Security Properties and Java Card Specificities To Be Studied in the SecSafe Project

Date: 2001/08/23 08:40:24 (UTC)
Authors: Renaud Marlet, Daniel Le Métayer (Trusted Logic)
Classification: Public
Number: SECSAFE-TL-006
Version: 1.2
Status: Stable

Abstract. This report describes security properties to be studied in the SecSafe project. These properties are to be checked by static analyses. The current version of the document only contains a first set of properties; this set will be refined and expanded in the course of the project.

The document also highlights characteristics of Java Card applets that make some analyses particular, even though the corresponding property can be "classic". These specificities originate from the Java Card language and runtime environment as well as from the nature and programming style of typical applications.

Contents

1 Introduction

This report describes security properties to be studied in the SecSafe project. The set of properties described here is not definitive nor exhaustive; it will be refined and expanded in the course of the project.

Security properties of interest for the SecSafe project have been determined based on Trusted Logic's experience in security issues and applet development, using the following criteria.

- The security issues that are addressed by the properties are relevant for the typical Java Card domains (banking, GSM, etc.).
- The properties are either original or involve specific treatments due to the Java Card language and runtime environment as well as the nature and programming style of applications in the domain. This contributes to the novelty of the research carried out in the project.
- The properties are challenging: there is no obvious (good) solution. Analyses that efficiently and accurately check these properties are not available off-the-shelves.
- We focus for the moment on properties concerning a single applet, only communicating using APDUs. This corresponds to the present state of the market of Java Card applets. Properties concerning applets communicating via shareable interface objects will be studied later.

The document is organized as follows. Section 2 recalls notable specificities of the Java Card language as well as Java Card applications. This is important as some standard properties become harder in the context of Java Card. Section 3 defines and illustrates properties of interest, targeted at security. Section 4 concludes on the precision, usability and assessment of analyses that are to check these properties.

2 Java Card Specificities

Before describing security properties, we recall in this section what makes Java Card applications original, compared to other IT applications. This specificity not only stems from the programming language and runtime environment but also from the scarce resources (little memory), security concerns, application nature, programming style, etc.

2.1 Applet Lifetime

A Java Card applet goes into different logical stages.

- *Loading:* the code of the application is downloaded onto the card.
- *Installation:* an applet instance is created and registered. Several instances of the same applet may coexist.
- *Personalization:* a registered applet instance receives personalization data and initializes. An applet is personalized only once, and only then becomes operational. Note however that there is no explicit support in the system concerning personalization. Although it is a standard logical phase in the life of an applet, it has to be implemented explicitly by the programmer. In particular, there is nothing in the system to prevent re-personalization or use of an un-personalized (hence not operational) applet; it is up to the programmer to set up an explicit flag.
- *Processing:* a registered, operational applet is selected, receives several commands (and performs them), and is deselected. This can be repeated.

In addition, at any point during processing, the power can be *reset* and the processing stage stops. However, new processing phases can be performed afterwards when power is up again. As loading, installation and personalization are performed in a secure environment, it is enough to consider that reset can only happen during processing.

Packages and applet instances can be deleted. If not, their lifetime is that of the JCVM running on the card. (In some circumstances, the JCVM may stop forever.)

All properties must be true for the whole applet lifetime, or part of it (e.g., after personalization). An analysis verifying these properties must thus take into account all the above stages.

2.2 Entry Points, Reset and Default Applet Selection

Contrary to ordinary programs that have a single "main()" entry point, Java Card applets have several entry points, that are called when the card receives various APDU commands. These entry points roughly match the above lifetime stages. They correspond to the following methods (see class javacard.framework.Applet and interface org.globalplatform.Application):

- *install()* is called to instantiate and register a new applet. The system guarantees that any selectable applet has been created by the JCRE calling the install() method. Note that install() is a static (class) method, as opposed to the following instance methods.
- *processData()* is called to provide personalization data to a registered applet (OP 2.1 implementations only, see command STORE DATA). It is usually called once, sometimes a few times, and only for personalization. However, this calling pattern is not guaranteed by the system: processData() can be called at any time, any number of times.
- *select()* is called on an unselected applet, to select it. The system guarantees that the currently selected applet is deselected — method deselect() is then invoked — before a new applet selection.
- *process()* is called on a selected applet (including when a SELECT APDU is received, but after the invocation of the select() method). This command is typically called several times before the applet is deselected.
- *deselect()* is called on a selected applet, to deselect it. The system guarantees that the currently selected applet is deselected before a new applet selection.

In addition, as noted in the preceding section, a *reset* may occur at any time, interrupting and terminating the current computation. In general, after a reset, no applet is considered selected. In some cases though, an applet may be selected by default. In this case, its select() method is called but the process() method is not called as there is no SELECT APDU to treat.

All the above methods are called by the JCRE when receiving the corresponding APDU commands. However, note that nothing prevents an applet from explicitly calling the above methods. It will have no impact on the internal state though. In particular, calling install() and attempting to register an applet instance will fail. Likewise, calling select() or deselect() will have no impact on the currently selected applet. Note that public methods of an applet can be also called by another applet via a shareable interface object or via the JCRE calling the method getShareableInterfaceObject().

2.3 Persistence

There are five persistence / volatility levels for data areas on a card:

- *Persistent class data areas* correspond to static fields. They are made available when a class is loaded onto the card. They live until the package of the class is possibly deleted from the card.

- *Persistent instance data areas* correspond to instance fields (of the applet instance as well of objects allocated by the applet) as well as arrays. They are made available when an applet instance is installed, possibly creating other objects and arrays. They live until the applet is possibly deleted from the card.
- *Card session data areas* correspond to CLEAR_ON_RESET transient arrays. They are made available only during a card session, i.e., between a power up and a power down of the card. On a power up, such arrays contain a null or zero value (depending on their type).
- *Applet session data areas* correspond to CLEAR_ON_DESELECT transient arrays. They are made available only during an applet session, i.e., between a selection and a deselection of an applet instance. On a selection, such arrays contain a null or zero value (depending on their type); they are cleared on a deselection.
- *Method invocation data areas* correspond to local variables (present on the method invocation frame). They are available only as long as a method is being executed[1].

In the event of a reset (which can occur at any time), only persistent data areas are preserved (see section “Atomicity” just below though); data in other areas are lost.

2.4 Atomicity

Writing persistent data can be performed atomically, i.e., without being interrupted “half way through” by a reset. Atomicity is based on a notion of *transaction*, as made available by the following API methods (see class javacard.framework.JCSystem):

- *beginTransaction()* starts an atomic update of persistent data.
- *commitTransaction()* performs and terminates an atomic update. If a reset occurs before a transaction is committed, the transaction is aborted (see below).
- *abortTransaction()* aborts an atomic update, i.e., all writings into persistent data areas are undone to recover the state of persistent memory just prior to the previous call to beginTransaction().

Note that non-persistent data areas (i.e., transient array and local variables) are unaffected by transactions: changes are never undone.

Some API methods (see class javacard.framework.Util) also perform atomic array updates, as if in a transaction (but can be used whether or not a transaction is currently active):

- *arrayCopy()* atomically copies an array segment into another one.
- *setShort()* atomically copies two bytes in an array.

Note that, conversely, some methods perform non-atomic writings, *even if a transaction has already been started*:

- *arrayCopyNonAtomic()* non-atomically copies an array segment into another one.
- *arrayFillNonAtomic()* non-atomically fills an array segment with a given value.

2.5 Memory Management

In Java Card, memory allocation is mostly dynamic: all data areas are dynamically allocated, except static fields. However, there is no garbage collection (as of Java Card 2.1) and memory cannot be

[1]Depending on implementations, these data areas may still contain information in memory although it is unreachable by the application. This memory can be observed via hardware means though (see also §3.7).

explicitly freed — in a context where memory is a rare and expensive resource. Consequently, as memory is a scarce resource, most applications allocate dynamic data areas at installation and personalization time, and never allocated any data area later on. Objects and arrays are thus generally allocated only during the execution of methods install() and processData(), as well as in process() in the part of the code that corresponds to personalization and that can be executed only once (see §2.1).

As a result, the data structure of an applet is usually fixed, as if immutable. Still, a few instance fields of reference type are sometimes used as caches (rather than as pointers to substructures): their value may vary during processing.

2.6 Firewall: Object Creation and Access

The Java Card firewall takes into account the following aspects:

- the currently selected applet,
- object ownership (and transience status),
- context switching (when invoking a method),
- access across contexts (via global arrays and JCRE permanent/temporary entry point objects,
- shareable interface objects.

Any attempt to go through the firewall raises a SecurityException. An analysis must thus model the above features of Java Card to address SecurityException issues.

2.7 Use of Arrays

One of the characteristics of a Java Card applet, compared to other IT applications, is the heavy use of arrays. Arrays are the main communication medium, starting with the APDU buffer used as I/O for APDU command processing as well as for data communication via a shareable interface object.

Arrays are often heterogeneous, i.e., various data (sometimes unrelated) are grouped into the same array. It is the case in particular for the APDU buffer that groups input (resp. output) data. Different “logical” arrays are also sometimes grouped into a single actual array in the implementation to reduce the number of objects and hence to reduce the memory footprint. A security analysis must thus be able to tell apart different array elements, i.e., array accesses at different indices. Otherwise, all information in the array has to be considered the same.

Besides, the same array is often used with different contents during the execution of methods. This is the case in particular of the APDU buffer, that is first used to read input data (possibly in several successive chunks if the data are too large), then used as a temporary array during processing, and last used to write output data (possibly in several successive chunks too). A security analysis must thus be program-point sensitive, at least regarding some important arrays.

2.8 Silent Overflow

In Java Card, arithmetic operations silently overflow: additions, subtractions, multiplications and negations are performed with an implicit modulo which corresponds to the size of the (signed) integer type. This is not a specificity of Java Card but it is something to take into account when considering the security of Java Card applets. In many cases, existing applets do not check possible overflows when performing arithmetic operations.

2.9 Control Flow

The control flow of Java Card applets is that of programs in object-oriented languages, with the following specificities.

- *Virtual method invocation.* There is very little use of inheritance. (Only a few classes are typically defined anyway.) Besides, there usually is no method overriding, except for the implementation of abstract methods or “dummy” methods (such as the default methods of class javacard.framework.Applet).
- *Interface method invocation.* Usually, only a few classes implement an interface.
- *Exceptions.* Java Card applet programmers generally do not use exceptions as an algorithmic programming means (as opposed to just a way to signal errors). Still, the whole possible control flow has to be taken into account to yield safe analysis results, including the case where an exception is caught before being rethrown or before another exception is thrown.

3 Security Properties

This section provide a preliminary list of security properties to be studied in the SecSafe project, in the light of the Java Card specificities mentioned in section 2. This list is preliminary for two reasons: (1) we are still waiting for possible feed back from the other partners, (2) the set of properties will grow and adapt as the project develops.

Many of the examples used in the following to illustrate properties are extracted from or inspired by the Demoney application [MM01], that is a demonstrative electronic purse.

For readability reasons, all examples provided in this document are in Java Card, as opposed to the JCVM language. However, the translation to JCVML as well as to Carmel (the JCVML dialect studied in the SecSafe project) [Mar01] is straightforward.

3.1 Memory Allocation Control

It is very important for a Java Card applet to control the use of the (meager) memory resources (see §2.5). The basic memory allocation control reads as follows.

> **Bounded Memory Allocation:** *The dynamic memory allocated by an applet instance must be bounded.*

Checking this property requires exploring all the control flow of the applet, including the different patterns of calls to the applet entry points. In particular, a graph of all reachable states (memory configuration patterns) could be constructed to determine “points of no return” after which no memory allocation can be performed.

As applets generally allocate dynamic data areas at installation and personalization time (see §2.1), and never allocate any data area later on (see §2.5), another aspect of memory allocation control is:

> **No Memory Allocation after Personalization:** *Memory allocations must be performed during installation and/or personalization only.*

As there is no way to implicitly know when personalization is finished (see §2.1), this second property has to be parameterized by an explicit characterization of personalization. There are two ways to do so.

- Personalization can be characterized by the format of an APDU command, or a set of formats if personalization requires several commands. The last personalization command must mark a point where the applet cannot be repersonalized.
- Personalization can also be characterized by a condition expressed on the state of the applet (typically on an instance field acting as a personalization flag). The definitive truth value of this condition indicates that personalization is finished and that the applet cannot be repersonalized.

In both cases, "the applet cannot be repersonalized" means that if a new personalization command is received, then an error is raised without altering the state of the applet.

Here is an example:

```
DESKey adminKey;
OwnerPIN pin;

install(...) {
  // Allocation during installation
  adminKey = (DESKey) KeyBuilder.getInstance(...);
  ...
}
process(...) {
  switch (...) {
  case PERSONALIZE:
    if (pin != null)   // Already personalized?
      ISOException.throwIt(SW_CONDITIONS_NOT_SATISFIED);
    // Allocation during personalization
    pin = new OnwerPIN(maxPINtry, maxPINsize);
    break;
  case DEBIT:
    if (pin == null)   // Not personalized yet?
      ISOException.throwIt(SW_CONDITIONS_NOT_SATISFIED);
    ...
}
```

In this example, the applet is considered personalized (and thus operational) when the field `pin` becomes different from null. At this time, any attempt to repersonalize the applet (which would reallocate memory) raises an error.

3.2 Information Flow Control

One of the primary aspects of security is the preservation of secrets. Information stored in the a Java Card applet typically has to be typed with secrecy levels. Information with given secrecy levels are required not to be disclosed.

> **Information Privacy:** *Given types of information must not flow outside of the package or the API/JCRE.*

The different types of the information present in the applet has to be provided as a parameter to this property. There are two ways to do so.

1. A map of the applet data areas is specified and annotated with secrecy types.

2. Data provided to the applet during personalization is specified and annotated with secrecy types.

Both kinds of specifications have a practical interest. As a matter of fact, we actually have to consider two properties, depending on the way security information is specified.

Note that, in either case, the secrecy type of external methods, including API methods, must also be provided, including their side effects and the flow from the method arguments to the result of the invocation.

We are only interest here in *direct* leakage of information[2], i.e., secret information sent for output, provided as arguments to insecure external methods, provided as return value, etc. This property is illustrated by the following example.

```
DESKey adminKey;  // Secret

process(...) {
  ...
  // Store key at offsets 12–27 of APDU buffer (length of 2 key triple DES is 16 bytes)
  adminKey.getKey(adpuBuffer, (short)12);
  ...
  // Output info of APDU buffer at offsets 0–4
  apdu.sendBytes((short)0, (short)5);   // OK, not sending any secret
  // Output info of APDU buffer at offsets 10–17
  apdu.sendBytes((short)10, (short)8);  // BAD, sending part of secret
  ...
  sio.meth(apduBuffer,...);             // BAD, giving secret as argument
  ...
}
```

Analyzing this property requires in particular taking into account features such as the applet lifetime (§2.1) and entry points (§2.2), atomicity (§2.4), and the use of arrays (§2.7). The presence of the firewall (§2.6) is also to be considered to reduce the number of false positives as it dynamically prevents potential leaks: objects containing secret information can safely flow outside of the applet because attempts to access their secret fields fails and raises a SecurityException.

3.3 Service Control

Some services offered by an applet must be available only to a restricted, authenticated group of clients.

> **Conditional Execution Points:** *Given program points must be executable only if given conditions are satisfied.*

Note that this property has two parameters:

- a set of program points, to be defined on the applet bytecode (possibly via the program source),
- a set of corresponding conditions concerning values stored in the data areas of the applet.

[2] *Indirect* leakage of information corresponds to secret information that can be deduced analyzing the answers of the applet to various inputs (including state changes), observing the time required to provide an answer, etc. Indirect leakage can occur when there are conditions in the program that depend on secret information. We do not consider this kind of leakage here, nor do we consider *hardware* leakage such as the observation of the card memory — see §3.7 though.

Here is an example based on Demoney where a program point (where the balance is updated) must be executable only when a condition is satisfied (the security level must at least be “credit” and the PIN must be validated).

```
case CMD_CREDIT:
  if (securLevel < CREDIT)
    ISOException.throwIt(SW_SECURITY_STATUS_NOT_SATISFIED);
  else if (pin.isValidated())
  {
    ...
    // Program point of interest — OK, conditions to reach this point are satisfied
    balance += amount;
    ...
  }
  else
    ...
```

3.4 Error Prediction

Security often implies safety. In Java, static typing and (implicit) dynamic checking prevents most safety problems. Still, exceptions can be thrown and reach the toplevel.

In the case of Java Card, if any exception attains the toplevel, i.e., goes up past the applet entry point that was invoked by the JCRE, then the current command is aborted and the response message (possibly not completed yet) is terminated by a status word as follows: if the exception is an ISOException, the status word is given by the reason code of the exception; otherwise the reason code is 0x6f00, meaning “no precise diagnosis”.

Throwing an ISOException is the only way to provide a status word in a response message. As status words fully take part in applet specifications, throwing an ISOException is thus perfectly normal; it generally signals a misuse of the application. On the other hand, throwing any other exception generally denotes an abnormal behavior: something wrong happened and the applet invocation aborts without any specific explanation. (However a specific exception can perfectly be caught and possibly turned into an ISOException; what we consider here are exceptions that reach the toplevel.)

As a matter of fact, whereas an ISOException has to be explicitly thrown by the applet (i.e., using instruction `throw`), most other exceptions are implicit raised as executing an API method call or an instruction when an error occurs, e.g., a null pointer exception. (All exceptions can be explicitly thrown by the application though.)

We thus consider that an exception different from an ISOException reaching the toplevel is likely to denote something unexpected, hence possibly dangerous for the security. For instance, the applet might be left in an unpredicted and ill state.

In practice, exceptions other than ISOExceptions reaching the toplevel should only be considered as warnings rather than sure errors. In some cases, it is indeed difficult for an applet to provide a precise diagnosis and 0x6f00 has to be returned anyway. This can be the case when an applet uses a library that throws runtime exceptions. It makes sense then to deliberately rely on uncaught, implicitly thrown exceptions and not to impose catching such exceptions (to possibly turn them into ISOExceptions). Leaving exceptions flow up to the toplevel has a non-functional impact: as applet have to be as small as possible, the try-catch code can be saved. The following property is thus parameterized to possibly exclude some exceptions different from ISOException, that are accepted nonetheless at the toplevel.

Only ISOException at Toplevel: *No exception other than an ISOException should be thrown as a result of invoking an applet entry point, except for given exceptions thrown at given program points.*

Although it can be expressed on a single line, this property covers very different problems. An analysis must check conditions for a reference to be null (NullPointerException) or to have a wrong type (CastException, ArrayStoreException), for an integer value to be out of range (IndexOutOfBoundsException, NegativeArraySizeException), for an object not to be accessed in the proper context (SecurityException), etc. For the sake of the study, the property will probably have to be decomposed into different subproperties, where a single type of exception different from ISOException is prevented to reach the toplevel:

No X Exceptions at Toplevel: *No exceptions typed X should be thrown as a result of invoking an applet entry point, except if thrown at given program points.*

Moreover, the JCRE automatically raises a TransactionException (and thus returns a 0x6f00 status word) if an invoked entry point method returns without closing an open transaction. To take this behavior into account, a variant of the above property, specialized for the TransactionException, reads as follows. (See also §2.4.)

Well-formed Transactions: *For all execution paths:*

- *Do not start a transaction without having committed or aborted the previous one.*
- *Do not commit or abort a transaction without having started any.*
- *Do not let the JCRE close an open transaction.*

Note that the property does not mention the applet toplevel: this property says that a TransactionException must not be thrown *at all*, i.e., at any program point. Note also that we have excluded here the case where a TransactionException is thrown before the commit buffer is full as this is implementation dependent.

Here is an example.

```
process(...)
{
  if (...)
    abortTransaction();    // BAD, no transaction has started yet
  ...
  beginTransaction();      // OK, this is a new transaction
  if (...)
    beginTransaction();    // BAD, a transaction has already started
  else
    return;                // BAD, the current transaction was not terminated
  if(...)
    commitTransaction();   // OK, the started transaction is committed
  else
    abortTransaction();    // OK, the started transaction is aborted
}
```

The case where a transaction has *possibly* been started is trickier. For instance:

```
logRecord(...)
{
  // Remember if a transaction is started at entry point
  boolean inTrans = JCSystem.getTransactionDepth() > 0;

  if (!inTrans)
    beginTransaction();   // OK, new transaction started only if none already

  currRec = (currRec+1) % logRec.length;
  logRec[currRec].amount = amount;
  arrayCopy(buffer,DATE_OFFSET,DATE_LEN,logRec[currRec].date);

  if (!inTrans)
    commitTransaction(); // OK, there always is a started transaction

  // Transaction depth on return is the same as when entering the method
}
```

3.5 Atomic Updates

It is fundamental for Java Card applets not to go into an unsound state. Because a reset can happen at any time, applets must make sure that all their state updates are sound and atomic (see §2.4). This gives rise to the following property.

> **Atomic Updates:** *Given sets of objects fields and permanent array elements that are considered as related must be updated atomically, i.e., in the same transaction or using an atomic API method call.*

This property has got a parameter: sets of related persistent data areas that are to be updated simultaneously. These persistent data areas are static fields, instance fields and elements of non-transient arrays.

Here is an example based on Demoney where a balance and a log record (not detailed here) have to be kept in phase; however, the maximum balance can be updated independently. (The balance, the log record and the maximum balance are kept in persistent memory.)

```
beginTransaction();
balance += amount;
updateLogRecord(balance, terminalId, date);
commitTransaction();   // OK, balance and log record updated atomically

beginTransaction();
balance += amount;
commitTransaction();   // BAD, the balance is updated, not the log record

beginTransaction();
balance += amount;
updateLogRecord(balance, terminalId, date);
localVariable++;         // OK, local variables do not take part in the persistent state
```

```
transientArray[0]++;   // OK, transient arrays do not take part in the persistent state
maxBalance = newMax;   // BAD, transaction also used for other persistent data areas
commitTransaction();
```

3.6 Overflow Control

The specification of applets often mentions arithmetic operations on integers. Implementations must take care of the limited precision of primitive integers and on possible overflows (see §2.8). In practice, it is sometimes convenient to let some integer variables overflow, e.g., in the case of counters. We define the following parameterized property.

> **No Unwanted Overflow:** *Additions, subtractions, multiplications and negations must not overflow, except at given program points.*

This property is very important as overflows can make the application state inconsistent and hence lead to security flaws. Here is an example, difficult to find using test cases, where a possible overflow hides an error in the implementation of a specification.

```
short balance, maxBalance, credit; // Assumption: variables always positive

if (balance+credit > maxBalance)   // BAD, overflow is possible
  throwIt(SW_CREDIT_TOO_HIGH);
else
  balance += credit;           // BAD, overflow possible, balance possibly negative
```

With this implementation, crediting the purse can not only be turned into a debit but also can introduce an unexpected state as the balance, assumed to always be positive, can now be negative. This piece of code should rather have been implemented as follows.

```
short balance, maxBalance, credit; // Assumption: variables always positive

if (balance > maxBalance-credit)   // OK, no overflow possible
  throwIt(SW_CREDIT_TOO_HIGH);
else
  balance += credit;               // OK, no overflow, balance stays positive
```

A similar example could lead a debit transaction to actually crediting the purse.

3.7 Manipulation of Plaintext Secret

The hardware of smart cards is designed to be a kind of safe. However, it is not totally tamper-proof. Using appropriate devices, it is possible to observe the memory and the behavior of a running smart card. As a consequence, an applet should store and manipulate *plaintext* secrets (as opposed to crypted secrets) as little as possible.

As a first step towards limiting the manipulation of plaintext secrets, we consider the following property

> **Use of Plaintext Secret:** *If a program reads a given plaintext secret information in its memory, it must use it.*

In other words, there must be “good reasons” for taking the risk of a hardware observation. The secret (e.g., a key) must be used in some useful computation (e.g., the construction of a session key). Note that an observable use of a computation involving a plaintext secret can be in a subsequent call to method process().

Here is an example based on Demoney where a plaintext secret (the credit key) is used to construct a session key, that is used in turn to exchange data between the terminal and the applet.

```
byte[] keyData, random;
DESKey creditKey, sessionKey;

install(...) {
  keyData = makeTransientByteArray(...);
  creditKey = KeyBuilder.getInstance(...);
  sessionKey = KeyBuilder.getInstance(...);
}

process(...) {
  switch (...) {
  case PERSONALIZE:
    ...
    creditKey.setKey(...);   // Secret key is stored into the applet
    ...
  case CREDIT_AUTHENTICATE:  // Sometime, later, on other call to process()
    ...
    // Access to a plaintext secret
    creditKey.getKey(keyData, (short)0);
    // Computation of a session key in plaintext
    for (short i = 0; i < (short)16; i++)
      keyData[i] ^= random[i];
    // Record new session key
    sessionKey.setKey(keyData, (short)0);
    // BAD, part of plaintext secret never used later on
    random[0] = keyData[0];
    // Erase plaintext secret (but use of plaintext secret still visible in sessionKey)
    arrayFillNonAtomic(keyData, (short)0, (short)16, (short)0);
    // Appropriateness of other use of plaintext secret (to compute session key) still pending
    return;
    ...
  case GET_DATA:             // Later again, on yet another call to process()
    ...
    signature.init(sessionKey, MODE_VERIFY);
    signature.sign(inBuf, ..., apduBuf, ...);
    apdu.sendBytes(...); // OK!, previous use of plaintext secret now observable
  ...
}
```

Analyzing this property requires taking into account in particular features such as the applet lifetime (§2.1) and entry points (§2.2), as the use of a plaintext secret may span across multiple calls to the applet entry points.

4 What Analyses?

Static analyses are approximations. When given an application, a coarse analyzer can always answers “maybe something can go wrong”, which is not very useful. This concluding section elaborates on this issue and discusses the characteristics of analyses that are to verify the properties given the previous section.

Precision vs. Complexity. All incorrect programs must be detected, with as little false positives as possible. It is alright if the analysis takes time: a low complexity is important but precision is even more important (provided an answer can be computed in a few hours). It must be also remembered that Java Card applets are small (typically, a few Kbytes): a complexity that would be impractical for most real-life programs can perfectly make sense in the Java Card world.

Feedback Information. To provide useful feedback to the applet programmer or owner, it is not enough to say “typecheck error in program Foo” or “unsatisfiable constraint 237”. Error and warning messages should include items such as the location in the code and the involved program entities (data areas, etc.). While prototypes developed during the SecSafe project do not have to be fully usable in terms of user-friendliness, the techniques used for the analyses must allow the traceability of information so that this user-friendliness is possible if an actual product is later developed.

The assessment of the above features — as part of the SecSafe studies — will have to include comparison to related work, and in particular:

- differences in the nature of security properties,
- differences in precision,
- differences in usability.

References

[Mar01] Renaud Marlet. Syntax of the JCVM Language To Be Studied in the SecSafe Project. Technical Report SECSAFE-TL-005, v1.7, Trusted Logic, May 2001.

[MM01] Renaud Marlet and Cédric Mesnil. Demoney / Loyalex — A Demonstrative Electronic Purse And A Loyalty Application. Technical Report SECSAFE-TL-007, v1.0, Trusted Logic, June 2001.