Verification of Cryptographic Protocols Implemented in Java Card^{tm*}

Renaud Marlet Daniel Le Métayer
Trusted Logic[†]

June 30, 2003

1 Introduction

Cryptographic protocols are used to ensure secure communications in a hostile environment, when the decisions taken by the principals depend crucially on the authenticity and integrity of messages that they receive. Typically, when a message is received, a principal wants to be sure that this message has been created recently and in good faith for a particular purpose by the principal who claims to have sent it, and that it has not been altered. Such secure communications are indispensable in areas such as Internet, e-commerce and mobile applications.

A large number of cryptographic protocols have been specified and implemented. However, many of them have been shown to be flawed, even long after they were published. This has generated a line of works concerning the formal specification and verification of protocols, e.g., Casper [Low98], Capsl [DMR00], Casrul [JRV00]. Still, all these techniques and tools are based on conceptual views (or models) of protocols and provide no guarantee on their actual implementation. This paper reports on a method and analysis tool whose goal is precisely to bridge the gap between protocol models that can be proven flawless and their implementation.

Our target in this work is Java Card applets: In this case, applications on a card represent principals involved in authentication protocols, with strong security requirements. We have defined a domain-specific language, called DEXTRA, for expressing requirements regarding the implementation of a protocol in Java Card. We have also developed a prototype analyzer to verify the compliance of a Java Card applet with respect to such a specification of a protocol implementation. The verifier relies on static program analysis techniques.

This work is part of the EVA project[1]. The general goal of EVA is to design and develop protocol verification tools that are suitable for industrial needs. The project has formally defined a protocol specification language called LAEVA [LMJ01], that is used as the input of several verification tools: CPV [GL02], Hermes [BLP02], Securify [Cor02]. These tools are based on standard techniques such as abstraction and model checking.

The rest of the document is organized as follows. Section 2 states the general problem and issues of protocol implementation verification. Section 3 presents our particular setting: verifying LAEVA protocols implemented in Java Card. Section 4 describes our approach to address specification and verification issues. Section 5 presents our language for specifying protocol implementations as well as the corresponding verifications. Section 6 sketches our static analysis for protocol verification. Section 7 concludes with future work.

[*] Java Card is a trademark of Sun Microsystems, Inc. in the United States and other countries

[†] Authors: {Renaud.Marlet,Daniel.Le_Metayer}@trusted-logic.fr — Company Web Site: www.trusted-logic.fr

[1] EVA is funded by the French national network for software technologies (RNTL). See http://www-eva.imag.fr

2 Verification of a Protocol Implementation

In theory, verifying that a protocol implementation is flawless could be decomposed in three stages: (1) defining a refinement relation between the implementation and the protocol specification, (2) proving its correctness, and (3) proving the protocol specification flawless. In practice, as explained below, additional hypotheses are required. Besides, the complexity of verifications leads to approximations and possibly to partial results. The analysis of Java Card applets also raises specific issues.

2.1 Relating a Protocol to an Implementation

The implementation of a protocol is usually tightly interleaved with the rest of the application code, with no explicit and obvious connection with data elements mentioned in the protocol specification. A relation between a protocol specification and its use in an implementation has thus to be made explicit before verification can proceed. Establishing such a relation raises a number of issues which are listed below.

Scope of the Protocol within the Implementation. The implementation of a real application may simultaneously rely on several protocols. Consequently, messages considered in the model of a protocol may be interleaved in the implementation with other, unrelated messages. The boundary of the targeted protocol within the implementation thus has to be defined. Besides, the implementation may involve more actors than the principals mentioned in the protocol specification. In this case, those extra actors have to be assumed as trusted.

Message Fragmentation. For style or clarity reasons, protocol specifications may split a single set of information to be communicated into several successive messages. Likewise, an implementation may have to split information into several actual messages, because of message size limitations. For instance data in APDU messages are often limited to 255 bytes. As a result, relating an implementation to a protocol specification requires relating sets of messages rather than just individual messages.

Data Representation and Processing. After data are received and before they are sent, they may be encrypted or decrypted, duplicated, fragmented, stored and later retrieved, transformed, encoded into several actual pieces of data, etc. In fact, most of the data operations in an implementation involving a protocol are generally not related to the protocol but to the actual application processing. These operations have to be abstracted away for a protocol implementation to be related to a protocol specification. Besides, data representation details (e.g., little-endian or big-endian integers) are also left unspecified in protocol definitions.

Error Recovery. Most protocols are expressed as successions of communications and state changes. But the behaviors in the case of unexpected events (e.g., if a signature is incorrect) are generally left unspecified: the procotol just “fails”, there is no error recovery. However, a realistic application does not abort in case of a protocol failure. Whether the failure originates from an intended attack or from unintentional communication errors (including disconnections), the application has to be robust, reliable. Actual implementations thus include recovery mechanisms, to always keep the system in an active and consistent state.

2.2 Hypotheses For Proving a Protocol Implementation Flawless

Another significant issue when turning to a real implementation concerns the fact that proving a protocol flawless generally relies on idealized hypotheses that may not hold in absolute terms. The following assumptions are common in the protocol verification literature:

Perfect Cryptography. Protocol cryptography is generally assumed to be perfect : crypted data cannot be deciphered if the encryption key is unknown. In practice, crypanalysis attacks may reveal data without the key being known, possibly depending on the key size, on the strength of the encryption algorithm, or even on hardware-software protections (e.g., against DPA attacks). In practice, cryptography thus offers only "probabilistic" protection.

Perfect Nonces. Protocol nonces are also generally assumed perfect, i.e., new nonces are not equal to any past nonce. However, in practice, past nonces are not memorized to make sure new ones are indeed different. Challenges often are random numbers and collisions are thus theoretically possible. Strictly speaking, the freshness of nonces is thus also "probabilistic".

Complete Models. Protocol specifications are deemed to model all actors and all knowledge exchange: attackers are assumed to have no initial knowledge (other than specified) and to be able to gain knownledge only by observing communications or forging messages. In practice, this may require organizational measures, hardware protections, etc. At the software level, it is possible to observe all possible communications of an implementation, providing security hypotheses on the underlying operating system.

2.3 The Asnwers of a Verifier

After a relation between a protocol specification and an implementation is stated, compliance can be verified. In pratice, this verification problem is undecidable and require approximations to guarantee termination. Hence, three kinds of answers can be expected to the question " does application I implement protocol P (according to relation R) ? ":

(1) I certainly implements P,

(2) I possibly implements P (or, likewise, I possibly does not implement P),

(3) I certainly does not implement P.

While security evaluation requires "positive" answers, i.e., answers (1) and human-controlled forms of (2), the search for attacks and flaws is more concerned with "negative" answers, i.e., answers (3) and human-discovered errors based on answers (2).

3 Verifying LAEVA protocols implemented in Java Card

In this work, we consider the special case of protocols expressed in LAEVA, and applications implemented in Java Card.

3.1 LAEVA Protocol Specifications

The protocol specification language LAEVA was designed with the following constraints: it should be suitable for e-commerce protocols, it should be close to the (often informal) languages used in the protocol literature, it should be precise, i.e., formally defined and it should enable verifications [LMJ01]. Presenting LAEVA is out of scope for this report; we only provide here a taste of LAEVA through the example of the Diffie Hellman protocol.

```
Diffie_Hellman

// Declarations
A, B :         principal
P, G, Xa, Xb : number

one() :         number
kap(number, number, number) : number hash
       // kap(P, G, X) is meant to be GˆX mod P

// Initial knowldege
A knows A, B, kap, one
B knows B, kap

// Notations
alias Ka = kap(P, G, Xa)
alias Kb = kap(P, G, Xb)
alias Kab = kap(P, Ka, Xb)
alias Kba = kap(P, Kb, Xa)
axiom kap(p,kap(p,g,y),x) = kap(p,kap(p,g,x),y)
  [p, g, x, y : number]
a, b : principal

// Message exchange
{
 1. A -> B : P, G
 2. A -> B : Ka
 3. B -> A : Kb
 4. A -> B : { one() }_(kap(P, Kb, Xa) % kap(P, Ka, Xb))
}

// Session and Property
s. session A=a, B=b
claim Agreement(Kba@s.A, Kab@s.B)
```

Principals and data (called numbers) are explicitly declared. Initial knownledge (using construct `knows`) is declared as well. Then comes a sequence of exchanged messages. The interested reader is refered to [LMJ01] and the EVA web site (http://www-eva.imag.fr) for details.

3.2 Analysing Java Card Applets

In this work, we consider the special case of applications implemented in Java Card[2]. As explained below, several arguments motivate the choice of Java Card to study the verification of protocol imple-

[2] More precisely, we support Java Card 2.1.2 [Sun01]; we do not support yet Java Card 2.2.

mentations. This choice also raises new challenges:

Addressing Java Card Implementations. There are several reasons why smart cards and Java Card are a good target for studying the verification of protocol implementations:

- Applications on a card naturally represent principals. In fact, smart card applications, especially in the banking and identity area, already make use of cryptographic and authentication protocols, with strong security requirements.
- A smart card can be considered as a safe: it comes with strong guarantees that are taken as assumptions by protocols (see Section 2.2).
- Java Card includes all cryptographic features that are needed to implement authentication protocols, as well as a firewall that guarantees applet isolation and thus supports the privacy hypotheses. As a matter of fact, Java Card stands out as a standard for such applications.

Analysing Java Card. The choice of Java Card for applications involving protocols has a number of advantages over other langages regarding static program analysis:

- Generally speaking, Java is simpler to analyze, either at source level compared to C, or at object file level compared to machine code. In particular, it is well typed and it does not involve pointer arithmetic.
- Besides, unsupported complex program features make the analysis of Java Card easier: compared to Java, there is no dynamic loading, no threads, and the standard API is small.
- Because the cryptographic API is well-defined, it is easier to relate the operations of a protocol with a corresponding implementation.

Verification Challenges. However, Java Card program analysis also brings new challenges:

- Memory can be dynamically allocated. Data structures are generally allocated and their size fixed at installation and/or personalization. While dynamic memory allocation is not a new problem, the required precision to make meaningful verification is a new challenge.
- Communication with a Java Card applet is low-level: a message consists of a byte array[3]. This makes program analysis much more difficult because each array element has to be precisely modeled.
- Besides, unconventional features have to be addressed, such as persistency, transient data, reset, firewall rules, etc.

4 A Pragmatic Approach

Rather than trying to solve the problem in its most general form, we follow a pragmatic approach:

Specification of a Protocol Implementation. First, we merge the protocol specification and the relationship between the protocol and the implementation (see 2.1) into a single specification. More precisely, our analyzer takes two pieces of information as input: a Java Card applet implementation I

[3] As of version 2.2, a terminal may also perform remote method invocations (RMI) in the Java Card applet. Communicated data types in this case are thus higher level, which facilitates program analysis.

and the specification of a protocol implementation Q, written in a dedicated language called DEXTRA. In such a specification, as opposed to LAEVA specifications, messages are given a low-level representation based on byte formats, to be easily mapped into Java Card applet messages. This specification allows data fragments to be named, including random data (for nonces) as well as encrypted or signed data.

Moreover, an actual protocol specification P, written in LAEVA, can be automatically extracted from Q, and proved flawless using the other tools developed in the EVA project. In other words, the verification of a protocol specification is performed on P, while the implementation verification is performed on Q. The extraction of the protocol specification P is straightforward, which makes the approch more robust.

Detection of Wrong Implementations by Static Program Analysis. Second, as a preliminary step, we focus on negative answers (see 2.3): verification fails when it can be shown that the implementation does not comply to the protocol specification.

Our approach to verifying the specification of a protocol implementation Q against an implementation I is based on static program analysis. The application I is executed on abstract domains, taking in turn each expected input message patterns, as specified in Q, and checking the correctness with respect to the corresponding expected output message.

5 Checking Implementations against DEXTRA Specifications

This report does not present a formal definition of DEXTRA. It only provides, through examples[4], an overview of the specification language and associated verification.

5.1 Basic Information Communication

A DEXTRA specification consists of information declarations, followed by message exchanges between the CAD and the applet. As explained above, all messages are APDUs, i.e., byte arrays. The following example illustrates a simple DEXTRA specification.

```
Number myInfo1, myInfo2;
CAD -> Applet : [8*myInfo1, 16*myInfo2];
Applet -> CAD : [16*myInfo2];
```

Two data involved in the protocol are first declared: `myInfo1` and `myInfo2`. The protocol then states that the applet first receives a message consisting of 8 bytes from `myInfo1`, followed by 16 bytes from `myInfo2`. The applet then has to provide as a response a 16 bytes message taken from `myInfo2`. The following Java Card code fragment is compliant with the above protocol.

```
byte[] apduBuffer = apdu.getBuffer();
byte[] info1 = new byte[8];
byte[] info2 = new byte[16];
for (short i = 0; i < 8; i++)
   info1[i] = apduBuffer[i];
Util.arrayCopy(apduBuffer, 8, 16, info2, 0);
```

[4] Some of these examples are extracted and adapted from the specification of a protocol implementation concerning an electronic purse.

```
Util.arrayCopy(info2, 0, 16, apduBuffer, 42);
apdu.setOutgoingAndSend(42, 16);
```

Note that information transfer can be performed explicitly on byte array elements or using API method `Util.arrayCopy`. Array indices are verified with precision to make sure all specified bytes are correctly read (on reception) or written (on sending

5.2 Cryptographic Information Communication

The following protocol fragment illustrates the communication of encrypted information.

```
SecretKey myKey;
Number myInfo;
...
Applet -> CAD : [..., 8*{8*myInfo}_(16*myKey), ...];
```

This protocol fragment specifies that the applet has to return a message consisting of 8 bytes originating from the encryption of `myInfo` by `myKey`, that is a 16-byte key. The following Java Card code excerpt is compliant with the above protocol fragment.

```
Key key = (DESKey) KeyBuilder.buildKey(
   KeyBuilder.TYPE_DES,
   KeyBuilder.LENGTH_DES3_2KEY, false);
key.setKey(buffer, offset);

Cipher algDES_ECB_noPad = Cipher.getInstance(
   Cipher.ALG_DES_ECB_NOPAD, false);

algDES_ECB_noPad.init(key, Cipher.MODE_ENCRYPT);
algDES_ECB_noPad.doFinal(infoBuf, off, len, apduBuf, off2);
```

The above Java Card code shows the construction of both the key and encipherment algorithm, as well as the information encryption and copy into the APDU buffer for message emission.

Similarly, information decryption occurs at message reception. Signed information may also be expressed in DEXTRA specification, using syntax `myInfo%myKey`.

5.3 Nonces and Random Numbers

The following protocol fragment illustrates the communication of nonces.

```
Random  N1, N2;
...
Applet -> CAD : [..., 8*N2, ...];
```

This protocol fragment specifies that the applet has to return 8 bytes originating from a random number. The following Java Card code excerpt is compliant with the above protocol fragment.

```
RandomData rnd = RandomData.getInstance(
   RandomData.ALG_SECURE_RANDOM);
rnd.generateData(sessionData, offset1, len); // N1
rnd.generateData(sessionData, offset2, 8); // N2
```

Contrary to the declaration of numbers and secret keys, the order of the nonce names matters: each random number successively generated is associated with the successive nonce names in the declaration.

5.4 Determination and Undetermination in Messages

As illustrated by the above examples, a DEXTRA specification expresses what messages are received. This is mainly a binding requirement: received data are given names as indicated by message components. A specification also expresses what messages are sent. This is a verification requirement: sent data have to include the named data as specified.

Message formats may also express determined, undetermined and arbritrary values, as illutrated by the the following example.

```
CAD -> Applet [42, num, 8*hostChallenge, ^, 6*^];
```

This message pattern splits the byte array into groups of contiguous bytes and expresses the following constraints regarding the received data. The first byte is given value 42; this is mainly used to identify commands and options involved in a protocol. The second byte is bound to name `num` and can be reused in the rest of the protocol messages. The following segment is an 8-byte segment bound to name `hostChallenge`. Symbol ^ indicates a 1-byte undetermined value while `6*` expresses a 6-byte undetermined segment.

The above message is a received message. It corresponds to value assignement in the implementation. Messages that are sent have a different interpretation: they result in verifications. In other words, the implementation has to return response messages as indicated. This is illustrated by the following example.

```
Applet -> CAD [0xB3, ^, _, 8*cardChallenge];
```

For verification to succeed, the sent message has to be 11 bytes long. The first byte in the sent data must be `0xB3`. The second byte (^) is undetermined: it must *not* be any named information; for instance, it is an error if it is a byte from `cardChallenge`. On the contrary, the third byte (_) is arbitrary: it is unchecked; in particular, it could be a byte from `cardChallenge`. And the last 8 bytes have to be 8 bytes originating from `cardChallenge`.

5.5 Initial Knowledge

There is a slight difference between LAEVA and DEXTRA concerning the initial knowledge of principals. In LAEVA, initial knowledge is simply declared. In DEXTRA, it has to be indirectly communicated through installation and personalization commands.

6 Checking Wrong Implementations by Static Program Analysis

As said earlier, we focus on negative answers (see 2.3): the implementation is analyzed to verify that it cannot send messages other than those defined in the specification of the protocol implementation. This section outlines the static analysis that supports this verification.

Verification is performed by executing the application on abstract domains, taking in turn as input each message pattern sent by the CAD. The abstract domains used for this execution manipulate symbolic information. Values can be known constants (e.g., 42), fragments of symbolic information (e.g., `hostChallenge`) or undetermined values (ˆ). The processing of named pieces of data by the applet includes signature, decryption and encryption (using named keys), as well as random numbers. In this execution, dynamic memory allocation is supported.

The analyzer follows all execution paths from the call of an entry point method (e.g., `process`) until either it returns or throws an exception. Infinite paths originating from loops and recursion are folded into (approximated) finite paths. Computations whose value is static also help trim the execution tree. To cover message sequencing with high precision, the heap state is maintained across all successive calls to entry point methods. The state graph of the applet can thus be fully explored.

For each termination point of the different, possibly approximated, execution path, the output response is checked. Two cases may arise:

- If an exception is raised, it is not considered as a failure from the point of view of verification: the application may fail for reasons that are independent of the protocol.
- If a message is sent, it is matched against the corresponding expected message in the specification of the protocol implementation, as indicated above. Verification fails if any discrepancy is found: the implementation certainly does not implement the given protocol (see Section 2.3). For instance, the verifier detects any emission of a piece of information that is not explicitly mentioned in the corresponding expected response messages.

Installation and personalization are modeled by corresponding abstract calls to methods `install` and `process`.

Our analyzer actually works on Java Card object files (i.e., CAP files) rather than source code. Therefore, we do not have to assume the correctness of compilation, optimization, or conversion; the verification is performed on the final form of the application, before it is loaded on a card.

7 Future Work

We believe that the verification of actual implementations of cryptographic protocols (as opposed to models of such protocols) has not received enough attention from the community in the past. We have described in this paper the first results towards bridging this gap in the verification chain. This work paves the way for the development of new tools which could benefit the security industry and could be used in the certification of critical pieces of software such as cryptographic protocols.

A number of possible extensions to this work are currently underway, relating to both DEXTRA and the verification tools.

For the moment, a specification in DEXTRA can only address a subset of the features existing in LAEVA. The tool will be extended to address features such as hashing, axioms and multiple sessions.

Currently, our verifier does not check if all bytes of a given information are actually sent. It verifies that at least a fraction of this information is sent, or that no fraction at all is sent. This catches most error cases and agrees with the verification objective, which is to prove compliance with model rather than functional correctness.

For the moment, we only consider applets that communicate via APDU messages, thus excluding communications via shareable interfaces. We relate protocol messages to Java Card command and response APDU messages, i.e., byte arrays. If shareable interfaces are actually used in the implementation, other communicating agents on the card are assumed to be safe. In other words, we check only one principal, represented by an applet instance. This principal is assumed to encompass the applet as well as all actors on the card that may, directly or indirectly, communicate with the applet. Alternatively, this assumption can be relaxed by just making sure that no information involved in the protocol can be communicated to other actors on the card.

References

[BLP02] Liana Bozga, Yassine Lakhnech, and Michaël Périn. L'outil de vérification HERMES. Technical Report EVA-6-v1, Verimag, Grenoble, France, May 2002. Available from http:://www-eva.imag.fr.

[Cor02] Véronique Cortier. L'outil de vérification SECURIFY. Technical Report EVA-7-v1, Laboratoire Spécification Vérification (LSV), Cachan, France, May 2002. Available from http:://www-eva.imag.fr.

[DMR00] Grit Denker, Jonathan Millen, and Harald Ruess. The Capsl integrated protocol environment. Technical report, SRI International, 2000.

[GL02] Jean Goubault-Larrecq. Outils CPV et CPV2. Technical Report EVA-8-v1, Laboratoire Spécification Vérification (LSV), Cachan, France, May 2002. Available from http:://www-eva.imag.fr.

[JRV00] Florent Jacquemard, Micahel Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In Springer Verlag, editor, *7th International Conference on Logic for Programming and Automated Reasoning*, Lecture Notes in Artificial Intelligence, November 2000.

[LMJ01] Daniel Le Metayer and Florent Jacquemard. Langage de spécification de protocole cryptographiques de EVA : syntaxe concrète. Technical Report EVA-1-v3.17, Trusted Logic, November 2001. Available from http:://www-eva.imag.fr.

[Low98] Gavin Lowe. Casper: a compiler for the analysis of security. Technical report, University of Leicester, UK, July 1998. Available at http://www.mcs.le.ac.uk/˜glowe/Security/Casper/index.html.

[Sun01] Sun Microsystems. Java Card 2.1.2 Development Kit, 5 April 2001. See http://java.sun.com/products/javacard/dev_kit.html.