A Meta-Object Library for Cryptography

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

Relatório Técnico IC–99-06

Fevereiro de 1999

A Meta-Object Library for Cryptography

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

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

Abstract

This work describes a meta-object library and a reflective object-oriented framework for cryptography-based security, which focuses on three points: the easy reuse of cryptography-aware code, the easy composition of cryptographic services and the transparent addition of cryptography-based security to third-party code, from the application programmer's point of view. Such a framework is applicable to both third-party commercial-off-the-shelf applications and legacy systems, and is based on a reflective variation of a pattern language for cryptographic software we had proposed. We are using *Guaraná*, a reflective variation for the Java programming language which encourages composition of meta-objetcs, to implement our framework.

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 2 reviews the main cryptographic services and the role of cryptographic patterns. Section 3 approaches the main aspects of MOLC's design. The design issues in adding security to third-party applications are in Section 4. The meta level reconfigration policy is treated in Section 5. Section 6 outlines implementation issues. Conclusion and future work are in Section 7.

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 [BRD99] 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 subsystem and are treated by the application itself. The cryptographic design patterns corresponding to the basic services and their valid compositions are summarized in Table 2. The Generic Object-Oriented Cryptographic Architecture (GOOCA) is an abstraction for the patterns' 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 [BDR99a] 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	**Scope**	**Purpose**
GOOCA	Generic	generic software architecture for cryptography-aware applications
Information Secrecy	Confidentiality	provide secrecy of information
Message Integrity	Integrity	detect corruption of a message
Sender Authentication	Authentication	authenticate the origin of a message
Signature	Non-repudiation	provide the authorship of a message
Signature with Appendix	Non-repudiation	separate message from signature
Secrecy with Integrity	Confidentiality and Integrity	detect corruption of a secret
Secrecy with Sender Authentication	Confidentiality and Authentication	authenticate the origin of a secret
Secrecy with Signature	Confidentiality and Non-repudiation	prove the authorship of a secret
Secrecy with Signature with Appendix	Confidentiality and Non-repudiation	separate secret from signature

Table 2: The Cryptographic Design Patterns and Their Purposes.

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.

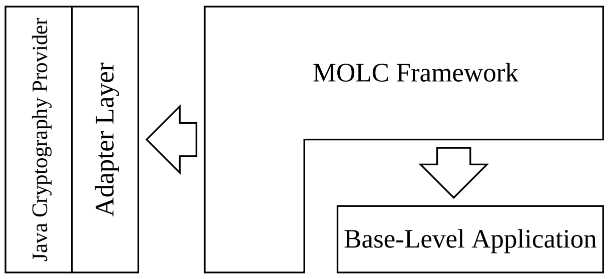

Figure 1: Architecture of MOLC.

The general software architecture for this meta-object library is shown in Figure 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 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.

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 by 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.

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 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 2, MetaCryptoEngine has four direct subclasses, divided in two pairs. MetaTransformationParams and MetaReverseTransformationParams are responsible for performing transformations and their reverses over parameters. MetaTransformation-Result 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.

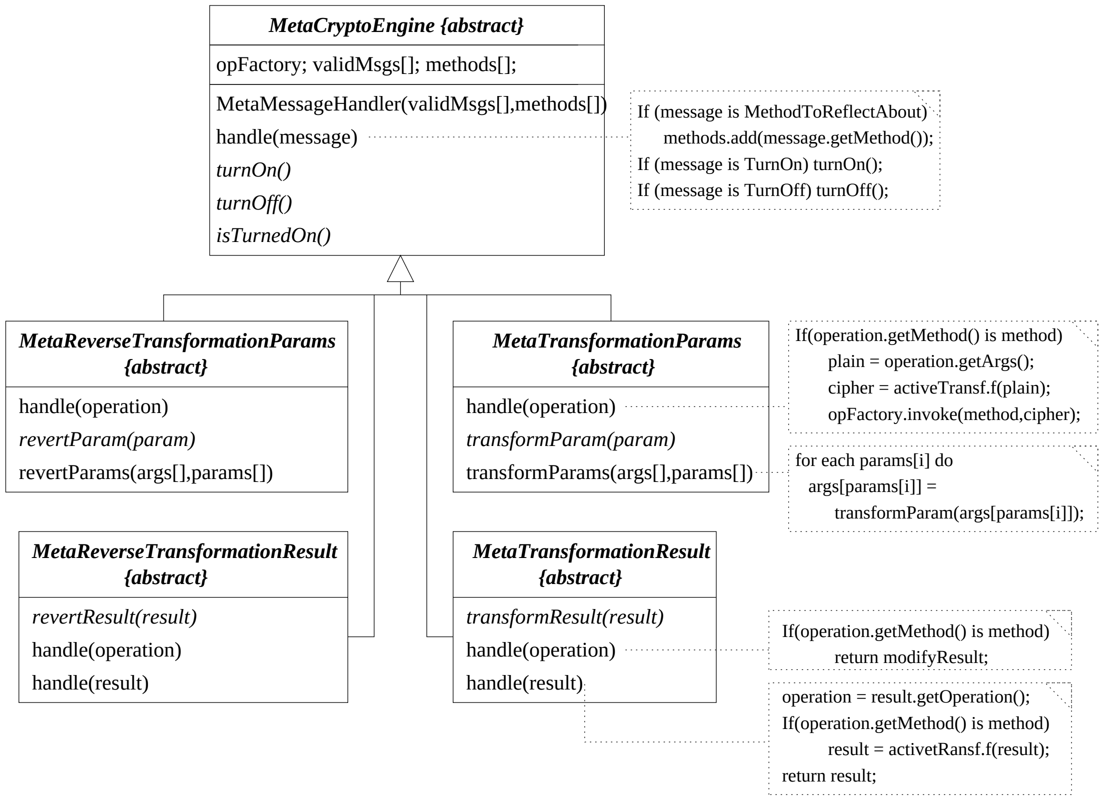
MetaCryptoEngine {abstract}
opFactory; validMsgs[]; methods[];
MetaMessageHandler(validMsgs[],methods[])
handle(message)
turnOn()
turnOff()
isTurnedOn()
If (message is MethodToReflectAbout)
methods.add(message.getMethod());
If (message is TurnOn) turnOn();
If (message is TurnOff) turnOff();
MetaReverseTransformationParams {abstract}
handle(operation)
revertParam(param)
revertParams(args[],params[])
MetaTransformationParams {abstract}
handle(operation)
transformParam(param)
transformParams(args[],params[])
If(operation.getMethod() is method)
plain = operation.getArgs();
cipher = activeTransf.f(plain);
opFactory.invoke(method,cipher);
for each params[i] do
args[params[i]] =
transformParam(args[params[i]]);
MetaReverseTransformationResult {abstract}
revertResult(result)
handle(operation)
handle(result)
MetaTransformationResult {abstract}
transformResult(result)
handle(operation)
handle(result)
If(operation.getMethod() is method)
return modifyResult;
operation = result.getOperation();
If(operation.getMethod() is method)
result = activetRansf.f(result);
return result;

Figure 2: Abstract Meta-Objects for Generic Transformations.

3.2.1 Specific Cryptographic Transformations

There are four subclasses for each meta transformation of Figure 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 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.

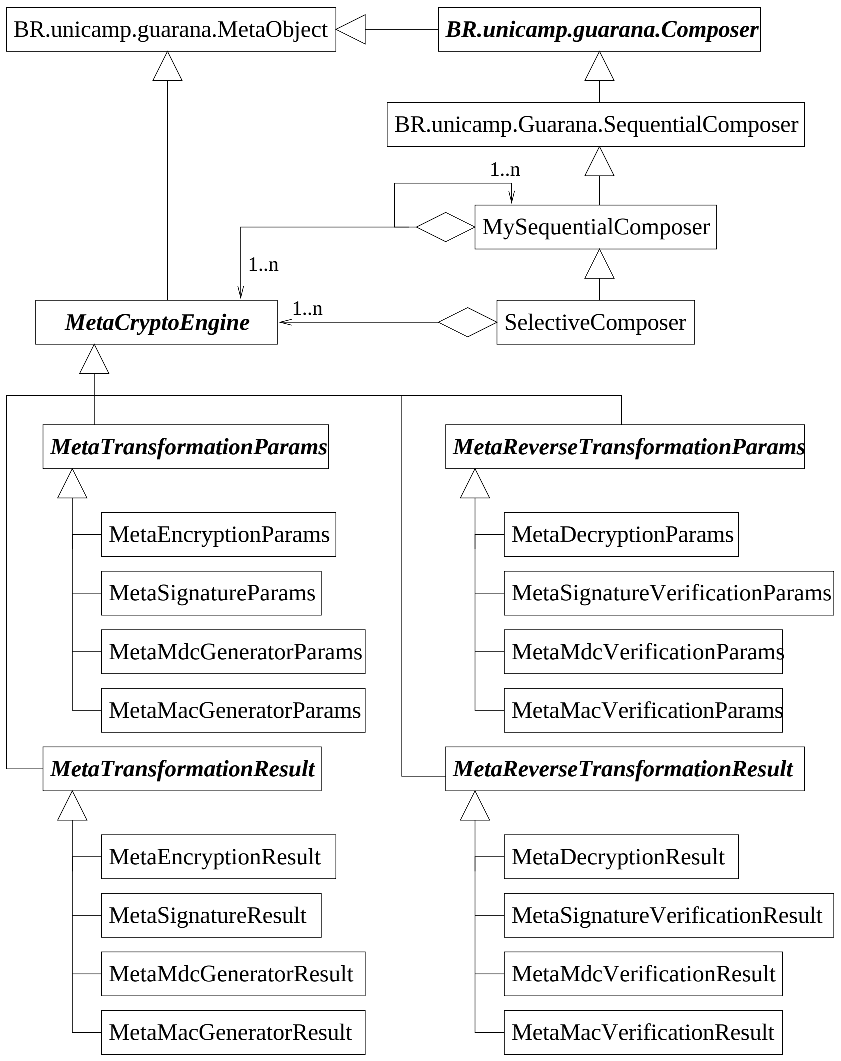

Figure 3: Meta-Objects for Cryptographic Services and Their Compositions.

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 [BRD99] 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.

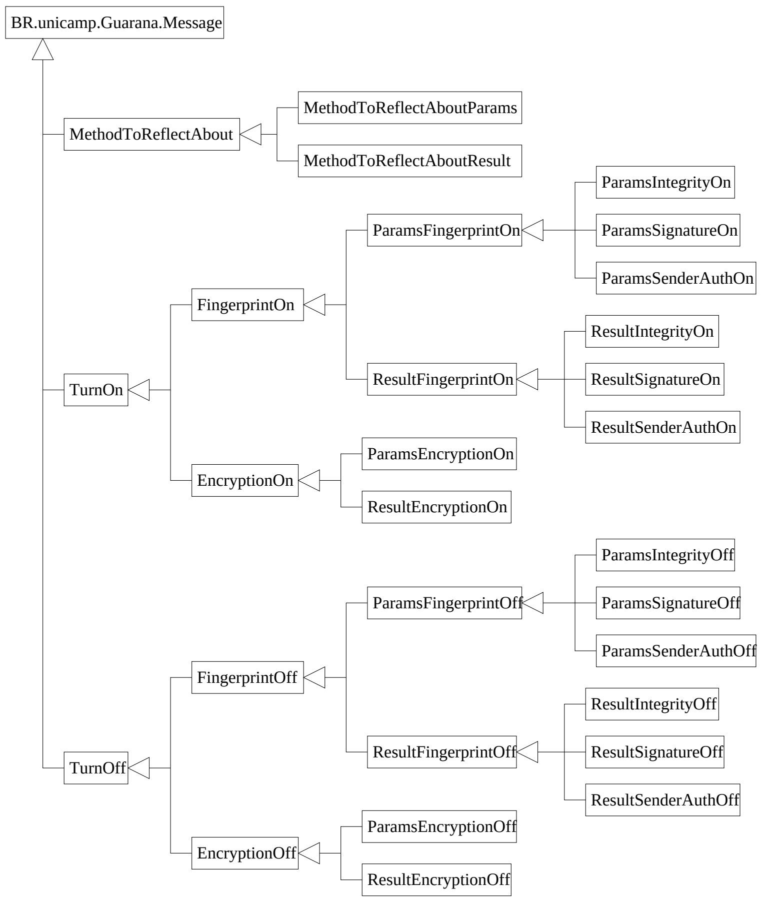

Figure 4: Message Hierarchy for Cryptographic Service Selection.

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 4. The static relations among these three meta objects are shown in Figure 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 cryptographic meta objects either individually or in compositions, is an instantiation of the reflective cryptographic pattern [BDR99a]. An interesting property of our implementation is that cryptography-aware meta objects can be composed in any order.

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.

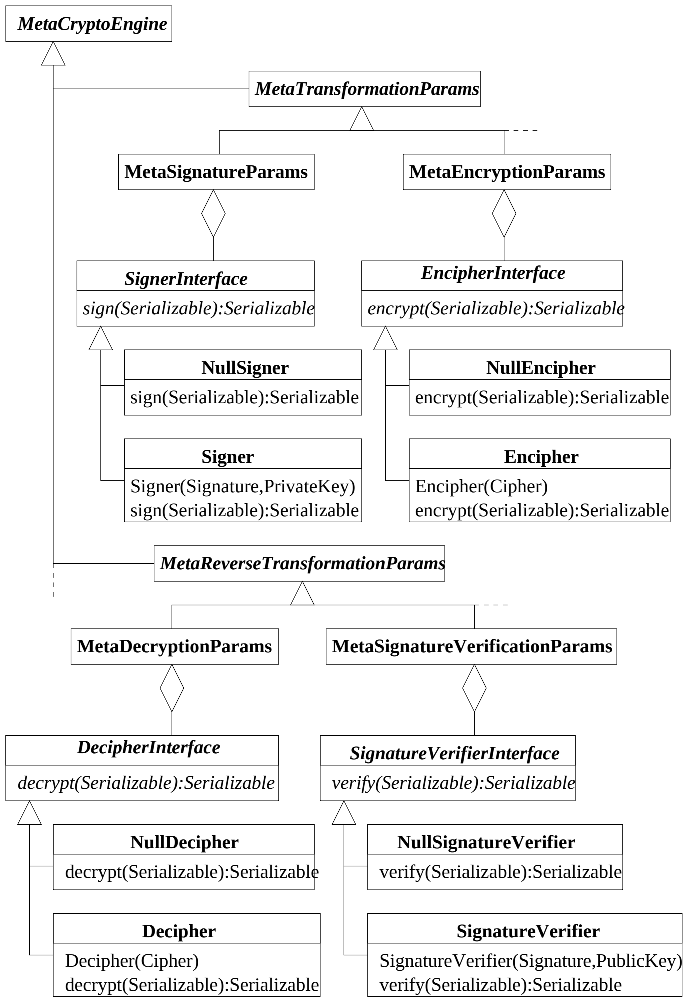
MetaCryptoEngine
MetaTransformationParams
MetaSignatureParams
MetaEncryptionParams
SignerInterface
sign(Serializable):Serializable
EncipherInterface
encrypt(Serializable):Serializable
NullSigner
sign(Serializable):Serializable
NullEncipher
encrypt(Serializable):Serializable
Signer
Signer(Signature,PrivateKey)
sign(Serializable):Serializable
Encipher
Encipher(Cipher)
encrypt(Serializable):Serializable
MetaReverseTransformationParams
MetaDecryptionParams
MetaSignatureVerificationParams
DecipherInterface
decrypt(Serializable):Serializable
SignatureVerifierInterface
verify(Serializable):Serializable
NullDecipher
decrypt(Serializable):Serializable
NullSignatureVerifier
verify(Serializable):Serializable
Decipher
Decipher(Cipher)
decrypt(Serializable):Serializable
SignatureVerifier
SignatureVerifier(Signature,PublicKey)
verify(Serializable):Serializable

Figure 5: Relationships among Meta-Objects and Adapters.

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. 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 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 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.

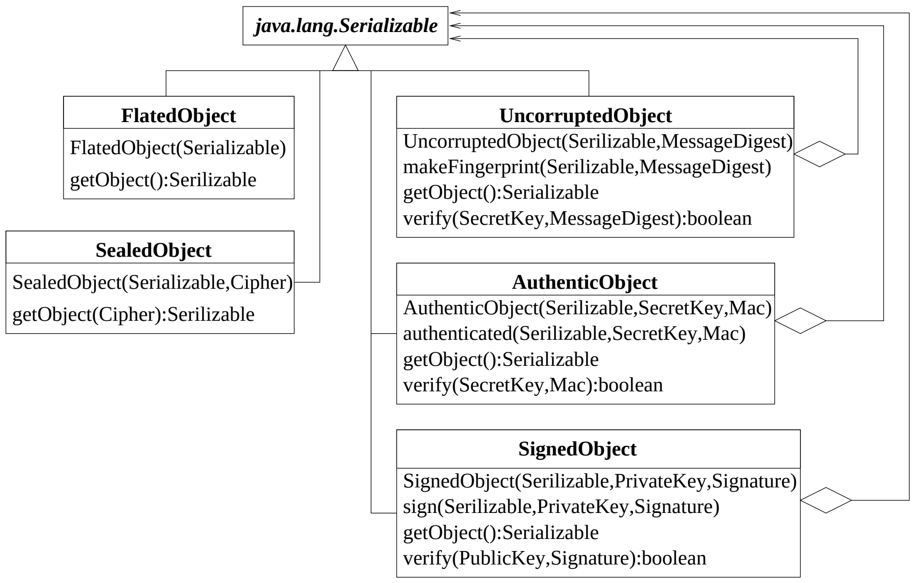
java.lang.Serializable
FlatedObject
FlatedObject(Serializable)
getObject():Serilizable
SealedObject
SealedObject(Serializable,Cipher)
getObject(Cipher):Serilizable
UncorruptedObject
UncorruptedObject(Serilizable,MessageDigest)
makeFingerprint(Serilizable,MessageDigest)
getObject():Serializable
verify(SecretKey,MessageDigest):boolean
AuthenticObject
AuthenticObject(Serilizable,SecretKey,Mac)
authenticated(Serilizable,SecretKey,Mac)
getObject():Serializable
verify(SecretKey,Mac):boolean
SignedObject
SignedObject(Serilizable,PrivateKey,Signature)
sign(Serilizable,PrivateKey,Signature)
getObject():Serializable
verify(PublicKey,Signature):boolean

Figure 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

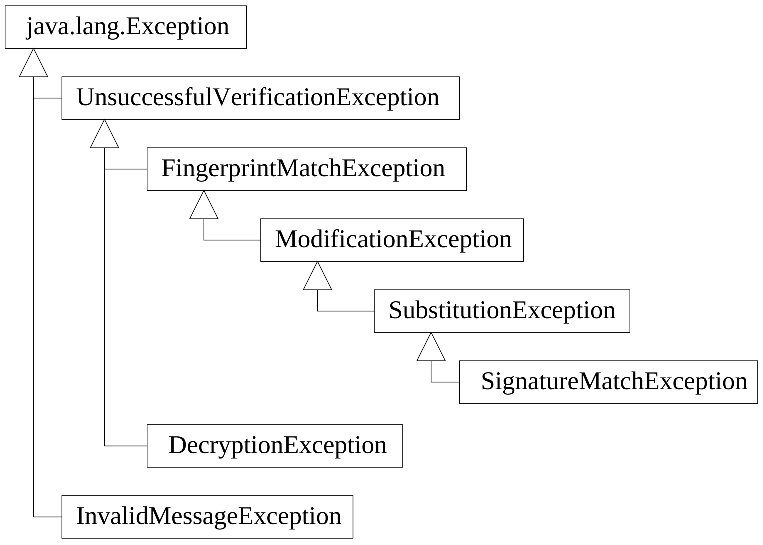

Figure 7: Cryptographic Service Verification Exceptions.

the class hierarchy of Figure 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. Configurable-Composers and MetaCryptoEngines throw an InvalidMessageException upon receiving an unknown Message's instance. All the exceptions of Figure 7 can be encapsulated in a *Guaraná*'s MetaException.

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.

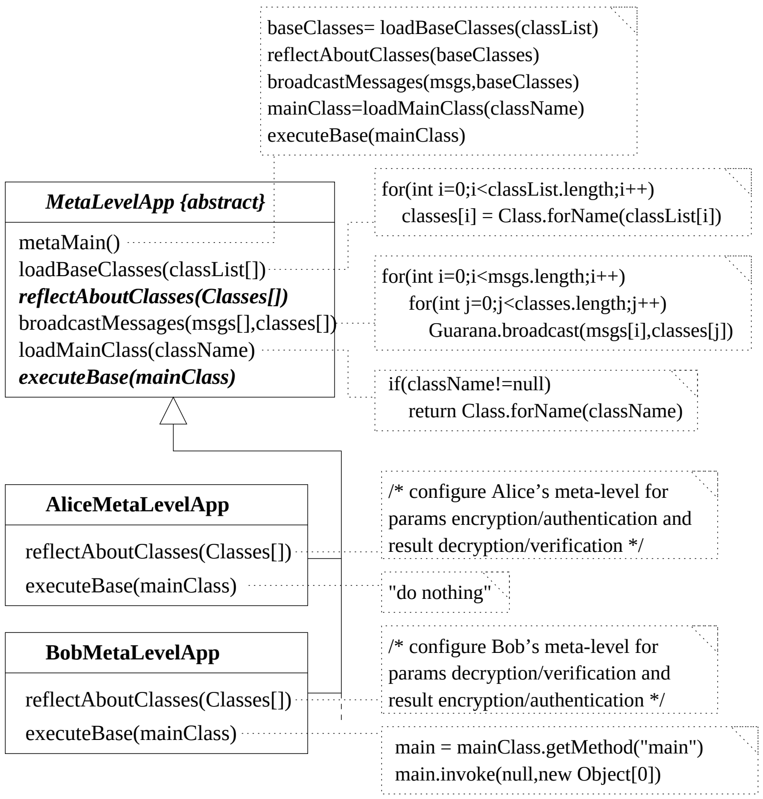
baseClasses= loadBaseClasses(classList)
reflectAboutClasses(baseClasses)
broadcastMessages(msgs,baseClasses)
mainClass=loadMainClass(className)
executeBase(mainClass)
MetaLevelApp {abstract}
metaMain()
loadBaseClasses(classList[])
reflectAboutClasses(Classes[])
broadcastMessages(msgs[],classes[])
loadMainClass(className)
executeBase(mainClass)
for(int i=0;i<classList.length;i++)
classes[i] = Class.forName(classList[i])
for(int i=0;i<msgs.length;i++)
for(int j=0;j<classes.length;j++)
Guarana.broadcast(msgs[i],classes[j])
if(className!=null)
return Class.forName(className)
AliceMetaLevelApp
reflectAboutClasses(Classes[])
executeBase(mainClass)
/* configure Alice's meta-level for params encryption/authentication and result decryption/verification */
"do nothing"
BobMetaLevelApp
reflectAboutClasses(Classes[])
executeBase(mainClass)
/* configure Bob's meta-level for params decryption/verification and result encryption/authentication */
main = mainClass.getMethod("main")
main.invoke(null,new Object[0])

Figure 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 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 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 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.

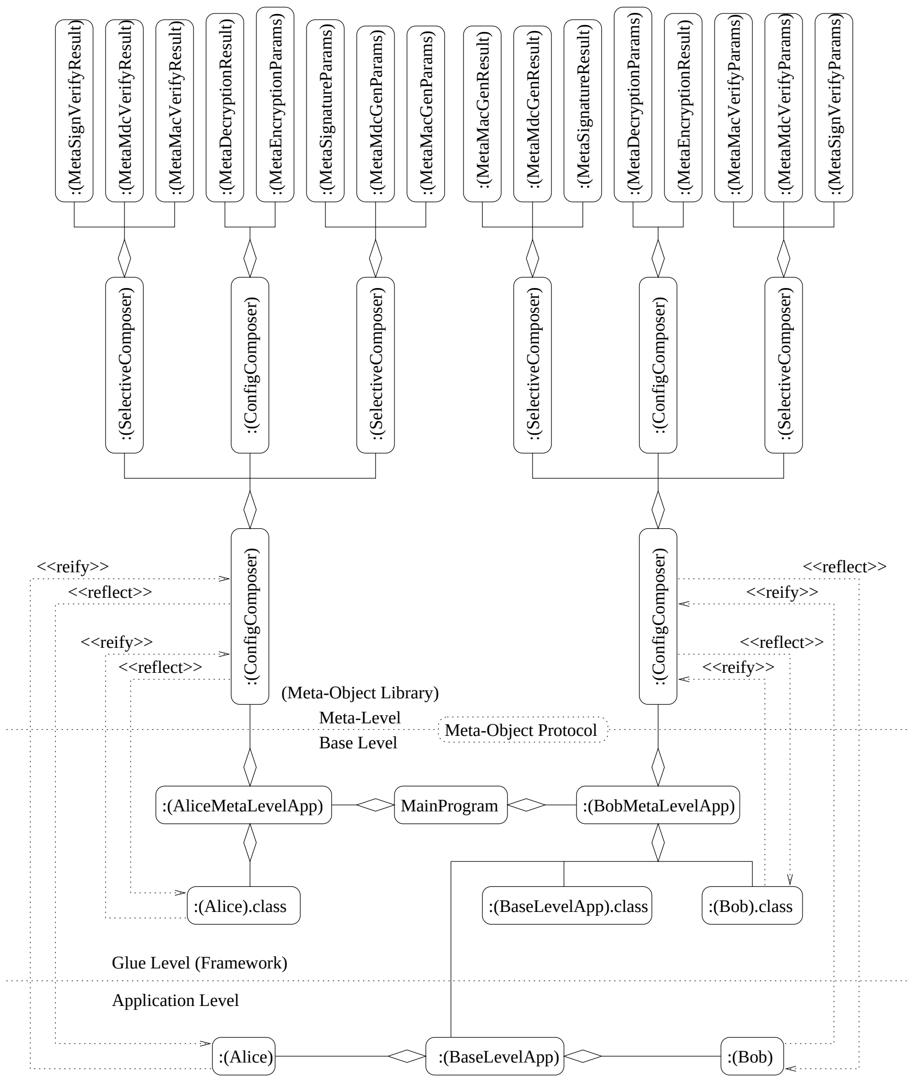

Figure 9: Runtime configuration for the example application.

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 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 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 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:

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

Figure 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.

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 9 and outlines the implementation of the hooks of Figure 8.

`MetaLevelApp`'s subclasses implement the abstract method `reflectAboutClasses()`, which is responsible for creating meta configurations. The code below 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.

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

                                          ⋮

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

                                          ⋮
```

A ConfigurableComposer, a_cc, contains instances of both MetaEncryptionParams and MetaDecryptionResult. A SelectiveComposer's instance, a_sc1, contains an array of MetaEncryptionEngine, which contains instances of MetaMacGenerationParams, MetaMdcGenerationParams and MetaSignatureParams. Another MetaEncryptionEngine's instance, a_sc2, 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.

```
                                        ⋮
    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);
}
```

MetaLevelApp's execute() 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.

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

Meta proxies for cryptographic keys can be created in the following way. 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.

```
                                        ⋮
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"));
                                        ⋮
```

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, below, is responsible for initializing the meta configuration from a secure state, creating adapters and 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.

```
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();
    }
}
```

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.

8 Acknowledgments

This work is partially supported by CNPq; PRONEX-Finep, grant 107/97 for Ricardo Dahab and FAPESP, grant 97/11128-3 for Alexandre Melo Braga and 96/15329 for LSD-IC-UNICAMP.

References

[BDR99a] Alexandre M. Braga, Ricardo Dahab, and Cecília M. F. Rubira. A Reflective Variation for the Cryptographic Design Pattern. Technical Report IC-99-04, 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.

[BRD99] Alexandre M. Braga, Cecília M. F. Rubira, and Ricardo Dahab. *Tropyc*: A pattern language for cryptographic software. Technical Report IC-99-03, State University of Campinas, Institute of Computing, 1999.

[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.

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

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

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

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

[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.

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