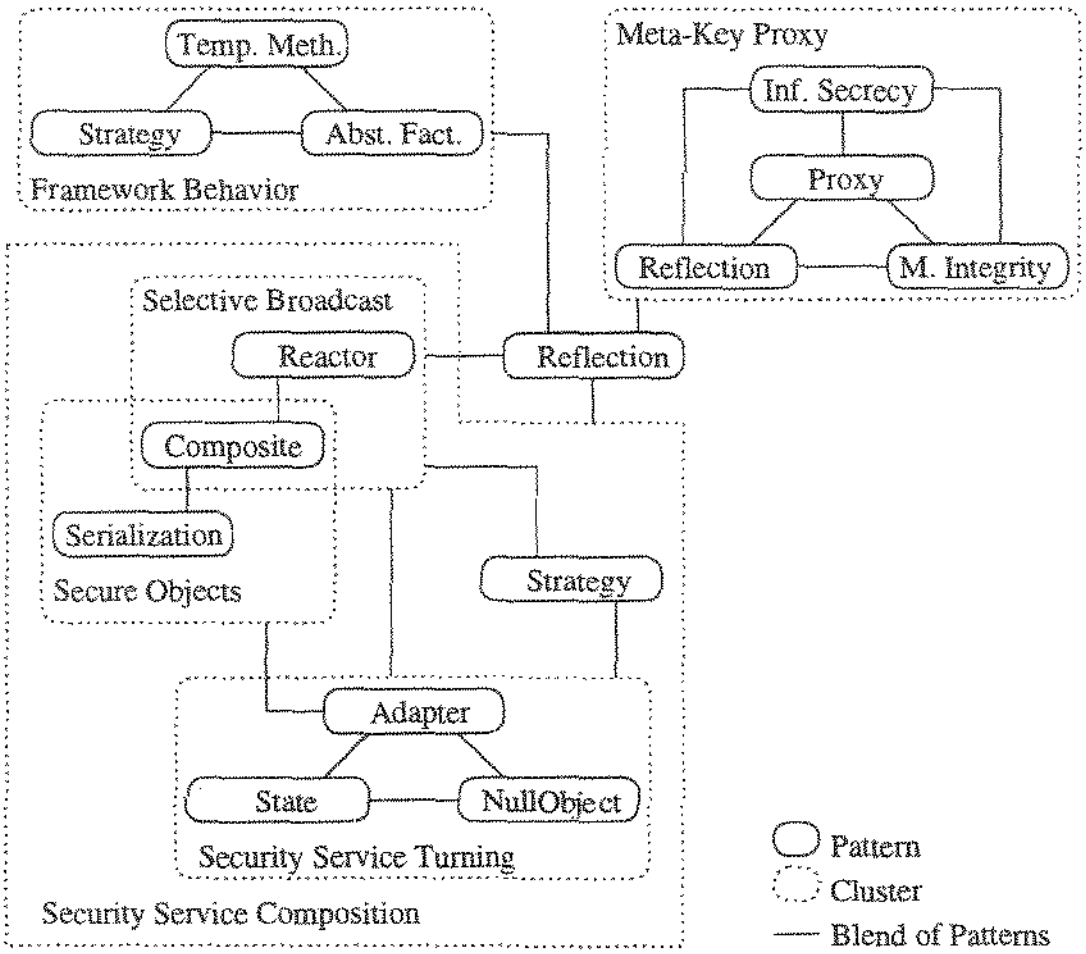

Figura 6.7: Patterns in the Design of the Cryptographic Framework.

6. *Security Service Composition.* In this cluster, meta objects for cryptographic mechanisms are specializations of a generic cryptographic engine. Members of its class hierarchy share the ability to turn themselves on and off upon receiving an event. There are two ways for composing cryptographic mechanisms, mutually exclusive composition, used for mechanisms which should not be used simultaneously (for example, data integrity should not be used with signatures, since the second may imply the first) and chaining composition, used to combine orthogonal mechanisms, for example, encryption and data integrity.

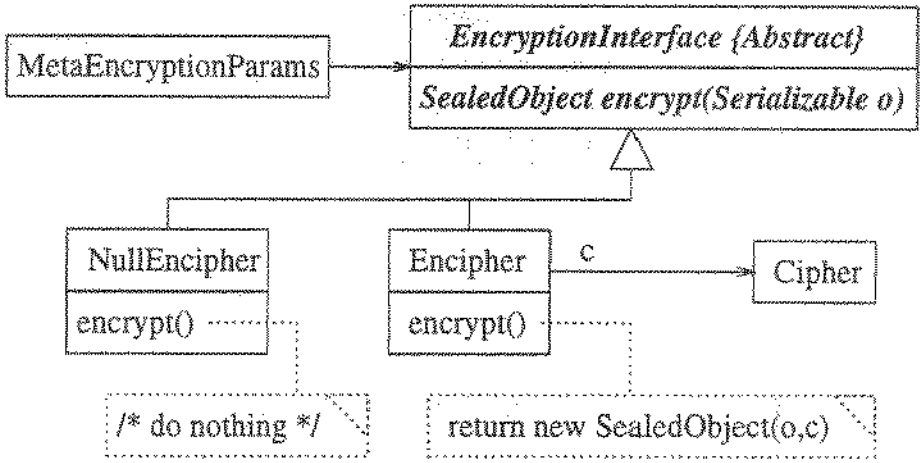

Figura 6.8: Instantiations of Adapter and NullObject Patterns.

In Figure 6.8, an *Adapter* [GHJV94] pattern instantiation, MetaEncryptionParams uses the simpler interface from EncryptionInterface, instead of the low-level one from Cipher, through an Encipher. A particular feature of this implementation, which distinguishes it from the standard *Adapter* pattern, is that Ciphers, the adaptees, are not called directly by Enciphers, the Adapters. Insteat, SealedObjects are responsible for the details of this calling. This instantiation simplifies not only meta object design, but also adapter design, since details of ciphers, such as padding, are also hidden from adapters. A NullEncipher instantiates the *NullObject* [MRBV97] pattern when MetaEncryptionParams should be turned off.

Levels 3 and 4: Classes and Source Code

Level 3 can detail not only individual classes, but also class relationships not shown by patterns, as deep inheritance hierarchies and package grouping. The class hierarchy for MetaEncryptionParams is in Figure 6.9. This class is a MetaEncryptionEngine specialization for parameters encryption, so that it is a MetaTransformationParams. It inherits

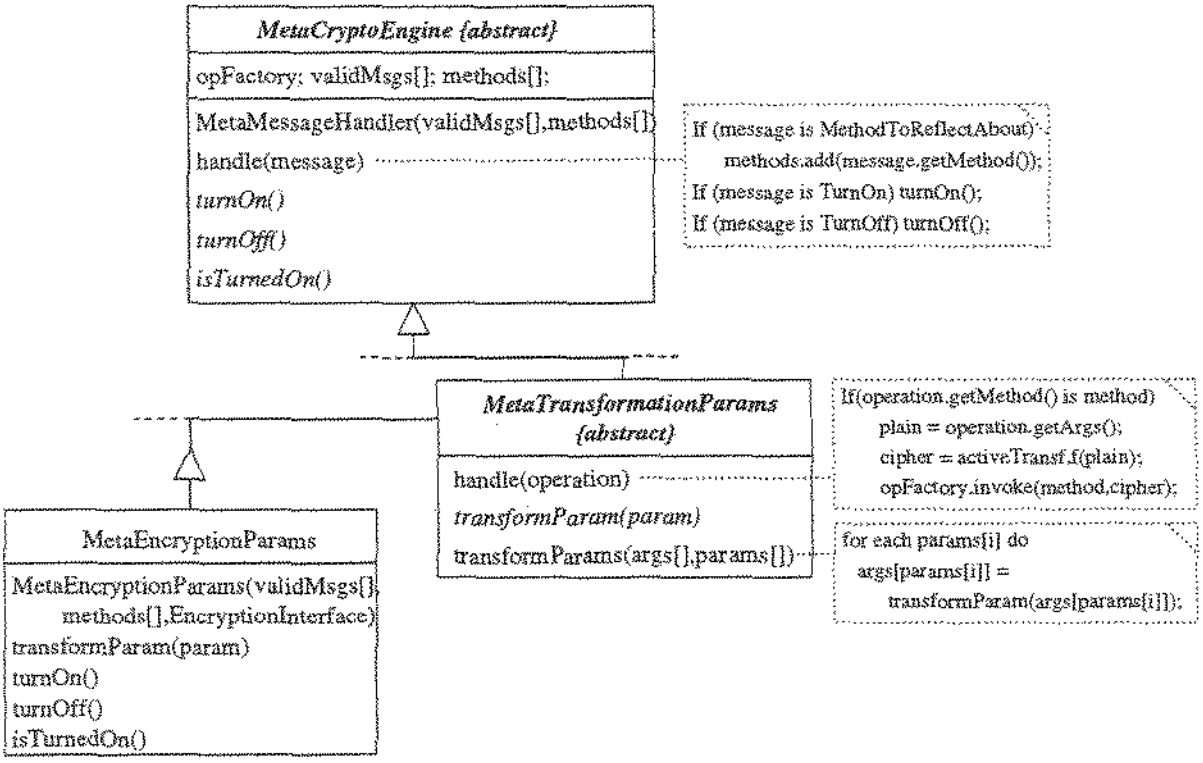

Figura 6.9: MetaEncryptionParams' super classes.

the ability for reacting to events from its grandparent class and the ability for intercepting base-levels operations and capturing its parameters from its parent class. It also implements the hook methods in order to obtain the ability for encryption of data and turning encryption on and off.

In order to complete this case study, we present the source code for two classes. Class 6.4.3 contains the source for MetaEncryptionParams and Class 6.4.4 details SealedObject. Two interesting points are the focus of that code samples. First, how adapters make MetaEncryptionParams's implementation really small and simple. Second, how the knowledge about the underlie cryptographic library is encapsulated by SealedObject and potentially hidden from upper levels. Its important to mention that sealed objects, *per si*, are not a new concept [GS98]. One of the contributions of this framework is to extend the encrypted-object concept to all other cryptographic mechanisms, producing a family of secure objects.

Class 6.4.3 MetaEncryptionParams.java

```
public class MetaEncryptionParams extends MetaTransformationParams{
    protected EncipherInterface transf, active_transf;

    public MetaEncryptionParams(Class[] valid_msgs, EncipherInterface t,Method[] methods)
        throws InvalidMessageException
    {super(valid_msgs,methods);this.transf = transf;}

    Serializable transformParam(Serializable p){ return(active_transf.encrypt(p));}

    final void turnOn(){active_transf = transf;}
    final void turnOff(){active_transf = new NullEncipher();}
    final boolean isTurnedOn() {return!(active_transf instanceof NullEncipher);}
}
```

Class 6.4.4 SealedObject.java

```
public class SealedObject implements Serializable {
    private byte[] encryptedObject;

    private byte[] addPadding(byte[] serialForm, Cipher c){ ... }

    private byte[] subPadding(byte[] serialForm){ ... }

    public SealedObject(Serializable object, Cipher c){
        try{ encryptedObject = c.doFinal(addPadding(Serializer.serialize(object),c));}
        catch (Exception e){ System.out.println(e);}
    }

    public final Object getObject(Cipher c){
        Object object = null;
        try{object = Serializer.unserialize(subPadding(c.doFinal(encryptedObject)));}
        catch (Exception e){ System.out.println(e);}
        return(object);
    }
}
```

6.5 Performance Evaluation

The goal of the performance measurements below is to evaluate the impact of the meta-object library for cryptography [BDR99a] over the implementation of common security services. The measurements were performed over methods of type Alice.send() and Bob.receive(). In the first, Alice.send(), method interception and cryptographic transformations are performed over arguments; in the second, Bob.receive(), over returned results.

We measured the cost for using the framework in three situations: (*i*) over method interception only (that is, using a null adapter); (*ii*) over cryptography transformations when performed by either meta objects or Alice and Bob themselves; (*iii*) and over storing secured objects when security is either in meta level or base level. Here, encryption means the encapsulation of a serialized encrypted form of the original object in a sealed object. This meaning is analogous for both integrity and authentication. Securing means the encapsulation of a serialized encrypted form of the original object in a sealed object, which is also encapsulated (along with a modification detection code) by an integrity object.

The measurements were performed on a 100 MHz MicroSparc workstation running SunOS 5.5.1 and a 110 MHz SparcStation-5 running SunOS 5.5.1. The Java virtual machine was Kaffe openVM 1.0.b2 tempered with Guaraná 1.5.1. The cryptographic function for encryption was the DES symmetric-key algorithm in ECB mode. The MD5 hash function was used for integrity checking purposes and was applied after encryption and before decryption. MD5 was also used for MAC generation. A 1000-iteration loop was used for measurement. Time, in milliseconds, was measured as the elapsed time between two calls of the System.currentTimeMillis() function. All graphs below show mean values.

Figure 6.10 compares the time, in milliseconds, required by a cryptographic meta object for both method interception and re-sending in two different workstations. In this case, we used a null adapter so that arguments and results were not modified by meta objects. Both argument and result have a size of 128 bytes after serialization. This figure also shows that method interception and re-sending can be very fast for cryptographic meta objects. Interception of Bob.receive() result is faster than interception of Alice.send() arguments, because the latter should parse the argument list for serializable elements.

Figures 6.11 and 6.12 put together times for performing some cryptographic transformation, with and without reflection (in the figures, w.R. stands for with Reflection) for two different machines: a MicroSparc 100 MHz (Figure 6.11) and SparcStation-5 110 MHz (Figure 6.12). Measurements were made for integrity checking, authentication, encryption and encryption with integrity[1]. The bars in these figures are analogous and roughly differs

[1] A current incompatibility between kaffe 1.0.b2 and Sun's library for large integers made impossible

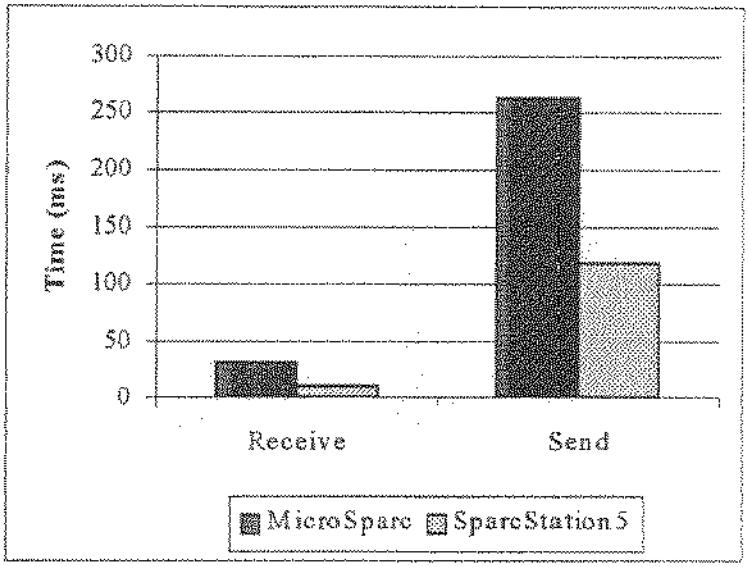

Figura 6.10: Cost for Method Interception and Re-sending.

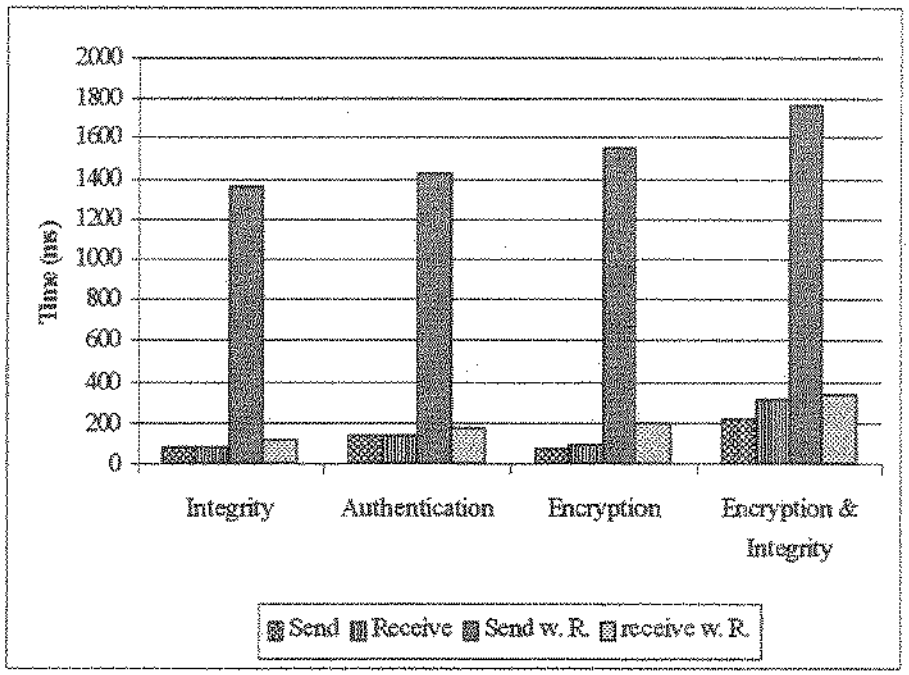

Figura 6.11: Cost for Some Cryptographic Transformations Performed by Meta Objects in a MicroSparc.

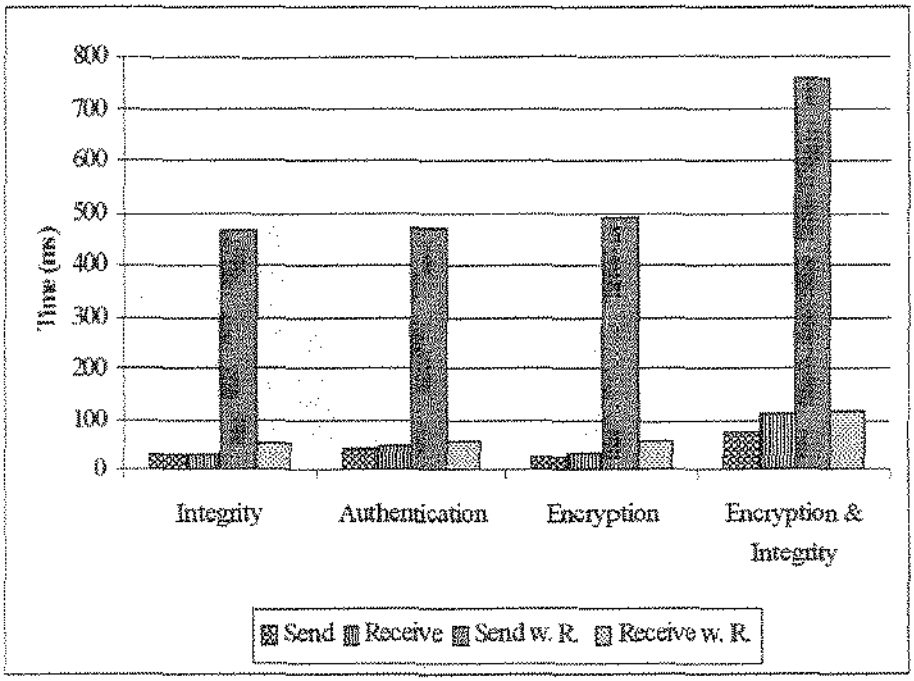

Figura 6.12: Cost for Some Cryptographic Transformations Performed by Meta Objects in a SparcStation-5.

only by a scale factor. These figures show that times for protecting receive() methods, with and without reflection, are very similar, despite method interception. Times for sending differ due to parsing and element substitution in the argument list of intercepted methods.

Figure 6.13 shows the cost in time for storing and loading objects, either alone or secured with encryption and integrity checking, for four different object sizes. The cost for storing, as well as the cost for encryption and integrity checking during storage, is directly proportional to object size. Storing is usually faster than loading, because the later requires not only decryption, but also a successful integrity checking.

Figure 6.14 contains the cost in time for storing and loading objects when security is performed by cryptographic meta objects. This cost is also proportional to object size and can be very high for larger objects due to their manipulation in meta level (that is, serialization, creation, encryption). This figure also shows the size of secured objects (beside input size) handled at meta level. Different from Figure 6.13, loading is faster than storing, because the cost for computational reflection is greater than the cost for cryptography transformations.

In general, the cost for performing cryptography transformation inside meta objects

measurements concerning digital signatures.

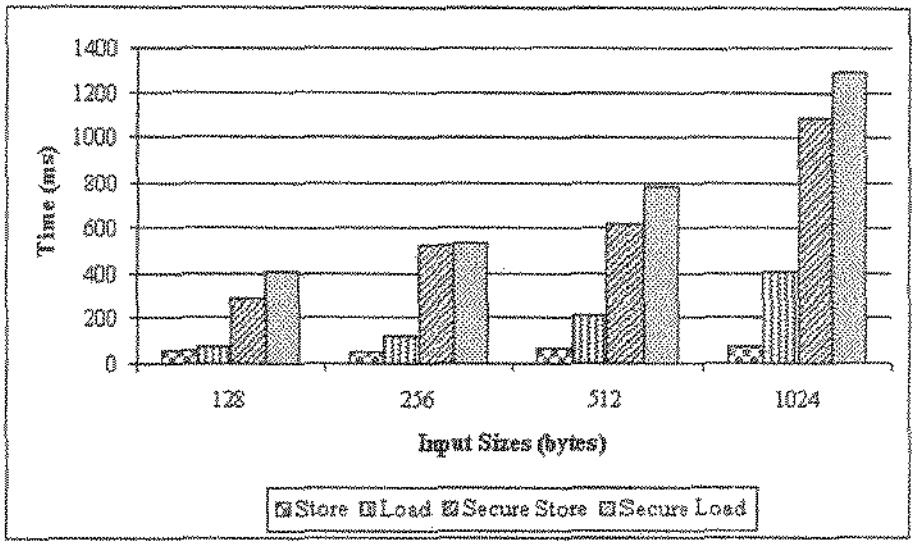

Figura 6.13: Cost for Storing Objects of Different Sizes, with and without Security in a MicroSparc 100 MHz.

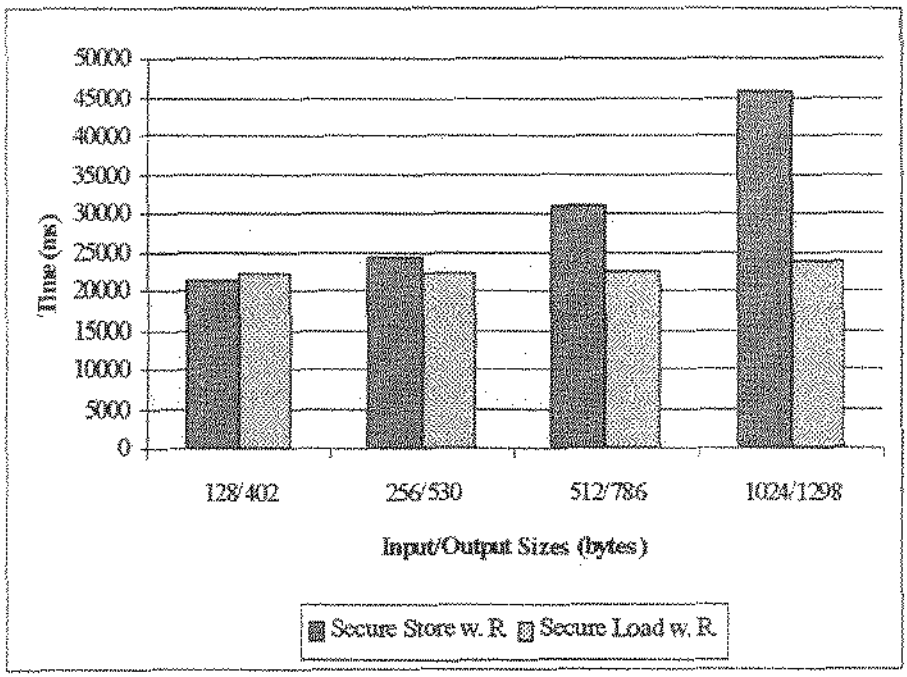

Figura 6.14: Cost for Storing Objects of Different Sizes, with Meta-Level Security in a MicroSparc 100 MHz.

is greater than the cost for performing them inside Alice and Bob. This conclusion was waited, however. It is important to notice that the impact of using cryptographic meta objects, instead of implementing it in base level, is not so high and, as technology evolves, can become cheaper. Another consideration is that the benefits of flexibility for reuse and ease of use can worth a manageable decrease in performance.

6.6 Conclusion and Future Work

As the necessity for computer communication grows, grows also the urgency for preserving privacy, integrity and authenticity of exchanged data. Open networks, such as the Internet, had found a variety of unexpected uses these days and cryptography-based security, formerly a military exclusive subject, are becoming a default feature in most modern software present in household desktop computers. Even those not interested in encryption can benefit from cryptography-based integrity checking and authentication mechanisms. Cryptographic software cannot still be developed as it used to be four decades ago. It is not economically feasible to rewrite cryptographic software every time a security function should be added to either a legacy system or a new project. Instead, cryptographic services should be able to reuse in large scale. In this text we have shown how modern software structuring techniques and concepts can be successfully applied to the development of cryptographic software in order to achieve ease of use and flexibility for reuse in the large.

The use of design pattern for implementing secure communication imposes discipline on programming along with advantages and disadvantages. Program understanding is easier when software is structured according to patterns. For example, the task of adapting a third-party software for security strengthen is greatly simplified if the aspect to be secured is localized, for instance, in a single class. However, this assumption cannot always be made for third-party software. The use of framework technology applied to cryptography is a promising subject. However, problems, as performance constraints and key management, still remain and suggest branches for future research.

Capítulo 7

ReflectiveMiMi: A Reflective Security Wrapper to the MiMi E-Commerce Tool

7.1 Introduction

Web-based electronic commerce appeared as a response to the common business problems of cost lowering in product distribution, brokers elimination in reaching customers, and market growing by exposition to a larger audience. The claim for secure electronic transactions in open networks, such as the Internet, led to the use, by commercial application, of cryptography-based security protocols. Traditionally free Internet resources, such as web pages, are becoming sources of sensitive data which should no more be accessed free of charge. This change of paradigm, from free unrestricted access to charged restricted use of network resources and information, produces a lack of software systems targeting the new fast-growing commercial branch of Internet. This lack can be filled out by new applications as well as adapted legacy systems.

Security is often a non-functional requirement, that not directly related to the main functionality of applications, which is usually neglected by software developers during rapid release cycles. In this situation even the more rudimentary security procedures can be forgotten. In other cases, application's requirements may change so that software should be adapted to a new context. For example, the charging of formerly free web pages requires not only a scheme for electronic payment, but also mechanisms for avoiding (or at least inhibiting) thieving of sensitive information.

This paper exemplifies a reflective approach to adapt third-party applications for stronger security requirements by (potentially transparent) addition of cryptography-based security facilities. ReflectiveMiMi is a reflective wrapper for addition of both privacy

protection and integrity checking to MiMi [VHH97]. MiMi is an electronic commerce tool for purchasing of web pages over the Internet, which encompasses an experimental implementation of the MicroMint [RS96] payment scheme.

The text is organized as follows. Section 7.2 offers an overview of MiMi and analyses some points in which its security can be improved. Section 7.3 describes the design issues, as well as some implementation tradeoffs, of ReflectiveMiMi. Section 7.4 presents some performance measurements. Conclusions and future work are in Section 7.5.

7.2 MiMi E-Commerce Tool

MiMi [VHH97] is an electronic commerce tool for purchasing web pages over the Internet, which encompasses an experimental implementation of the MicroMint payment scheme. MicroMint [RS96] is a micropayment scheme which provides security at a very low cost and is optimized for low-value unrelated payments. A strong requirement of this payment scheme is to completely avoid, for efficiency reasons, the use of public-key operations. It was designed to discourage large-scale attacks, such as massive coin forgery or persistent double spending caused by coin thieving. Micropayments are payments of very low value, done very quickly, possibly at high frequencies. The support for micropayments requires high efficiency from applications, especially during coin minting, otherwise the cost of the mechanism will exceed the value of the payment.

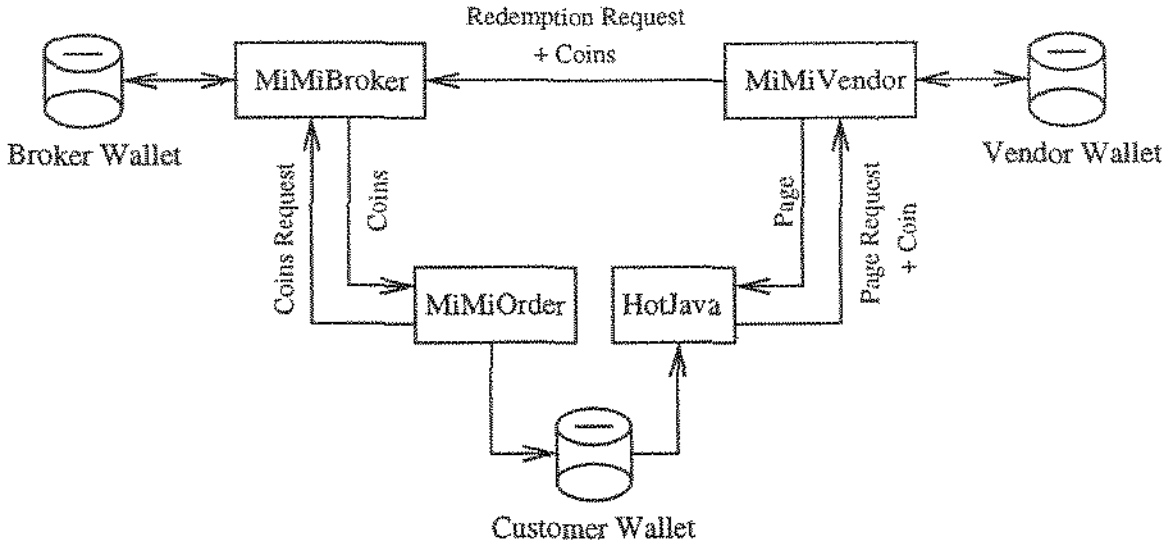

Figura 7.1: Flow of Sensitive Data through MiMi Participants.

Figure 7.1 shows the general organization of MiMi and the main flow of sensitive data (electronic coins and payed web pages) among its components. There are three

main entities: MiMiBroker, responsible for coin minting and issuing; MiMiVendor, the purchaser of web pages; and the customer. The customer part is divided in two modules: a coin ordering tool (MiMiOrder) and a modified HotJava browser, which also recognizes mimi links in addition to HTML ones. The customer orders coins from MiMiBroker and stores them in his wallet; a modified HotJava browser can then be used to withdraw electronic coins from the locally stored customer wallet and buy web pages identified by a mimi tag. Each web page costs one coin. MiMiVendor periodically performs coin redemption requests to MiMiBroker.

Since the focus of MiMi's design was the implementation of the micropayment scheme; some security requirements, as authentication of communicating parts and privacy (important even on micropayment-based applications), were neglected by the designers during implementation. For example, electronic coins are both stored in a wallet file and transmitted over the network in an unencrypted and non-authenticated format, leaving the present MiMi's implementation susceptible to eavesdropping, message tempering and masquerading. The rationale behind MiMi's low security is that the system can afford thieving of a few coins, while large-scale thieving is inhibited by the payment scheme.

7.3 ReflectiveMiMi Wrapper

ReflectiveMiMi is a reflective security wrapper for addition of both symmetric-key encryption and integrity checking to MiMi. It provides confidentiality, weak authentication and integrity while preserving MiMi's original functionality and performance requirements. This wrapper extends a reflective object-oriented framework for cryptography [BDR99a] based on *Guaraná* [OGB98], a meta-object protocol for the Java programming language.

Some spots for security improvement in MiMi, targeted by this wrapper, are the following:

- Authentication of vendors by brokers and encryption of the channel during redemption requests. This security improvement can prevent false vendors from being redeemed for possibly stolen coins.
- Authentication of customers by vendors and encryption of the channel in payment transactions. This security feature can prevent false customers from buying web pages by using stolen coins.
- Authentication of brokers by customers and encryption of the channel upon receiving new coins. This security improvement can prevent customers from receiving false coins.

- Encryption and integrity checking of coins during storage. This feature can prevent coin losses due to either accidental or intentional data corruption while inhibiting coin thieving.

Security aspects related to web pages were not covered in this work. Authentication of vendors by customers can assure customers receive the requested page. Pages can also be protected against modification by integrity checking. Privacy of customers may also be desirable.

ReflectiveMiMi's design follows the steps for using the reflective framework for cryptograhy [BRD99b]:

1. Determine the adequate cryptographic pattern for instantiation.
2. Find out the base-level application's hook interface.
3. Define the event flow, if any, from base level to meta level.
4. Find or implement an specialization of MetaLevelApp that addresses both the pattern from step 1 and the hook interface from step 2.
5. Implement a glue program for framework initialization and launching.

The cryptographic pattern instantiated by ReflectiveMiMi should avoid public-key operations, a constraint inherited from the payment scheme used by MiMi. On the other hand, it should provide not only secrecy and integrity, but also weak authentication, since strong authentication with public-key signatures cannot be used. This leads to reflective variation [BRD99a] of the *Secrecy with Integrity* [BRD98a] pattern with symmetric-key cryptography.

The tasks of transmiting coins through the network and storing them in wallets are handled by the class Coin itself. However, there are no methods in Coin which could be mapped to Alice.send() and Bob.receive(), the hook methods in cryptographic patterns. Methods Coin.readFrom() and Coin.writeTo(), shown in Figure 7.2, operate over streams and do not provide a hook interface with serializable arguments and returned results which could be handled by meta objects. In order to overcome this limitation, two methods were added to Coin: Coin.loadFrom(), called from Coin.readFrom(), provides the hook interface for interception and decryption of returned results and method Coin.storeTo(), called from Coin.writeTo(), offers the necessary interface for argument encryption after method interception.

Figure 7.2 shows the class hierarchy for electronic coins. Originally, class Coin only belonged to MiMi. However, in order to preserve performance requirements during coin minting and overcome implementation restrictions in the web browser, two more types

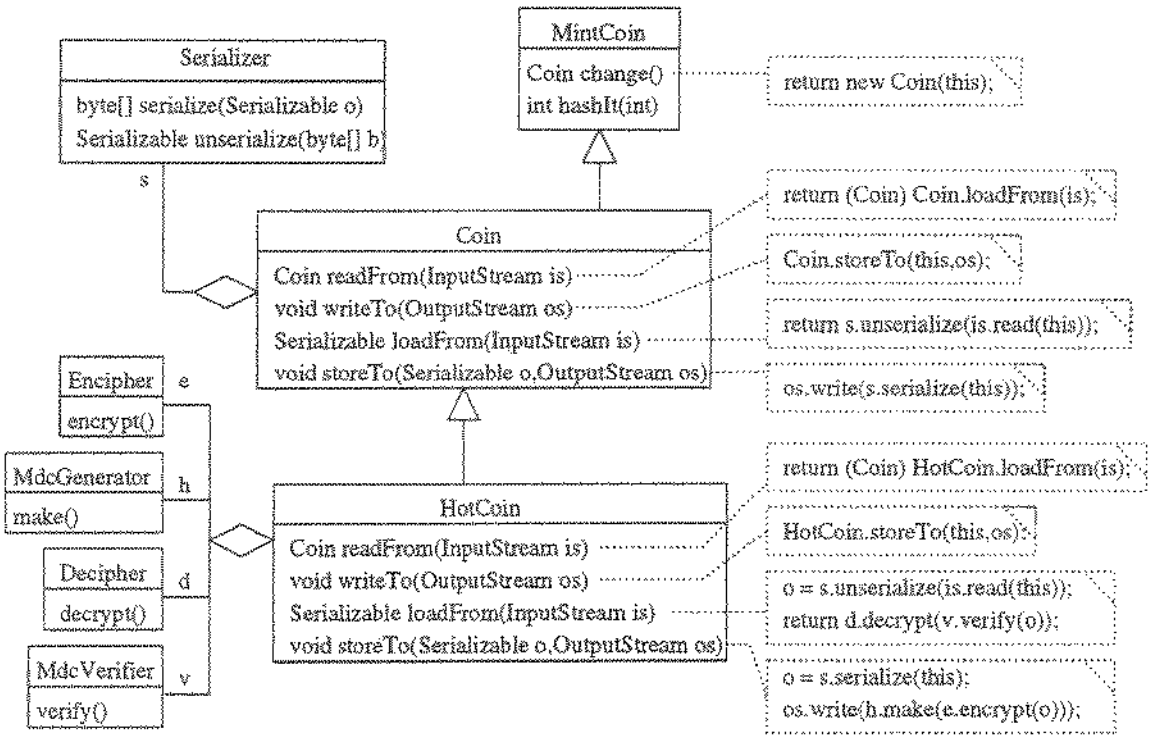

Figura 7.2: Coin Class Hierarchy.

of coins were added to MiMi. Class MintCoin is used by MiMiBroker only during coin minting in order to avoid the decrease of performance caused by Coin, subjected to method interception. Coin minting must be fast (a requirement of the payment scheme [RS96]), computational reflection can slowdown the rate of coin minting to unacceptable levels, so that its use should be avoided in this situation. After minting, by the time of ordering, MintCoin instances are converted by the method MintCoin.change() to Coin instances, susceptible to reflection.

The HotJava Browser should also be able to handle coins in a secure way. However, it is not possible to change browser's Java virtual machine to one capable of method interception, such as *Guaraná*, then cryptographic meta objects cannot be used. In this situation, encryption and integrity checking should be explicilty added to coins handled by HotJava. The class HotCoin, used only by Hotjava's MiMi handle, overloads methods Coin.loadFrom() and Coin.storeTo() and adds encryption and integrity to them, as shown by Figure 7.2. Again, by the time of payment, HotCoin instances are converted to Coin instances.

Its important to notice that MiMiOrder and MiMiVendor do not know either MintCoin, which cannot be reflected, or HotCoin, which has its own secure methods. Hence, encryption and integrity are added to these MiMi modules, in a complete transparent

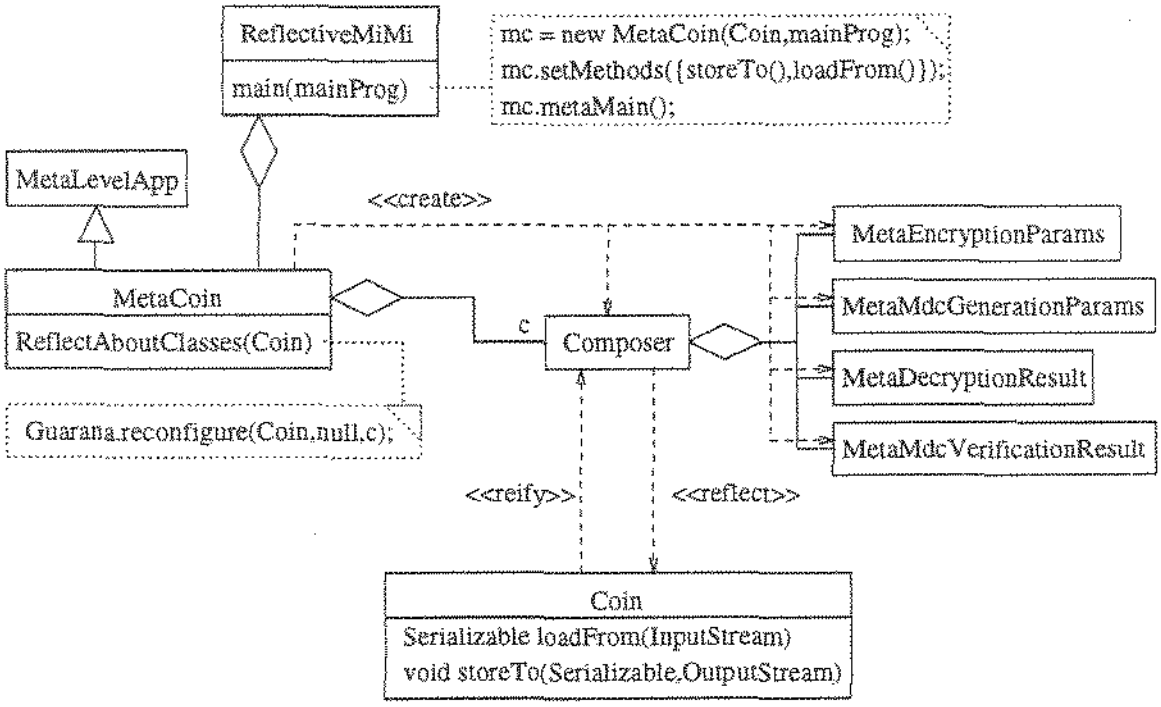

Figura 7.3: ReflectiveMiMi Class Diagram.

way, by intercepting method calls. Furthermore, MiMi does not have any knowledge about security features reflectiveMiMi adds to it. Hence, there is no control information from MiMi to its wrapper, which takes care of all aspects of security.

Figure 7.3 contains the class diagram, in UML-like notation, for ReflectiveMiMi applicaction. Classes MetaEncryptionParams, MetaMdcGenerationParams, MetaDecryptionResult, and MetaMdcVerificationResult perform cryptographic transformations and belong to the meta-object library. Class Composer is a meta object responsible for combinig the previous ones. Class MetaCoin, responsible for creating cryptographic meta objects and binding them (using method Guarana.reconfigure()) to class Coin, is created as a specialization of MetaLevelApp, reused from the cryptographic framework. Class ReflectiveMiMi is the glue between MiMi and MetaCoin. Its responsabilities are pass MetaCoin the main program's name (MiMiBroker, MiMiVendor or MiMiOrder) and inform MetaCoin which methods of class Coin should be intercepted and made secure, in this case Coin.writeTo() and Coin.readFrom(). An instance of Composer is the primary meta object of class Coin. Methods Coin.loadFrom() and Coin.storeTo() are static, so that their calls are always handled by class Coin. This means that instances of Coin do not need a meta configruration since those methods which worth intercepting belong to their class.

It is important to present some numbers on code reuse in order to illustrate the cost (in

terms of code rewriting and recompilation) for adapting MiMi to stronger security requirements. The original MiMi had 42 classes; three of them were modified and recompiled (MiMiOrder, Coin and RunURLConnection); four classes were added (reflectiveMiMi, HotCoin, MintCoin, and MetaCoin). This leads to 39 unchanged classes from a total of 46 and 84.8% of MiMi's code reuse.

The framework for cryptography has 160 classes to accomplish services for confidentiality, integrity, authentication and non-repudiation. 36 classes were used directly (through instantiation) or indirectly (through inheritance of their functionality by directly instantiated classes) by ReflectiveMiMi to accomplish confidentiality and integrity, reaching 22.5% of framework reuse. The final application has 82 classes and 91.5% of code reuse.

7.4 Performance Evaluation

The goal of the performance measurements below is to evaluate the impact of the meta-object library for cryptography [BDR99a] over a real application, the MiMi e-commerce tool. The time for transferring a pack of coins, with and without encryption, from a broker to a vendor, is measured, as well as the time required for purchasing a web page. Other performance measurements for this cryptographic framework can be found in [BRD99b].

The measurements were performed on a 100 MHz MicroSparc workstation running SunOS 5.5.1 (customer side) and a 110 MHz SparcStation-5 running SunOS 5.5.1 (broker and vendor sides). These machines belonged to the same local area network. The web browser was HotJava 1.1.5. The Java virtual machine was Kaffe openVM 1.0.b2 tempered with Guaraná 1.5.1. The cryptographic function for encryption was the DES symmetric-key algorithm in ECB mode. The MD5 hash function was used for integrity checking purposes and was applied after encryption and before decryption. A 10-iteration loop was used for measurement. Time, in milliseconds, was measured as the elapsed time between two calls of the System.currentTimeMillis() function. A MiMi's electronic coin is 219-byte-long in serialized form and 497-byte-long after encryption. Encryption here means the encapsulation of an encrypted form of the original object in a sealed object, which is also encapsulated (along with a modification detection code) by an integrity object. Decryption means a successful verification of the modification detection code, followed by object decryption and de-serialization.

Figure 7.4 compares the time required for MiMiOrder receive coins in two cases: (*i*) when it is launched by ReflectiveMiMi (indicated by the legend "With Reflection"), in which case coins are encrypted; and (*ii*) when it is executed directly (indicated by a "Without Reflection" legend). The bars in Figure 7.4 indicate that time for transmitting coins through the network is roughly the same, for a small number of coins (less then 100), without encryption and reflection, and proportional to the number of coins when

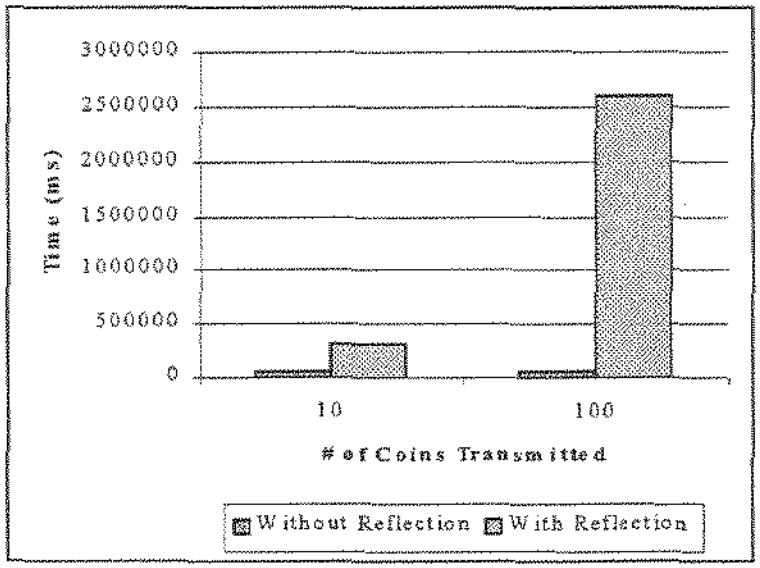

Figura 7.4: Time Required by MiMiOrder to Receive # Coins With and Without Encryption in the Meta Level.

reflection and encryption are used by both ends of the channel.

Figure 7.5 presents the time cost for encrypting a 128-byte-long object and decrypting it with integrity checking, when cryptography transformations are performed by the application itself (indicated by a "Without Reflection" legend) or delegated to meta objects (indicated by the legend "With Reflection"). Encryption takes roughly 250 ms, without reflection, and 1750 ms with reflection, the delay is due to method interception and re-sending.

Figure 7.6 presents the time consumed by both MiMi protocol handler, locally embedded in HotJava, MiMiVendor, with and without reflection, when performing secure payment transactions. A payment transaction is the delivering of a web page, by MiMiVendor, upon receiving an electronic coin from customer's browser. The transaction is said to be secure when coins are encrypted and protected against corruption. Figure 7.6 shows that MiMiVendor, under ReflectiveMiMi's control, is slower than when it is executed alone. An HTML file of 4 kbytes was used in this experiment. It should be noticed that file transfer takes most of time.

The apparent low performance of ReflectiveMiMi can be overcome by performing decryption on demand (that is, per payment) instead of on receiving. However, this adaptation would lead to a deeper intrusion into MiMi's code, reducing the presently high rate of code reuse. MiMi was design so that a few powerful classes perform a lot of functions. For instance, coins not only are a repository of sensitive data, but also perform hash calculations and streams input/output operations. Most of the security

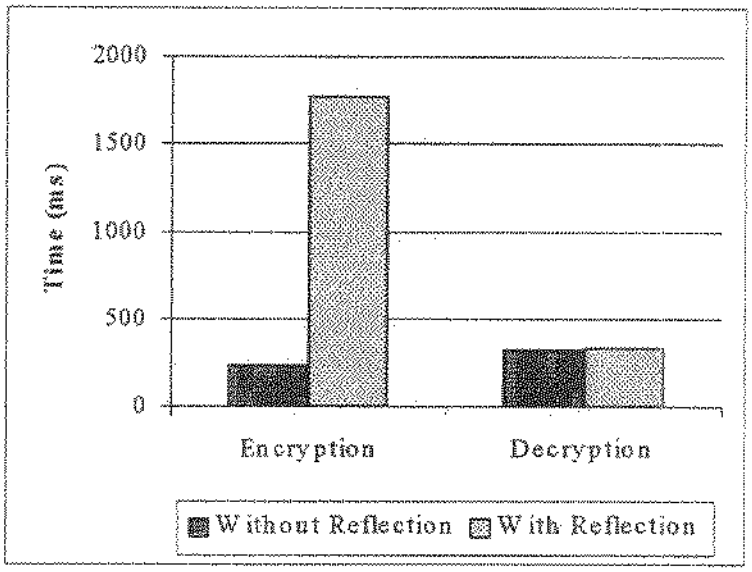

Figura 7.5: Time Required to Perform Encryption and Decryption with Integrity Checking, With and Without Reflection.

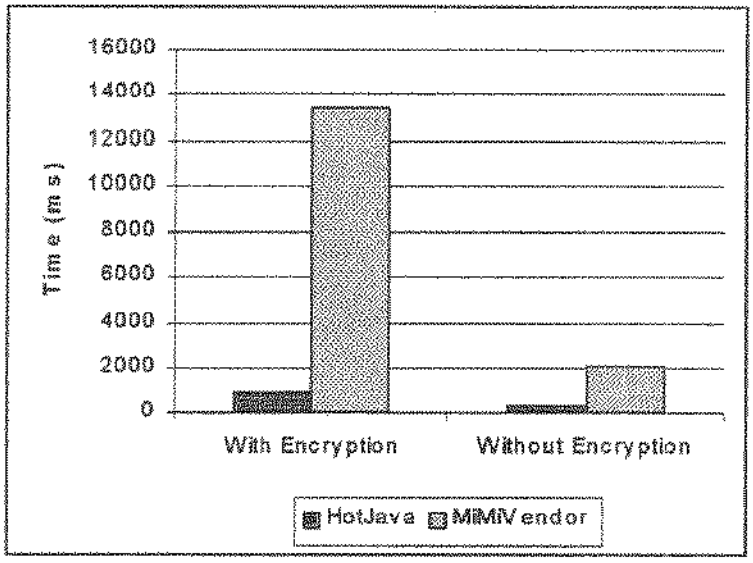

Figura 7.6: Time Consumed by HotJava (With Encryption) and MiMiVendor (With Encryption in the Meta Level) to Process a Purchase Transaction.

improvements should be made over coins. However, the I/O operations do not know whether they are being performed over either files or communication links, in such a way that the same security issues should be applied to storage and communication. The Splitting of Coin functionality into more classes would facilitate strong security policies adoption as well as offer better performance.

7.5 Conclusions and Future Work

This paper illustrates a reflective approach for adding security features to third-party applications, with minimal (potentially none) modification in target application's code. Addition of privacy, integrity and weak authentication to MiMi, an experimental electronic commerce application, was accomplished in three different manners:

1. Completely transparent addition to MiMiOrder and MiMiVendor.
2. Addition, after small adaptations on software, to MiMiBroker;
3. Explicit inclusion of cryptographic services calls to the MiMi protocol handler embedded in HotJava.

The addition of cryptography-based security features to third-party applications is a tradeoff among several, perhaps conflicting, forces and ranges from completely transparent addition of powerful security services to software re-design for security support with explicit implementation of security services. Some of these forces, identified in this project, are the following:

- The complexity of the security policy adopted. Different security services should be applied to different aspects of software functionality according to a great variety of criteria.
- The level of intrusion into target application. The amount of both knowledge about target application's internal behavior and modification necessary to support security services. Modification may not only be restricted to source code, but also reach design decisions and architectural issues. The strength of security in the resulting application is directly related to the original application's software architecture, which usually not contemplates security issues and may not be adequate for addition of security features a posteriori, requiring adaptaion.
- The observance of target application's original requirements. Addition of security features should not break critical aspects of software, usually supported by strong requirements.

The reflective object-oriented framework for cryptography was valitated by this case study. The knowledge obtained in this experiment suggests branches for future research in both architectural issues for electronic commerce applications and reflective implementation of security policies as an evolution to the current approach of treating security services.

7.6 Acknowledgments

This work was partially supported by the ALFA Project for interchange of post-graduates between Europe and Latin America. Is also used computational resources from the Department of Distributed Systems, Institute for Information Systems, Technical University of Vienna.

Capítulo 8

Conclusão Geral

Os mecanismos da criptografia (ciframento, integridade, autenticação e assinaturas) são utilizados em um conjunto limitado de cenários. Estes cenários estabelecem padrões de comportamento que, por sua vez, produzem um número também limitado de combinações simples. Este conjunto de padrões e suas combinações definem os usos comuns da criptografia na maioria das aplicações. Estes cenários ou padrões de utilização podem ser capturados por um arcabouço de software para criptografia capaz de proporcionar não só reutilização de algoritmos criptográficos, mas também dos próprios cenários.

A abordagem para adição de aspectos de segurança a software de terceiros e sistemas legados desenvolvida neste texto é potencialmente transparente. Entretanto, aplicações reais exigem um compromisso entre a complexidade da política de segurança adotada, a quantidade de conhecimento sobre o funcionamento interno do software alvo e a preservação dos requisitos originais.

Esta dissertação é a soma de uma sequência de resultados na qual cada parte fundamenta as suas sucessoras e é ao mesmo tempo validada por elas. Esta sequência de contribuições é a seguinte:

1. O estudo comparativo de bibliotecas criptográficas e suas interfaces de programação no qual os aspectos explorados nesta dissertação (reutilização em larga escala e facilidade de uso e composição) foram identificados.

2. A proposição de um conjunto coeso de dez padrões de projeto que documentam os aspectos arquiteturais do software criptográfico e as boas soluções conhecidas para os problemas comuns na segurança de informações baseada em criptografia.

3. A definição de um modelo de utilização para reflexão computacional no desenvolvimento de software criptográfico. Este modelo, definido como um padrão de projeto derivado da combinação dos padrões criptográficos e um padrão reflexivo de arquitetura, lança os fundamentos para um arcabouço de software para criptografia.

4. O desenvolvimento de um arcabouço de software reflexivo orientado a objetos para reutilização em larga escala de mecanismos criptográficos. Este arcabouço é composto de uma coleção de algoritmos criptográficos, uma biblioteca de metaobjetos especializados em transformações criptográficas sobre argumentos e resultados de métodos interceptados, e um conjunto de passos e algumas diretrizes para disciplinar o uso e a extensão do arcabouço.

5. A extensão de uma ferramenta experimental de pagamento eletrônico com o uso do arcabouço de software para criptografia.

Duas contribuições são consideradas principais. A primeira delas, caracterizada por aspectos abstratos e abordagem inovadora, é o conjunto fortemente coeso de padrões de projeto para criptografia. A outra, caracterizada por aspectos técnicos e usos práticos, é o arcabouço de software para criptografia.

As aplicações das idéias apresentadas nesta dissertação são as seguintes:

- Auxílio a engenheiros de software com pouca experiência em criptografia durante a identificação dos mecanismos criptográficos capazes de atender aos requisitos de segurança de seus sistemas.

- Adição pouco intrusiva (potencialmente transparente) de mecanismos criptográficos a software de terceiros e sistemas legados.

- Implementação rápida de serviços de segurança a partir de um arcabouço reutilizável de software.

Algumas questões em aberto e aspectos não tratados neste projeto são os seguintes:

1. A implementação de mecanismos elaborados para gerência de chaves e certificados. Atualmente, o arcabouço possui mecanismos bastante simples para armazenamento seguro de chaves criptográficas e geração de chaves a partir de senhas.

2. O tratamento de um número maior de serviços de segurança. A implementação atual do arcabouço trata os mecanismos criptográficos.

3. O uso do arcabouço criptográfico no desenvolvimento e extensão de um número maior de aplicações.

Pesquisas futuras se direcionam para o uso de técnicas da engenharia de software para auxiliar o desenvolvimento dos aspectos de segurança em sistemas distribuídos. Os seguintes tópicos despertam nosso interesse:

- A investigação de novos padrões de projeto relacionados não somente aos mecanismos criptográficos, mas também a outros aspectos da segurança de informações.
- A formalização da organização de alto nível de sistemas de comércio eletrônico como uma arquitetura de software específica para este domínio de aplicações.
- O apoio do arcabouço de software a políticas de segurança. Pode ser obtida com metaobjetos capazes de realizar planos ou sequências de ações no apoio a segurança de sistemas de computação.
- A utilização do arcabouço de software em sistemas de comércio eletrônico e agentes móveis.

Este projeto de pesquisa identificou a necessidade de uma abordagem diferenciada para os mecanismos criptográficos, estabeleceu modelos abstratos e padrões genéricos de arquitetura e comportamento das aplicações com requisitos de segurança baseados em criptografia, implementou estes modelos em um protótipo e os validou em situações reais.

Apêndice A

Basic Cryptographic Concepts

Modern cryptography is a broad subject, encompassing both the study and the use of mathematical techniques to address information security problems, such as confidentiality, data integrity and non-repudiation.

It can also be defined as the discipline that embodies the principles, means, and methods for the transformation of data in order to hide its semantic content, prevent its unauthorized use, or prevent its undetected modification [iso89]. Usually four objectives [MvOV96], or services [iso98], of cryptography are considered: confidentiality, integrity, authentication, and non-repudiation. Accordingly, there are four basic cryptographic mechanisms: encryption/decryption, MDC (Modification Detection Code) generation/verification, MAC (Message Authentication Code) generation/verification, and digital signing/verification. These four services can be combined in specific and limited ways to produce more specialized services. For further information about cryptography, see [MvOV96, Sch96, Sti95].

Confidentiality is the ability to keep information secret except from authorized users. Data integrity is the guarantee that 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 Mechanisms

Cryptographic transformations are mainly based on one-way functions, which are mathematical functions for which it is computationally easy to compute an output of an input, but it is computationally difficult to determine any input corresponding to a known output. One-way functions with trapdoors are one-way functions for which it is compu-

tationally feasible to compute an input corresponding to a known output, if additional information is provided (the trapdoor). One-way functions with trapdoors are the basic construction components of reversible cryptographic transformations in which the trapdoor information works as the cryptographic key.

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

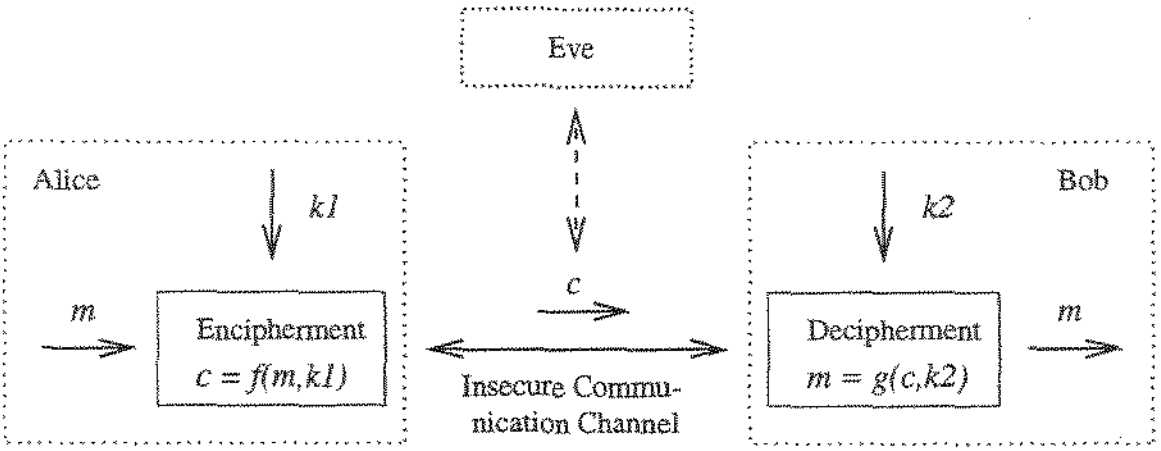

Figura A.1: A Typical Cryptosystem.

Traditionally, the two ends of a communication channel are called Alice and Bob. Eve is an adversary eavesdropping the channel. Figure A.1 shows the overall architecture of cryptosystems. 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.

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) and are used to guarantee data integrity. Message Authentication Codes (MACs) are usually implemented as hash values generated by cryptographic hash functions that take as input a secret key as well as the usual input stream. MACs are used to provide not only authentication, but also integrity implicitly.

Digital Signatures are electronic analogs of handwritten signatures, which serve both as the signer's agreement to the information a document contains and as evidence that can be shown to a third party in case of repudiation. A basic protocol of digital signatures based on public-key cryptography is: first, Alice encrypts a message with her private key to sign it; second, Alice sends the signed message to Bob; and third, Bob decrypts the received message with Alice's public key to verify the signature. Digital signatures must provide the following features: they are authentic, that is, when Bob verifies a message with Alice's public key, he knows she signed it; they are unforgeable, that is, only Alice knows her private key; they are not reusable, that is, the signature is a function of the data being signed, so it cannot be used with other data; and they cannot be repudiated, that is, Bob does not need Alice's help to prove she signed a message. 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 either Alice or Bob (or their secret shared key), all other attacks could be easily performed. Eve can attack a cryptosystem in four basic ways. First, she can eavesdrop the channel. Eavesdropping an open channel is easy. However, to understand eavesdropped messages of a cryptographically secured channel, the key (or keys) being used by Alice and Bob are required. Second, she can resend old messages. This attack is possible if messages do not have temporal uniqueness, which can be obtained using timestamps or by changing keys periodically. Third, 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. Fourth, 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 both 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 them.

A.3 Auxiliary Services

An important issue of implementations of cryptographic services is whether they are supported by an infrastructure that 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: it can be stored in tamper-proof hardware; it can be stored, in both encrypted and authentic forms, in general purpose hardware, such as random access memories, magnetic disks and tapes. This requires a key-encryption key which, in turn, must be protected.

Bibliografia

[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.

[BDR99a] Alexandre M. Braga, Ricardo Dahab, and Cecília M. F. Rubira. A Meta-Object Library for Cryptography. Technical Report IC-99-06, State University of Campinas, Institute of Computing, 1999.

[BDR99b] Alexandre M. Braga, Ricardo Dahab, and Cecília M. F. Rubira. Composing Cryptographic Services: A Comparison of Six Cryptographic APIs. Technical Report IC-99-05, State University of Campinas, Institute of Computing, 1999.

[BJ94] K. Beck and R. Johnson. Patterns Generate Architectures. In M. Tokoro and R. Pareschi, editors, *ECOOP'94*, pages 139–149. Springer-Verlag, 1994.

[BMA97] Davide Brugali, Giuseppe Menga, and Amund Aarten. The Framework Life Span. *Communications of the ACM*, 40(10):65–68, October 1997.

[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.

[Bra99] Alexandre M. Braga. ReflectiveMiMi: A Reflective Security Wrapper to the MiMi E-Commerce Tool. Technical Report TUV-1841-99-12, Technical University of Vienna, Information System Institute, Distributed System Department, July 1999.

[BRD98a] Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. Tropyc: A Pattern Language for Cryptographic Software. In *5th Pattern Languages of Programming (PLoP'98) Conference*, 1998. Washington University Technical Report #WUCS-98-25 and State University of Campinas Technical Report #IC-99-03 (Updated Version).

[BRD98b] Alexandre M Braga, Cecília M. F. Rubira, and Ricardo Dahab. Um Sistema de Padrões para Software Criptográfico Orientado a Objetos. In *XII Simpósio Brasileiro de Engenharia de Software*, pages 171–186, Maringá, Paraná, Brasil, October 1998.

[BRD99a] Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. A Reflective Variation for the Secure-Channel Communication Pattern. In *PLoP'99 Conference - 6th Pattern Languages of Programs*, 1999. State University of Campinas (Brazil) Technical Report #IC-99-04.

[BRD99b] Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. The Role of Patterns in an Object-Oriented Framework for Cryptography. Technical Report TUV-1841-99-11, Technical University of Vienna, Information System Institute, Distributed System Department, July 1999.

[BRD99c] Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. Tropyc: A Pattern Language for Cryptographic Object-Oriented Software. In Neil Harrison, Brian Foote, and Hans Rohnert, editors, *Pattern Languages of Program Design 4*, chapter 16. Addison-Wesley, 1999. to Appear.

[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.

[CS95] Jamer O. Coplien and Dougles C. Schmidt, editors. *Pattern Languages of Program Design 1*. Addison-Wesley, 1995.

[css97] Common Security Services Manager API, draft 2.0. 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.

[FP96] Jean-Charles Fabre and Tanguy Pérennou. Friends: A Flexible Architecture for Implementing Fault Tolerant and Secure Distributed Applications.

In Andrzej Hlawczka, João Gabriel Silva, and Luca Simoncini, editors, *Second European Dependable Computing Conference (EDCC-2)*, volume 1150 of *LNCS*, pages 3–20, Taormina, Italy, October 1996. Springer.

[FS97] Mohamed E. Fayad and Dougles C. Schmidt. Object-Oriented Application Frameworks. *Communications of the ACM*, 40(10):32–38, October 1997.

[gcs96] Generic Cryptographic Service API (GCS-API) – Base Draft 8, 1996. X/Open Preliminary Specification.

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

[GS98] Li Gong and Roland Schemers. Signing, Sealing and Guarding Java Objects. In G. Virgna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 206–216, Berlin Heidelberg, 1998. Springer-Verlag.

[Has97] Vesna Hassler. Internet Security: State-of-the-Art and Future Trends. Technical Report TUV-1841-97-08, Technical University Vienna, Information System Institute, Distributed System Department, April 1997.

[Her97] Michael Herfert. Security-Enhanced Mailing Lists. *IEEE Network*, pages 30–33, June 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*, pages 939–951, 1997.

[iso89] Information Processing Systems – Open Systems Interconnection – Basic Reference Model – Part 2: Security Architecture. ISO/IEC 7498-2, 1989.

[iso98] Information Technology – Vocabulary – Part 8: Security. ISO/IEC 2382-8, 1998.

[JBK98] Jonathan B. Knudsen. *Java Cryptography.* O'Reilly, 1998.

[JDK+91] Don B. Johnson, George M. Dolan, Michael J. Kelly, An V. Le, and Stephen M. Matyas. Common Cryptographic Architecture Cryptographic Application Programming Interface. *IBM Systems Journal*, 30(2):130–150, 1991.

[Joh92] Ralph Johnson. Documenting Frameworks Using Patterns. In *OOPSLA'92*, volume 27 of *ACM SIGPLAN Notices*, pages 63–76, October 1992.

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

[Lew96] T. Lewis. *Object-Oritented Application Frameworks*. Prentice-Hall, 1996.

[Lin93] J. Linn. Privacy Enhancement for Internet Electronic Mail, Part 1: Message Encipherment and Authentication Procedures. RFC 1421, February 1993.

[LMJW93] An V. Le, Stephen M. Matyas, Donald B. Johnson, and John D. Wilkins. A Public Key Extension to the Common Cryptographic Architecture. *IBM Systems Journal*, 32(3):461–485, 1993.

[Mae87] Pattie Maes. Concepts and Experiments in Computation Reflection. In *OOPSLA'87*, volume 22 of *ACM SIGPLAN Notices*, pages 147–155, December 1987.

[MCK97] Matthias Meusel, Krzysztof Czarnecki, and Wolfgang Köpf. A Model for Structuring User Documentation of Object-Oriented Frameworks Using Patterns and hypertext. In Mehmet Aksik and Satoshi Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 206–216. Springer-Verlag, 1997.

[MDOY98] Robert Macgregor, Dave Durbin, John Owlett, and Andrew Yeomans. *JAVA Network Security*. Prentice Hall, 1998.

[Mic] Microsoft Corporation. Using CryptoAPI. msdn.microsoft.com/library/sdkdoc/crypto/1using_80kp.htm.

[Mic96] Microsoft Corporation. Applications Programmer's Guide: Microsoft CryptoAPI. Version 2.0, 1996.

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

[msc] Comparison of the Open Group's GCS-API and Microsoft CryptoAPI V1.0. www.opengroup.org/public/tech/security/gcs/ms_comp.htm.

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

[Oak98] Scott Oaks. *Java Security*. O'Really & Associates, 1998.

[OB98a] Alexandre Oliva and Luiz Eduardo Buzato. Composition of Meta-Objects in Guaraná. Technical Report IC-98-33, Institute of Computing, State University of Compinas, September 1998.

[OB98b] Alexandre Oliva and Luiz Eduardo Buzato. Guaraná: A tutorial. Technical Report IC-98-31, Institute of Computing, State University of Campinas, September 1998.

[OB98c] Alexandre Oliva and Luiz Eduardo Buzato. The Implementation of Guaraná on Java. Technical Report IC-98-32, Institute of Computing, State University of Campinas, September 1998.

[OGB98] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The Reflexive Architecture of Guaraná. Technical Report IC-98-14, Institute of Computing, State University of Campinas, April 1998.

[OQC97] Georg Odenthal and Klaus Quibeldey-Cirkel. Using Patterns for Design and Documentation. In Mehmet Aksik and Satoshi Matsuoka, editors, *ECOOP'97*, volume 1241 of *LNCS*, pages 206–216. Springer-Verlag, 1997.

[Pre95] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.

[RS96] Ronald L. Rivest and Adi Shamir. PayWord and MicroMint: Two Simple Micropayment Schemes. http://theory.lcs.mit.edu/sinrivest/RivestShamir-mpay.ps, 1996. Presented at the RSA'96 Conference.

[Sch95] Dougles C. Schmidt. Using Design Pattern to Develop Reusable Object-Oriented Communication Software. *Communications of the ACM*, 38(10):65–74, October 1995.

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

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives in an Emerging Discipline.* Alan Apt, 1996.

[SRC84] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Science*, 2(4):277–288, November 1984.

[Sti95] Douglas R. Stinson. *Cryptography Theory and Practice.* CRC Press, 1995.

[SW96] Robert J. Stroud and Zhixue Wu. Using Metaobject Protocols to Satisfy Non-Functional Requirements. In Chris Zimmermann, editor, *Object-Oriented Metalevel Architectures and Reflection*, chapter 3, pages 31–52. CRC Press, 1996.

[Tea97] NSA Cross Organization CAPI Team. Security service API: Cryptographic API Recomendation Updated and Abridged Edition. Technical report, The National Security Agency, Ft. Meade, Meryland, July 1997.

[VHH97] Michael Fischer Vesna Hassler, Robert Bihlmeyer and Manfred Hauswirth. MiMi: A Java Implementation of the Micromint Scheme. In *World Conference of the WWW, Internet and Intranet (WebNet'97)*, Toronto, Canada, 1997. Technical University of Vienna Technical Report #TUV-1841-97-04.

[Wel97] Ian Welch. Adding Security to Commercial Off-the-Shelf Software. In *European Research Seminar on Advances in Distributed Systems (ERSADS)*, Zinal, Switzerland, March 1997.

[WS98a] Ian Welch and Robert Stroud. Adaptation of Connectors in Software Architectures. In *ECOOP'98 Workshop on Component-Oriented Programming*, Brusells, Belgium, July 1998.

[WS98b] Ian Welch and Robert Stroud. Dynamic Adaptation of the Security Properties of Applications and Components. In *ECOOP'98 Workshop on Distributed Object Security*, Brussels, Belgium, July 1998.

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

[Zim95] P. R. Zimmermann. *The Official PGP User's Guide.* MIT Press, 1995.