A Reflective Variation for the Cryptographic Design Pattern

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

Relatório Técnico IC–99-04

Janeiro de 1999

A Reflective Variation for the Cryptographic Design Pattern[†]

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

State University of Campinas - Institute of Computing
P.O. Box 6176. Campinas-SP-Brazil, 13083-970
phone/fax:+55+19+7885842
e-mail:{972314,rdahab,cmrubira}@dcc.unicamp.br

Abstract

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. In this work, the combination of computational reflection and cryptographic design patterns is proposed in order to improve reuse of both design and code (while decreasing coupling and increasing flexibility) of cryptographic components. The usefulness of this approach is in the transparent addition of cryptography-based security to third-party commercial-off-the-shelf components. A reflective extension for the cryptographic design pattern [BRD98a] is proposed. This extension is implemented in *Guaraná* [OGB98], a meta-object protocol for Java.

Key Words: computational reflection, design pattern, cryptography-based security.

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 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]. The encapsulation of authentication facilities

[†]This paper was accepted to the IDEAS'99 2nd. Ibero-American Workshop on Requirements Engeneering and Software Environments, San José, Costa Rica, March 1999

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 R-GOOCA, a refinement of the Generic Object-Oriented Cryptographic Architecture (GOOCA) proposed in [BRD98b, BRD98a], in order to decouple objects responsible for cryptographic services from the application objects. This pattern is produced by combining the *Reflection* [BMR+96, 193] architectural pattern and our cryptographic patterns. The contribution of this work is the proposal of a design pattern obtained by the combination of GOOCA [BRD98a] pattern and *Reflection* [BMR+96, 193] architectural pattern. R-GOOCA 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. The diagrams in this paper are presented using Gamma *et al.'s* notation [GHJV94]. For those readers interested in cryptography techniques two good sources are [Sch96, MvOV96, Sti95].

2 The Reflective Cryptographic Design Pattern

Context We have proposed a pattern language for cryptographic software which is composed by a set of nine design patterns [BRD98a]: *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 features, such as secrecy, integrity, authentication and non-repudiation, for secure communication and storage applications. These patterns share the same structure and dynamic behavior. Such aspects can be abstracted in a Generic Object-Oriented Cryptographic Architecture (GOOCA). They do not, however, capture, explicitly, the design of cryptographic services as non-functional requirements.

Figure 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-ware 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, $x = g(f(x))$. The transformation and its reverse are based on the same cryptographic algorithm. The objects' interaction diagram is shown in Figure 2.

A limitation of this design is that it forces functional objects (instances of the classes Alice and Bob) to explicitly take care of non-functional (cryptography-ware) 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.

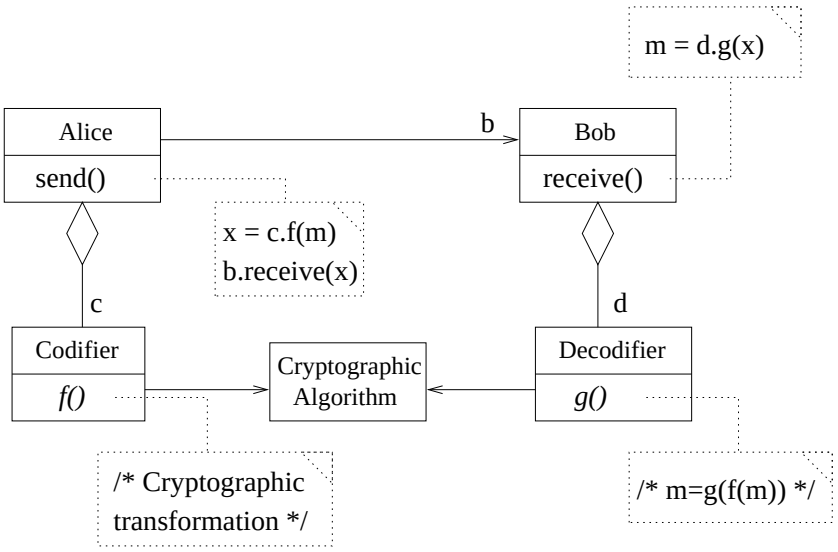

Figure 1: GOOCA Structure.

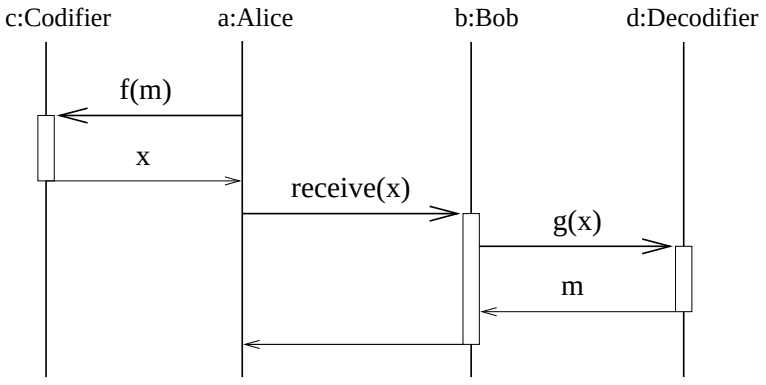

Figure 2: GOOCA Dynamics.

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

Applicability

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

Problem How could the distinction of concerns between application functional objects and cryptography-ware objects be explicitly represented in such a way that reuse and readability can be improved? Putting it another way, 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 functions decoupled from security services facilitates reuse and security policy changes, and frees application programmers from having to acquire cryptographic knowledge.
- The explicit distinction of concerns can lead application designers to: (*i*) procrastination of important security policy decisions in cryptography-ware designs applications; (*ii*) lack of control over cryptographic features, from the application programmers' point of view.
- Delegation of cryptography-ware 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 such a way that non-functional requirements are transparently accomplished by non-functional objects, which may not be known to the application functional objects.

The use of a meta-object protocol explicitly separates cryptographic requirements from application functionalities. Figure 3 approaches GOOCA in a reflective way. The classes

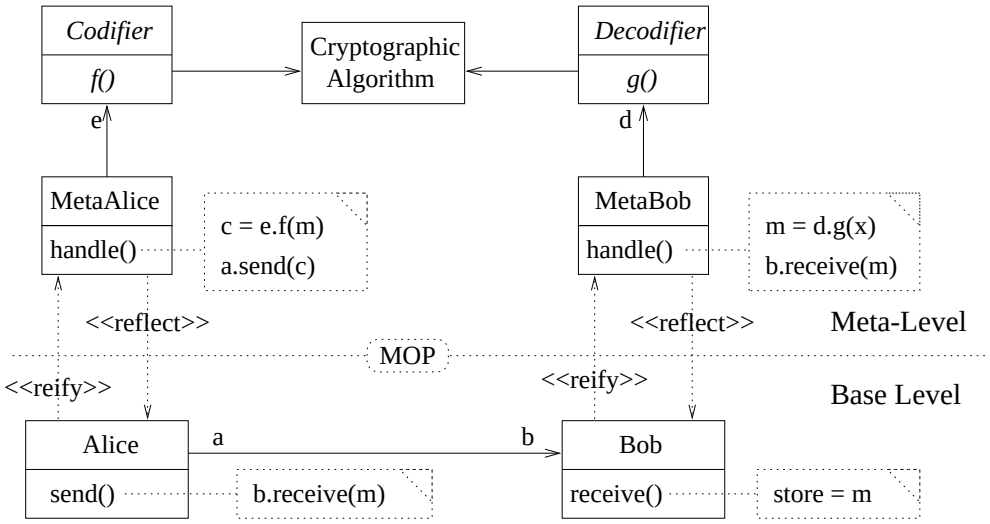

Figure 3: Reflective-GOOCA Structure.

MetaAlice and MetaBob are responsible for cryptographic method calls and for the re-sending of base-level methods, which were previously intercepted. Figure 4 shows the interaction diagram. For instance, the `send()` message 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 message is, then, re-sent (containing now the encrypted argument `c`) by the MOP's kernel to its original target. The same happens with the `receive()` message.

Consequences This design has the following advantages:

- Decoupling of functional objects from non-functional ones in such a way that application objects do not need to know what kind of (cryptographic) transformation is taking place or what kind of security requirements are being accomplished (secrecy, integrity, authentication, non-repudiation, or some valid combination of these).
- Separation of cryptographic objects from application objects in such a way that it is potentially possible to understand application code without cryptographic background.
- Development and testing of cryptographic components can be done only once, separately, and then used in many contexts.

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 the code.
- A time delay due to the message interception mechanism of the meta-object protocol.

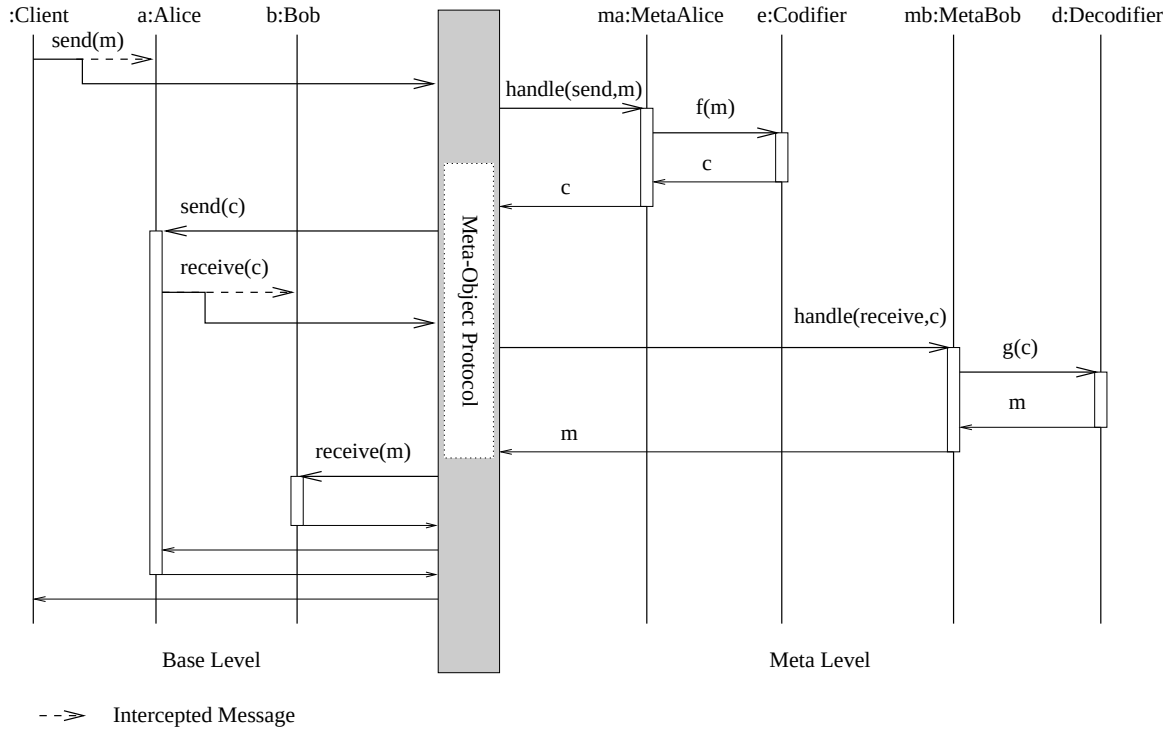

Figure 4: Reflective-GOOCA Dynamics.

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 of messages, 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 to be accomplished. The decision as to which position cryptography will occupy on 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

over 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.

- 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 result interception and modification facilitates transparent secrecy and authentication of operation results.

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

Example Modern software systems are being modeled according to architectural styles [SG96], which consist of groups of components glued together by connectors, according 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 to 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 secrecy, integrity or authentication of data, without COTS modification. In such a situation, a MOP can be used as the architectural connector that glues a cryptographic component to the COTS communication component. The diagrams on Figures 5 and 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 such a case, it requires at least a known (exposed) interface based on (static) class methods, which will be intercepted. For instance, Since Ted works as a class loader and do not have access to instances of AliceAndBob, methods `send()` and `receive()` must be static.

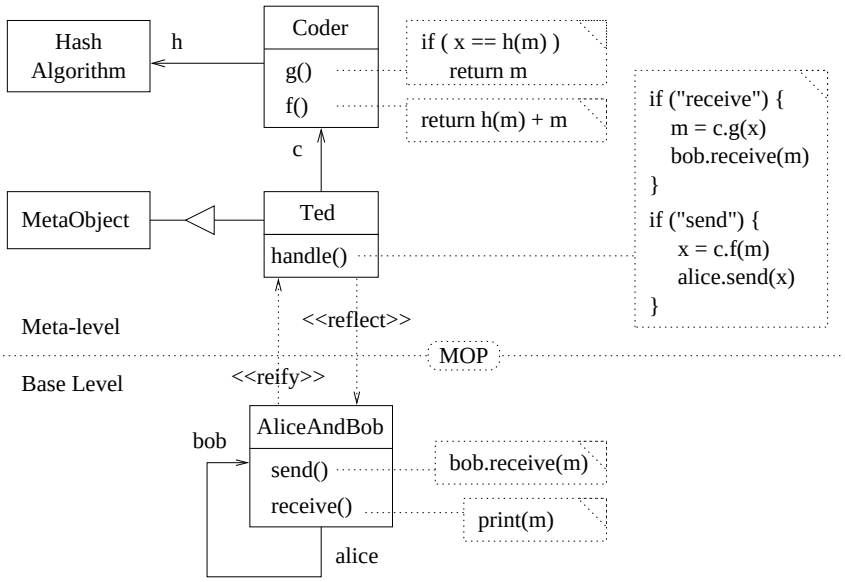

Figure 5: Ted Structure.

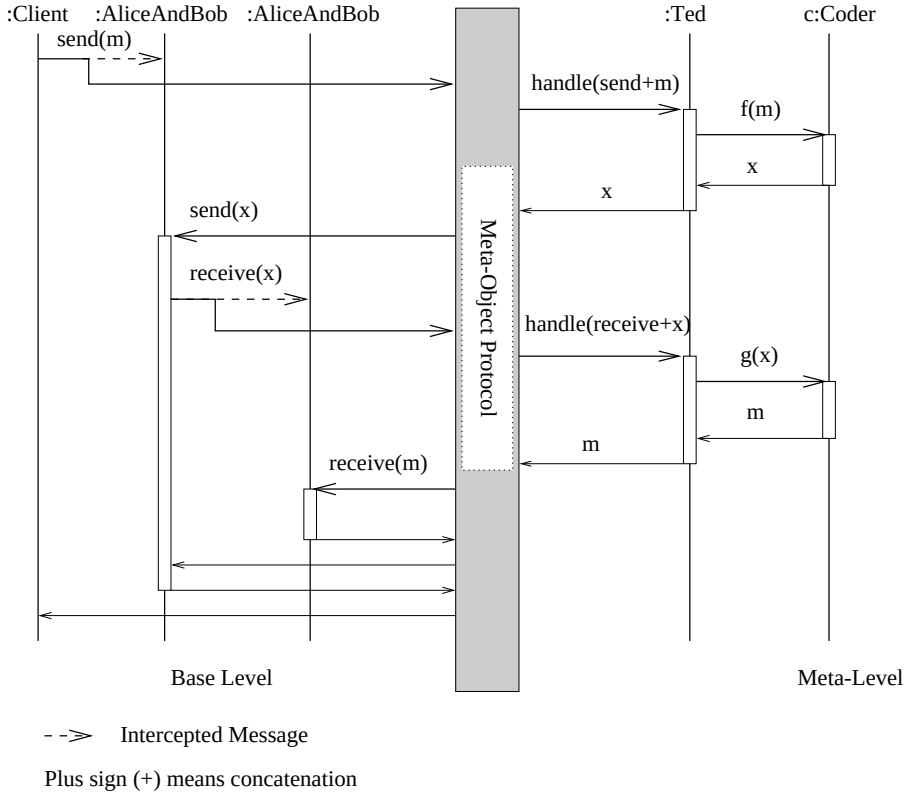

Figure 6: Ted Dynamics.

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 an exposed 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 compiled by the *Guaraná* compiler. On the other hand, AliceAndBob is a common Java ċlass file which is used without any modification.

Classes `UncorruptedObject`, `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, which is computed using 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 `UncorruptedObject`'s creation, in method `encode()`; fingerprint's verification and object recovery are in method `Coder.decode()`.

```
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);
}
```

Class `Ted` 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.

```
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;
}
```

Method `Ted.main()` 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 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.

```
    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 `AliceAndBob` 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.

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

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 architecure, is another interesting application of this pattern [WS98b, Wel97].

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

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-ware 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 such a way that both readability of application code and components reuse are increased. The composability of cryptographic services, such as encryption, integrity, authentication and non-repudiation, is also facilitated by a meta-object protocol in which meta objects could be easily composed, such as in *Guaraná*. The reflective variations of our cryptographic design patterns can be used to document a reflective cryptographic framework for secure object-oriented applications.

4 Acknowledgments

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

References

[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.

[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.

[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.

[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.

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

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

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

[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, Campinas, São Paulo, Brazil, April 1998. http://www.dcc.unicamp.br/ic-tr-ftp/1998/98-14.ps.gz.

[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.

[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.

[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.