REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
02-09-2003	Final Report	15 August 2001 - 15-Aug-03

4. TITLE AND SUBTITLE

Mobile Language Study

5a. CONTRACT NUMBER
F61775-01-C0006

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Professor Mads Dam, Pablo Giambiagi

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Swedish Institute of Computer Science AB
Kistagaangen 16
Kista SE-164 29
SE-164 29
Sweden

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0014

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
SPC 01-4025

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Swedish Institute of Computer Science AB as follows: Contractor will investigate security issues related to mobile code (dynamically loaded code) which can potentially compromise information system security. This is of special concern to the Air Force because of requirements to access data at various levels of classification. The contractor proposes realizing information flow controls for mobile code by expressing security policies as admissibility properties. The proposed architecture to be investigated has the following properties: code annotations and security guarantees, program analysis to verify security guarantees, and a specially designed user interface to aid in handing data of different security levels

The proposed effort spans two years and is divided into four tasks. Tasks 1 and 2 will be completed in the first year and Tasks 3 and 4 are to be completed the second year.

Task 1: Program analysis techniques for admissibility

Task 2: Experimentation and prototyping

Pending successful completion of the first year's research and availability of funding, the second years tasks include:

Task 3: Security architecture

Task 4: Java Card and JCVM

15. SUBJECT TERMS

EOARD, Software, Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18, NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			CHRISTOPHER E REUTER, Ph. D.
UNCLAS	UNCLAS	UNCLAS	UL	60	**19b. TELEPHONE NUMBER** *(Include area code)* +44 20 7514 4474

SPC 01-4025 Mobile Language Study
Final Technical Report

Mads Dam Pablo Giambiagi
Swedish Institute of Computer Science
Box 1263, S-164 49 Kista, Sweden; {mfd,pablo}@sics.se

August 18, 2003

1 Introduction

Security critical applications are now within reach of the common man. We no longer need to examine nuclear power plants, armed forces facilities or air traffic control towers to discover software that is expected to comply with the most stringent guarantees of correct operation. At an obviously more modest, personal level, it has become critically important that the software we use on a day-to-day basis protects our identity as well as our financial assets. Current smart card technology is a good example of an architecture operating at this level. A *smart card* is a plastic card, enhanced with small memory and processing units, commonly used to identify bank account holders, company employees or club members, and even as a convenient means of payment. In this project we have explored techniques and methodologies to increase the protection of confidentiality with a focus on smart card applications.

Smart cards can be programmed using general-purpose languages; but because of their limited resources, smart card programs are designed to perform very simple tasks. They delegate all other work to external systems with which they communicate through the card reader (which is the card's only interface). As a consequence, a card's duty is confined to administer small but very delicate units of information, like PIN numbers, keys and account balances, all of them critical ingredients in the completion of tasks like payments, cash withdrawals and access authorizations. If these information units were to reach the wrong hands, the damage to the card owner and eventually to the service providers could be considerable.

Although it not difficult to find other examples of secrecy critical applications for the individual, smart cards are nonetheless representative of a range of information-enriched devices to which we may entrust (whether aware or not) our identity, access codes or any other confidential data (e.g.

This material is based upon work supported by the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory, under Contract No. F61775-01-C0006. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the European Office of Aerospace Research and Development, Air Force Office of Scientific Research, Air Force Research Laboratory.

the SIM card needed to operate a GSM mobile phone is a smart card that identifies the owner of the phone line. Similar identification units are being proposed to interconnect all sort of personal gadgets). The more we rely on these devices, the higher our vulnerability in cases of lost confidentiality.

What makes smart cards specially suitable for this study? First, smart cards feature tamper-proof hardware which precludes most low-cost physical attacks, and a simple interface that reduces the number of information channels to consider. Moreover, the existence of highly adopted standards and common high-level card operating systems extend the applicability of any eventual analysis to a wide range of platforms. For the researcher there is even one further challenge: smart cards can host several applications at a time, adding extra complexity to the study of confidentiality.

In this report we examine the state-of-the-art in software-based secrecy verification, and discuss its chances to scale up to real Java Card applications (Java Card is one of the most widely deployed operating systems/programming infrastructures for smart cards). We stress the need to cope with the one aspect that sets these applications apart from the computer systems traditionally studied using information flow approaches: namely, an overall reliance on cryptographic protocols to manipulate and exchange secret information.

Standard models of secrecy take an operating systems viewpoint. Resources, i.e. files and processes, are accessed by other processes on behalf of users. It is well-known that access control methods are generally insufficient to prevent covert channels in this setting. So the alternative is to use a multi-level security model (MLS) and define an information flow property over it. In the MLS model, the objects and subjects of the system are classified into security levels (or *clearances*), and information flow properties are designed to forbid all flow of information from higher to lower levels.

Cryptography does not fit well into this picture. A high-level value, once encrypted, may be accessed by low-level users that have no knowledge of the decryption key. While in strict information theoretical terms this corresponds to a flow from high to low classification levels, in practical terms a high-level value has been declassified, i.e. made into a lower classification datum. Smart card applications make extensive use of this type of declassification that most information flow techniques (e.g. noninterference [16]) cannot cope with.

Because declassification lies outside most information flow models based on noninterference and MLS, the way to handle it has mainly been to delegate, on an external trusted agent, the responsibility of deciding whether a declassification is safe. In spite of this, noninterference models make the most thoroughly investigated approach to information flow analysis. Furthermore, they form the basis of various language-based techniques with the declared potential to verify real code, either automatically or semi-automatically (see [28] for a recent survey). Therefore, we take noninterference as a starting point for our investigations.

We organize our presentation around a case study, a realistic Java card applet implementing the Needham-Schroeder-Lowe public-key mutual authentication protocol, which we present in Section 2. We discuss some of the most important properties that such implementation should satisfy to guarantee confidentiality in Section 3. Facing the challenge of verifying those secrecy properties, we first report in Section 4 on our experience applying a noninterference approach to our code example: the decentralized label model of Myers [25], a rich instance of the MLS lattice. Although our example does not require the full expressive power of decentralized labels, our interest in this

model is easy to justify. A. Myers and his colleagues at Cornell have developed Jif [24], an extension of the Java language with a powerful static and dynamic type system for the verification of flow properties expressed in the decentralized label model. Besides being one of the few tools that deal with a realistic programming language, Jif is also rare in that it lets developers signal explicit label declassifications. However, being based on a noninterference labeling model, it provides no assistance in checking how declassifications affect confidentiality.

In the domain of smart card applications we can expect that declassification through cryptographic operations is justified by some cryptographic protocol that the code is designed to implement. This observation prompted us to seek a way of explaining the correctness of declassification operations by first verifying the correct implementation of the protocol. In Section 5 we review a very successful approach to the verification of functional properties using the extended-static checker ESC/Java. This tool, based on a weakest-precondition calculus for object-oriented guarded command languages is neither specially designed for Java Card nor is prepared to track information flows (as Jif does).

Our main thesis is that, in order to scale up secrecy verification to realistic code (that makes consistent use of cryptography), we need to combine the two approaches above. Tools like Jif lack the knowledge of the implemented protocol to reason about declassifications, while tools designed for the verification of safety properties of code (of which ESC/Java is just one example) lack the ability to reason about information flows. Furthermore, we sustain that such a combination of approaches can be given a formal foundation that does not differ much from recent process-algebraic presentations of information flow.

In Section 6 we review the definition of Admissibility [6, 11, 14] which is an information flow property parameterized by a dependency specification. The latter expresses the dependencies between API method calls that produce and/or consume secrets as stated by the abstract cryptographic protocol that the code should implement (A more detailed discussion of admissibility appears in [15], as an attachment to this report). The practical application of admissibility to our case studied is discussed in Section 7. The presentation uses an object-oriented guarded-command language, establishing the connection with tools like ESC/Java. We also discuss how to scale the approach to Java Card.

Finally, Section 8 mentions related work and Section 9 presents our conclusions. Appendix A contains a summary of the activities performed within the SPC 01-4025 Mobile Language Study project.

2 Case Study: The NSL Authenticating Applet

In this section we introduce a case study that is used through this whole report to illustrate and compare the different approaches to the verification of secrecy. As a representative of the applications raising concerns on their confidentiality properties we discuss (a part of) a Java Card applet implementing a well-known and well-studied mutual authentication protocol.

2.1 The Scenario

Consider a smart card equipped with a number of Java Card applets, each possibly produced by a different provider. When the card holder feeds the card into the card reader, which of all those applets gets actually executed?

As a first measure of protection, an applet may request the insertion of a personal identification number (PIN) before performing a task. In order not to exasperate users making them memorize a different PIN for each applet, the Java Card architecture provides an efficient way of associating a single PIN with the card. Applets can demand that the PIN be verified, without having the ability to ever set or update the PIN value. Such an implementation is routine and will not be discussed further in this report. The main point to make is that PIN authentication is an all too weak security mechanism that, in a realistic situation, does not give guarantees to the card holder on the specific applet that is actually executed by the card reader.

In fact, the user may never know which applets were selected by the card reader. All that matters is that this cannot be abused, and that no applet can be fooled into handing in secret information or enabling services for recipients other than the card holder. On the side of the service provider this amounts to correctly authenticating the identity of the card holder, so that the right user gets charged for the service. On the side of the applet itself, this requires authenticating the service provider's server to make sure that no third party gets hold of the secret credentials stored in the applet.

2.2 The Protocol

Our case study applet shows just one way, among many others, to accomplish mutual authentication between an applet and a service provider. It relies on an implementation of the Needham-Schroeder-Lowe public-key (NSLPK) protocol [23]. Besides guaranteeing mutual authentication between two principals, this protocol lets them agree on a temporary key, which could easily be used to secure all (further) communications in the current session. Figure 1 presents NSLPK in the customary notation for cryptographic protocols, where A stands for the Applet, S for the service provider's server, n and m for nonces, and $\{x\}_{Pub(B)}$ for the encryption of plaintext x under B's public key. A protocol round is initiated by the applet when it sends its *id* to the server. Observe that an NSLPK protocol round actually ends with the reception of message (4). We have added message (x) to illustrate how the applet (and the server) can then use m as a shared key for the rest of the session. It is well known that the notation of Figure 1 is far from formal, in the sense that the pretended secrecy properties of the protocol may fail to hold for some of the interpretations that are possible given the sequence of messages above. For example, nothing in the figure tells us what a protocol participant should do in the event of receiving a malformed or out-of-order message. The intended interpretation for the NSLPK protocol is that in all those cases the protocol round should be immediately terminated.

$$
\begin{array}{lll}
(1) & A \rightarrow S & : A \\
(2) & S \rightarrow A & : \{n, S\}_{Pub(A)} \\
(3) & A \rightarrow S & : \{n, m, A\}_{Pub(S)} \\
(4) & S \rightarrow A & : \{m\}_{Pub(A)} \\
(x) & A \rightarrow S & : \{secret\}_{m}
\end{array}
$$

Figure 1: The Needham-Schroeder-Lowe Public-Key Protocol

2.3 The Applet

In the Java Card execution model, a card and the applets it contains, always operate in client-server mode. Each iteration starts with the card reader sending a request to the Java Card Runtime Environment (JCRE) located on the card. The request takes the form of an APDU (Application Protocol Data Unit) which the JCRE can either process or forward to the currently selected applet. When a response is available, the JCRE passes it back to the card reader, after which a new cycle may begin. The APDU commands destined to the JCRE serve the purpose of installing new applet classes, registering applet instances and selecting/deselecting applets. APDU commands directed to the applet are digested by the applet's *process* method.

A common applet development strategy starts by identifying the stages an applet can go through during its lifetime, e.g. *Installation, Personalization, Processing* and *Locked*. Since we are more interested in how the applet can achieve mutual authentication with the server provider, rather than in its complete behavior, we limit our attention to the *Processing* stage and, in particular, to the stage changes caused by the reception of APDUs specifically related to the implementation of the NSLPK protocol. Following the methodology proposed in [19] we present a finite state machine (FSM) describing the state transitions produced by the *process* method in the applet at the different points in the execution of the NSLPK protocol (Fig. 2). The first transition, *sendId*, emits message (1) and leads the applet to the WAITING state. The reception of message (2) fires the *sendChallenge* transition whose response to the card reader reflects message (3). Finally, the *sendSecret* transition is fired by message (4). Its result is the transmission of message (x).

Such a finite state system suggests immediately the following implementation of the *process* method:

```
public void process(APDU apdu)
throws ISOException, APDUException, CryptoException
{
    switch(state) {
    case STATE_INIT: sendId(apdu); break;
    case STATE_WAITING: sendChallenge(apdu); break;
    case STATE_CHALLENGING: sendSecret(apdu); break;
    default: state = STATE_INIT;
             ISOException.throwIt
                 (ISO7816.SW_FUNC_NOT_SUPPORTED);
```

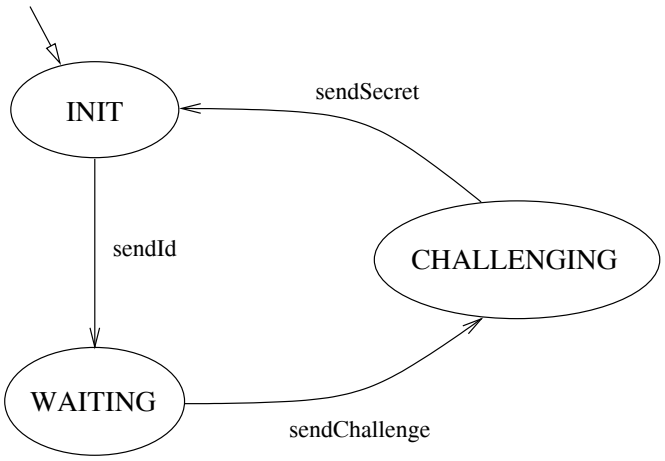

Figure 2: A finite state machine for the NSL Applet

```
    }
}
```

If, for some reason, the applet ends up in a state outside those in Fig. 2, an exception of type *ISOException* is raised. The JCRE is then responsible to translate it into a special APDU message containing the error code passed as a parameter to the *throwIt* method.

Take a look at the transitions in Fig. 2. They are certainly not atomic (e.g. *sendChallenge* takes care of receiving message (2), processing it and putting together message (3)). If anything goes wrong the applet should return to the INIT state. In fact, what we have is an FSM as it appears in Fig 3, where we have added *select* transitions to represent the re-selection of the applet (e.g. when the card is first removed and then reinserted into the reader). Observe that verifying that the applet implements the FSM in Fig. 3 requires checking (a) that each of the methods *sendId*, *sendChallenge*, *sendSecret* and *select* correctly establish their corresponding target state, and (b) that all error situations cause a transition to the INIT state together with a proper notification to the JCRE (in terms of an ISOException).

Verifying that the applet code (listed in Appendix B) actually satisfies these requirements is not as trivial as it may seem due to the possible throwing of many unchecked exceptions (In Java Card, as well as in Java, unchecked exceptions need not be mentioned in the throws clause of a method, so that the type-checker would tell us nothing about them). Since it is always desirable that the card reader gets a proper notification of any errors encountered during applet execution, card software developers look for guarantees that their code can only throw exceptions from within a very small set, commonly including *ISOException*. If that knowledge is available the verification of error transitions in Fig. 3 gets considerable simpler (of course, we still need to verify that no other unchecked exceptions can ever be thrown).

A few comments on the applet code in Appendix B: During the installation phase, after the class code is loaded onto the card, the *install* static method is invoked by the JCRE. At this point an

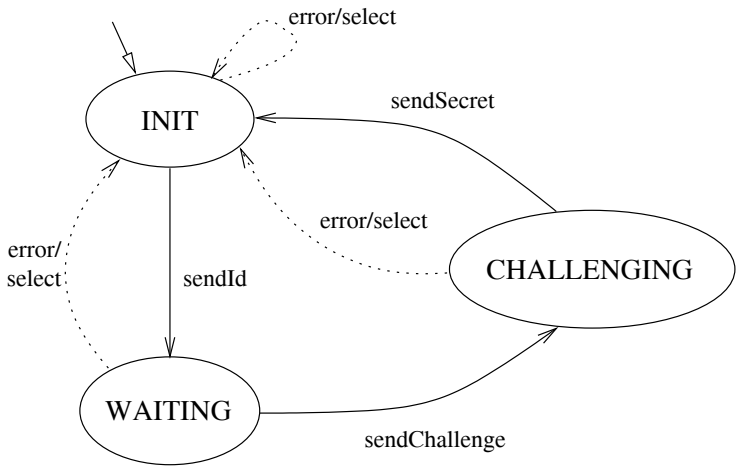

Figure 3: A finite state machine with error and select transitions

instance of the applet object is allocated and registered. The parameters of the *install* method serve to initialize the instance fields holding the applet and server id's (which, for the sake of simplicity, have been modeled as single bytes). This is ok as long as we are interested in having at most one instance of the applet in each card. The applet's RSA private key and the server's public key cannot be loaded using this method. The reason is a mere technicality: RSA keys would not fit into the *bArray* parameter of the *install* method (it cannot hold more than 32 bytes). In practice they must be uploaded during the *Personalization* phase, which has been left out from our code in its entirety. However, a cipher can only be fully initialized after getting its key, which means that the initialization of the cipher should be completed during the *Personalization* phase. This would be a serious problem if we wanted to execute our applet, but for the purpose of presenting and discussing secrecy verification we can simply put the initialization of the ciphers in the applet constructor assuming that the keys have been previously loaded into the fields *my_private_key* and *server_public_key*:

```
protected NSLClient(byte[] bArray, short bOffset, byte bLength)
    throws SystemException, CryptoException
{
    ...
    enc.init(server_public_key, Cipher.MODE_ENCRYPT);
    dec.init(my_private_key, Cipher.MODE_DECRYPT);
}
```

3 Secrecy of Java Card Applets

The NSLPK protocol guarantees the mutual authentication of principals S and A (cf. Fig. 1) and the secrecy of the agreed session key, provided that the private keys are only known to their owners and that the protocol steps are followed to the letter. Achieving this in practice, using a programming language with such a complex semantics as Java's is harder than what one may first think. There

are just too many possible program paths to consider, exceptions can be raised almost anywhere, nonces can be reused, messages be wrongly constructed or interpreted, ciphers could be initialized with wrong keys, etc. There are also many ways in which secret information could be leaked, even when the programmer is not trying to introduce security holes on purpose. In general one may classify information leaks into *explicit* and *implicit*. An example of an explicit leak would be a response APDU containing (part of) the private key in plaintext form. The notion of implicit leak is slightly harder to grasp. It is characterized by the encoding of secret information in terms of variations of observable behavior. Therefore, the nature of the indirect leaks to consider goes hand in hand with the observation powers we assume possible (or, at least, feasible if we introduce cost-efficiency considerations). Examples of indirect leaks include sending different seemingly innocuous messages, changing the time it takes to answer a process APDU or the probabilistic distribution of APDU contents, affecting termination behavior and even modifying power consumption patterns, all depending on the value of the secret to leak.

For our case study implementation we want to make sure that it preserves the secrecy of the applet's private key (field *my_private_key*), the temporary key *temp_key* and the secret value exchanged in message (x) (i.e. field *secret*). For the moment our interest is in detecting direct and indirect possibilistic channels only. We do not seek to detect timing, probabilistic or power channels. Although this would be desirable, it constitutes a longer term project. Many problems posed by possibilistic channels are still pending a solution in practical terms. Observe that, because we have opted not to represent the *Installation* and *Personalization* phases of the applet's lifetime, we cannot represent the moment some of these pieces of confidential information enter the applet. But we assume that this is done by a trusted user that places them only in the fields mentioned above. The *temp_key* is locally generated and the contents of the *secret* field should also be (this is not reflected in the code, though).

We are interested in verifying that the applet described in Section 2 does not leak directly or indirectly the secrets mentioned above. Given that these secrets are used to compute legal APDU responses, we must in fact relax such a strict requirement. In fact, what we need to verify is that, if any of the secrets is ever leaked, then this happens precisely in the way ordered by the NSLPK protocol.

4 Java Information Flow

In this section we discuss the noninterference approach to information flow. In particular, we apply the distributed label model to keep track of how the applet manipulates its secret data. This exercise makes heavy use of the declassification capabilities of the decentralized model. This, in turn, motivates some later comments on the model's inadequacy to justify declassifications from an information flow point of view.

4.1 The Model

The decentralized label model belongs to the class of multi-level security (MLS) models. Common to these models is the idea of using an extended language semantics where principals and objects

(values) are labeled with security clearances. The set of clearances is given a lattice structure under a partial order relation $\sqsubseteq$ so that a value labeled l can be inspected by users of clearance h only if $l \sqsubseteq h$. Since a label constitutes an abstraction of the actual semantic value, MLS models are quite appropriate for the application of diverse language-based techniques to the verification of information flow security.

A standard language-based approach consists in the description of a type system capable of determining good static approximations of runtime labels. This usually requires associating labels to positions in the store (object references and fields in an object-oriented language), and defining ad-hoc method signatures to describe how the execution of a method affects the distribution of confidential data in the store.

In the decentralized label model [24] labels are sets of *policies*. Each policy has the form $o : r_1, \ldots, r_n$, where o is the principal that owns the policy and gives read authorization to principals r_1, ..., r_n. If a value v is labeled $\{o : r_1, r_2\}$ then v is owned by o and can be read also by principals r_1 and r_2. To declassify, the owner o gives read access to more principals. To read a value, a principal must be authorized by each of the policies in its label.

Using this model, Myers has developed a type system for Jif [8], an extension of the Java programming language that incorporates notation to label field and method signatures. Although the decentralized label model is very expressive, our case study needs only simple policies. Essentially, we would like to differentiate two security clearances, one for data that is public, and the other for data that is secret. Public data is marked with the empty set of policies, $\{\}$. Secret data is marked as owned by the principal P, i.e. with label $\{P :\}$. This principal represents the applet, and the JCRE is expected to act for it, so that invocations of the different applet routines (e.g. *install* and *process* may use P's authority to declassify information when the protocol permits it. In Jif, the expression "$declassify(e, L)$" is used to change the label of expression e into L, if L differs from e's label only in policies owned by the executing process. The statement "$declassify(L)S$" executes S assuming the entry level of pc (the program counter) is L.

4.2 Jif Annotation of the NSL Applet

We can now proceed to annotate the NSLClient applet.

Field annotations: To start with, we identify those applet class fields whose content should be protected. In Jif, the private RSA key and the session key fields are annotated as owned by P. Instead, the default label of a class field is $\{\}$, so there is no need to annotate field *server_public_key*.

```
private RSAPublicKey server_public_key;
private RSAPrivateKey{P:} my_private_key;
private DESKey{P:} temp_key;
```

To protect the *secret* field, we not only give it label $\{P :\}$, but also change its type to $\mathbf{byte}\{P :\}[]$, a byte array where each element is also owned by P (cf. [24] for details on label-parametric types).

```
private final byte{P:}[]{P:} secret = ... ;
```

Method signatures: In Jif, a method signature can be annotated with security labels. As with standard type systems, this allows compositional verification, by avoiding the analysis of the method body each time the method is called. The label immediately after the method name is an upper bound for the label of the pc at invocation. It is called "the *end-label* of the method". The label associated to each formal parameter is there to restrict the label of the corresponding actuals; and the label after the colon, the *end-label* ($\{P:\}$ in the example below), conveys the information that is learned by observing the normal termination of the method.

```
public void process{}(APDU{} apdu):{P:}
    throws (APDUException, CryptoException, ISOException)
    where authority(P)
```

The *where* clause in the example above tells Jif to make sure that the *process* method is called only if the caller has the required authority.

Notice that it is expected that the normal termination of method *process* conveys some information labeled $\{P:\}$. This must a priori be considered a leak, since the information is bound to reach the card reader. It will be our task in the coming sections to justify that this leak is not dangerous.

One further point: Here it is assumed that the JCRE, which invokes *process*, has the authority to declassify data owned by principal P, but will not use method invocations to leak information conveyed by its pc. It is beyond our intention here to actually verify that this is so, since the JCRE lies outside the subject of study. A similar argument is used to annotate the Java Card API (more on this soon).

Label propagation: It is expected that partial results, computed from the secrets defined above, are stored into other class fields:

```
private byte{P:}[]{P:} outBuf, nonce;
```

That the nonce must be kept secret is clear, but notice that the applet is responsible for its generation. Therefore, we must make sure that each new nonce value is not manipulated before it is stored in the *nonce* field. We do this by attaching label $\{P:\}$ to the random number generator that produces the nonces:

```
private RandomData{P:} rand;
```

This is not enough, though. The signature of the *generateData* method, used to actually obtain a new random number, must reflect the idea that a *secret* RandomData object produces *secret* random numbers:

```
/* In class javacard.security.RandomData */ abstract public void
generateData(byte{this}[]{this} buffer, short{this} offset,
short{this} length);
```

Here **this** is a label *parameter* that gets assigned, upon method call, the label of the RandomData instance. Since the field *rand* is labeled $\{P:\}$, this means that the statement

```
rand.generateData(nonce, (short)0, NONCE_LENGTH);
```

forces nonce to be of labeled type **byte**$\{P:\}[]\{P:\}$, as wanted.

Ciphers: Like random number generators, Ciphers can *generate* secret data. This can happen for example if a public ciphertext is decrypted using a secret key. In our applet, this is the case with cipher *dec* which is initialized using the applet's private key. We tell Jif about this property of the cipher by labeling *dec* (and, for the same reason, cipher *sCipher*) with $\{P:\}$. On the other hand, the *enc* cipher is labeled $\{\}$ because it is only used to encrypt with the applet's public key.

```
private Cipher enc;
private Cipher{P:} dec, sCipher;
```

As with class RandomData, the API class javacardx.crypto.Cipher must be annotated to propagate labels appropriately. There is, however, two small complications: a cipher must be first initialized with a key; and then the security level of the output produced by a decryption/encryption operation depends not only on the level of the key, but also on the level of the input byte array. The solution, for both methods, is the same. All parameters of *doFinal* and *init* must be bounded by the label of the Cipher instance.

```
public abstract class Cipher{
  ...
  public native short{this}
      doFinal(byte[]{this} inBuff,
              short{this} inOffset, short{this} inLength,
              byte{this}[]{this} outBuff,
              short{this} outOffset):{this}
      throws (CryptoException);


  public native void init{this}(Key{this} theKey,
                                byte{this} theMode)
      throws (CryptoException);
  ...
}
```

Observe that, in Jif, arrays labels are invariant. This means that $L \sqsubseteq L'$ does not imply that $\mathbf{byte}\{L\}[\,] \leq \mathbf{byte}\{L'\}[\,]$, where $\leq$ is the subtype relation. Therefore, Jif would reject the following statement

```
dec.doFinal(inBuf, (short)2, KEY_LENGTH, inBuf, (short)0);
```

because the type of inBuf is not a subtype of $\mathbf{byte}\{P:\}[\,]$.

The example also illustrates a very convenient feature of Jif. API classes and method signatures can be given without having to annotate method bodies. This is achieved by marking those methods as *native*. The Jif compiler would then use the signature for the analysis, while runtime environment (not described nor used in this report) can still use the original API classes which contain the executable code. In this way, one looses the possibility of actually checking the API implementation (although the actual code can be added later if wanted), but lets one get started with an initial version of the full Java Card API with minimum effort (Our annotated Java Card API signatures will be made available from the project's web site).

Class annotations: We have called P the principal that owns the secret data stored in the applet. How could one change that, so that different instances of our applet are owned by different principals? This is not a real problem, since in Jif classes can be parametric on labels and principals. In fact, P is just a parameter with which we have parameterized the class:

```
public class NSLClient[principal P] extends Applet[P] authority(P) {
...
}
```

Notice first that this class has also an *authority* clause. This says that the process that creates an instance of $NSLClient[P]$ must have P's authority.

The superclass, i.e. javacard.framework.Applet, should also be parameterized with P because method overrides cannot freely modify a signature. For example, we must allow the install method to operate on secret data, and perhaps have different termination behaviors due to it. This is not a problem if it is guaranteed that the installation can only be done by principals with the authority to read those secrets, which is reasonable.

```
abstract public class Applet[principal P] authority(P) {
  ...
  public native static
      void install{}(byte[]{} bArray, short{} bOffset,
                     byte{} bLength ):{P:}
      throws (ISOException, SystemException, PINException);
  ...
}
```

Exceptions: Exceptional behavior can leak information, like any other branching of control. Therefore, Jif makes sure that all possible exceptions are accounted for in the signature of a method. This implies that, contrary to Java's approach, Jif requires both *checked* and *unchecked* exceptions to be listed in the throws clause of method. However, this leads to the unnecessary propagation of begin- and parameter-labels into end-labels. This loss of precision can easily result in the rejection of an otherwise completely safe method.

To overcome the problem we can introduce try-catch statements around the body of the method to prevent Jif from assuming the possible occurrence of an exception. Naturally, in order to do such a transformation safely, one must also prove that it is really the case that the exception cannot happen (This problem is addressed in Section 5).

```
private void sendId{}(APDU{} apdu)
    throws (APDUException)
{
    try {
        apdu.getBuffer()[0] = appletID;
        apdu.setOutgoingAndSend((short)0,(short)1);
        state = STATE_WAITING;
    }
    catch (NullPointerException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
}
```

Declassifications: Our NSL applet uses a single method to transmit messages (3) and (x) (cf. Fig. 1). This serves the important purpose of unifying in a single point the immediate requirement that only public information is transmitted to the card reader:

```
private void sendMessage{}(APDU{} apdu, byte{}[]{} buf,
                           short{} offset, short{} len)
...
```

As expected, all actual parameters to the method are requested to be public. Furthermore, the declaration includes the begin-label $\{\}$ to ensure that the invocation of *sendMessage* is not used to leak the label of the pc.

This decision forces us to take action to make Jif type-check the two calls to *sendMessage* (in methods *sendChallenge* and *sendSecret*), as they happen both at points where the program counter is labeled $\{P :\}$ and pass field *outBuf*. Since the calling methods ran with the authority of principal P, they can perform the necessary declassifications:

```
declassify({}){
    sendMessage(apdu, declassify(outBuf, {}),
                (short)0, MESSAGE_LENGTH);

    state = STATE_CHALLENGING;
}
```

4.3 Lessons Learned

Jif operational for real Java Advantages: progressive annotation of API files Disadvantages: buggy, still in development also, label creeping problem. Exceptions. Declassification mechanism is ok for data values, but it is hard to declassify the effects of invoking a method on the pc and exception paths. Our attempts to catch exceptions and use Jif's *single-path rule* (which re-establishes the entry pc after a sequence of statements if they can only terminate normally) where unsuccessful.

As mentioned before: no way to justify declassifications. Uses of *declassify* could be justified proving appropriate invariants, that should hold immediately before the statements requiring declassification (for example, just before a call to *sendMessage*).

The complete annotated source for the NSLPK applet appears in Appendix C.

5 Verification of Protocol Implementations

Much has been done in the business of verifying software for compliance with high-level, abstract specifications. Although the issue is not settle yet, there exist already tested approaches that happen to be quite successful when it comes to verifying sequential imperative programs. Object-oriented programming languages, specially because of dynamic binding, brought some new difficulties into this ground, but all the same, we can now find good verification tools to test, for example, that our NSLPK applet implementation behaves as described by the FSM model of Fig. 3.

A verification approach that has recently been applied to the verification of protocol implementations uses the Java Modeling Language (JML) and diverse tools based on an axiomatic semantics of the Java Card language [18, 17]. An example of these tools is the ESC/Java extended-static checker [22]. In order to apply ESC/Java, one begins by annotating the (Java source) code with class invariants and method pre- and post-conditions, all written in a a subset of the JML language. ESC/Java then applies a verification condition generator to produce a first-order logic formula that is subsequently given to an automatic theorem prover.

ESC/Java is neither sound nor complete, but anyway is able to find out many potential problems in code. If there is the need for stronger correctness guarantees, one may then use a more precise semantic model (e.g. as in the LOOP tool [30]). However, this comes at the high cost of lost automation.

Since this approach has been described in detail elsewhere in the literature (e.g. in [18, 17]), we only give an overview here on how it is applied to the verification of the NSLPK applet.

Our initial concern is to make sure that the applet implementation behaves as mandated by the FSM of Fig. 3. We start by identifying a class invariant that characterizes the possible states in the FSM, and the allocation status of several instance fields (obviously, an invariant is meant to start holding only after the execution of the class constructor):

```
/*@ invariant
  @       (state == STATE_INIT || state == STATE_WAITING ||
  @        state == STATE_CHALLENGING)
  @   && inBuf != null && inBuf.length == MESSAGE_LENGTH
  @   && outBuf != null && outBuf.length == MESSAGE_LENGTH
  @   && nonce != null && nonce.length == NONCE_LENGTH
  @   && dec != null
  @   && enc != null
  @   && rand != null
  @*/
```

Consider now the possible transitions originating in state INIT (cf. Fig. 3). The execution of the *sendId* method should leave the applet in the WAITING state, except when an exception is thrown. In the latter case, the applet should remain in the INIT state. This is specified in JML with the

following annotation:

```
/*@ requires
  @    apdu != null && apdu._APDU_state == 1
  @    && state == STATE_INIT;
  @ modifies
  @    state, apdu.buffer[*];
  @ ensures
  @    state == STATE_WAITING;
  @ exsures (Exception) state == STATE_INIT;
  @*/
  private void sendId(APDU apdu) throws APDUException { ... }
```

There are four clauses in this JML annotation: The first one, a "requires" clause, describes the method precondition, and the "modifies" clause lists the object fields and references that may be modified during the execution of the method. The last two clauses, "ensures" and "exsures", describe the state that should hold after normal, resp. exceptional, termination, provided that the precondition held at the time the method was invoked. In this case, the specification states that the *sendId* method should be passed a non-null APDU and should be invoked from state INIT. If that is the case, this method will either terminate normally in state WAITING, or exceptionally in state INIT, or it will not terminate at all.

Actually, checking that our NSLPK applet implementation respects the transitions in the FSM of Fig. 3 gains us little headway for the verification of confidentiality. But it definitely helps rule out several execution paths, and with them, its becomes possible rule out leaks that otherwise may be considered plausible. If we go back to the Jif model of the previous section, we can now confirm that all the empty exception handlers we introduced then do not really change the behavior of the program. ESC/Java can show that the corresponding exceptions are never thrown.

A more demanding task concerns the justification of the declassification statements we introduced in the Jif model. These come in two flavors: First, value declassifications, like in the expression declassify(outBuf, $\{\}$). The idea here is to prove an invariant at this precise program point to show that every time the outBufis declassified, its value agrees with what the protocol mandates of the corresponding response APDU. The second declassification flavor corresponds to the declassification of the pc label, like in the statement

```
declassify({}){
    sendMessage( ... );
}
```

For this case, it does not help to prove an invariant at the precise declassification point. Indeed the pc could have gotten its high level time before in some execution path leading to this point. The way to handle this case is to identify all points of control branching and, again, proving that they occur according to the protocol.

Actually, the latter approach is needed also to justify the end-labels given to methods *process*, *sendChallenge* and *sendSecret*. In all those cases the end-label is $\{P :\}$ which indicates that the normal termination of these methods may leak secret information. Obviously, our job would not be complete if we did not justify them as admissible declassifications, occurring in accordance to the

protocol. To do this we need to justify all branching of control due to exception throwing, which calls for a similar approach as for the declassify statement (but notice that Jif provides no statement for the explicit declassification of a method's end-label).

The full source for the NSLPK applet that is listed in Appendix D collects the ESC/Java annotations discussed so far. From this point onward the study of the NSLPK applet should proceed with the introduction of all the necessary invariants and their (usually needed) strengthening in ESC/Java.

It is our opinion that the present examples have made the case for a verification technique for confidentiality based on the tracking of value sensitivity (as provided by Jif) and the justification of declassifications of values and control branching by protocol implementation. The next section presents the theoretical background for such a combination of techniques.

6 Admissibility

We have argued that the problem of confidentiality in the context of applets which use encryption to communicate with the outside world shows the inadequacy of standard noninterference models. In this section, a semantical security condition is proposed to express and reason about confidentiality in such a setting[1]. Typically, the applets in question will implement some sort of security protocol, for authentication, secure communication, signing, etc. An applet claiming to implement such a protocol may in fact violate confidentiality for a number of reasons, advertently or inadvertently. The implementation may be a Trojan and use various forms of covert and subliminal channels to leak sensitive data. More typically, though, we are looking to catch common programming errors such as missing checks on nonces, misused random number generators, wrongly constructed field values and representations, etc. leading to information leaks that were not intended by the protocol designers. The general question we ask, thus, is this:

Does a given implementation of some security protocol contain only information flows allowed by the abstract protocol specification?

Admittedly, this is not a very precisely defined question, for two main reasons:

1. In practical cases it is not at all clear what "the abstract protocol specification" is, if it exists at all. It may be described as part of an IETF RFC or other standards document, as a sequence of abstract message exchanges in the usual style of security protocol theory, or as a formal object in some security protocol specification framework.

2. Concrete protocol implementations need to attend to a variety of tasks such as key storage and retrieval, management of cryptographic machinery, memory, buffer, and state management, address lookup, error handling, and much else, not very relevant at the protocol specification level.

[1]This brief discussion of Admissibility. A more detailed presentation can be found in an accompanying document [15].

Thus, the first problem is to find some way of specifying the intended information flows at the API level. For this purpose we introduce a simple rule-oriented notation for input/output dependencies across the protocol implementation interface. An example of such a rule might be:

$$\begin{aligned}\mathbf{send}(v, outchan) \leftarrow k := \mathbf{key}(Bob) \wedge m := \mathbf{receive}(inchan) \wedge \\ v := \mathbf{enc}(m, k)\end{aligned} \tag{1}$$

indicating that, if upon its last invocation of the $\mathbf{receive}$ method with argument $inchan$ the protocol received m (and analogously for $\mathbf{key}$, Bob, and k), then the next invocation of $\mathbf{send}$ with second parameter $outchan$ should, as its first parameter, receive the encryption of m with key k. In addition, a dependency specification determines a set of method invocations presumed to cause new secrets to be created, in this example maybe $\mathbf{receive}(inchan)$. The intention is such that, if (1) is the only rule available, then the only admissible invocations of $\mathbf{send}$ which depend directly on secrets are those which are justified by the rule. Any other invocation of any other method at the applet interface which depends, directly or indirectly, on secrets will be deemed inadmissible.

Observe that we make no attempt to quantify the admissible information flows in other ways than through direct and indirect dependencies between method invocations at the applet interface. Only recently have researchers began to apply information theoretic or complexity-based concepts to give quantitative measures of information flow in computer systems. However, these approaches are not applicable to real code (cf. [28] for an indication of the state of the art in this direction). In fact, making the confidentiality analysis of some given protocol implementation hinge on an information flow analysis of the protocol itself would, in our opinion, be a fallacy of method: It should be a matter for cryptographic protocol analysis, not program analysis, to determine whether the protocol is correct, and for program analysis to determine whether the protocol specification is correctly implemented.

The enforcement of the dependency policy proceeds in two stages: First, data flowing through the program at hand is annotated to make it possible to track direct flows whose secrecy or integrity (in the case of e.g. public keys) needs to be protected. This information, then, is used to derive a version of the admissibility condition introduced first in [6], which captures the absence of undesired information flows.

6.1 Annotations and Commands

For the purpose of illustrating our proposal we use a simple sequential language (IMP). Figure 4 shows an IMP implementation of the Initiator (the Server in our case study) in the Needham-Schroeder-Lowe public-key (NSLPK) authentication protocol (cf. Fig. 1). As discussed in Section 2, an implementation needs to deal with a lot more issues than what are explicitly addressed at the protocol specification level. Our implementation receives the identity of the responder from channel RESPONDER and its own private key from LOCALSKEY. Constant ME is the Initiator's id, and channel PORT is used for all incoming network communication. Function $\mathbf{mkNonce}$ generates a fresh nonce, $\mathbf{key}$ fetches a public-key from a local store, $\mathbf{enc}$ and $\mathbf{dec}$ encrypt and decrypt, and $\mathbf{lookup}$ returns the IP number associated with an agent id.

A standard operational semantics for IMP is extended to an annotated form, in which expressions and values are annotated with the method invocations causing them to be computed. An example

```
Program NSLPK_Initiator:

 1:  b := receive RESPONDER;
 2:  mySK := receive LOCALSKEY;
 3:  try
 4:    nonceA := mkNonce();
 5:    kb := key b;
 6:    send([ME; b; enc([nonceA; ME], kb)], lookup b);
 7:    r := receive PORT;
 8:    y := dec(r[2], mySK);
 9:    if (y[0] != nonceA or y[2] != b) then raise;
10:    send([ME; b; enc([y[1]], kb)], lookup b);
11:    send('Talking to ' + b, LOCAL);
12:  catch
13:    send('Error', LOCAL);
```

Figure 4: NSLPK protocol – sample Server implementation

of an annotated value might be

$$347 : \mathbf{enc}(717 : \mathbf{receive}\ a, 101 : \mathbf{key}\ 533)\ ,$$

indicating that the value 347 was computed by applying the method $\mathbf{enc}$ to a pair for which the first value is 717, which was in turn computed by evaluating $\mathbf{receive}\ a$ etc. Since the dependency rules make reference to "the last value returned by a given method invocation", some mechanism is needed to keep track of this information. So the configuration, or state of execution, of an annotated command c will be a triple $\langle c, \sigma, s\rangle$ where c is an (annotated) command, σ an (annotated) store, and s is a *context* mapping method invocations of the form $f(w)$ to values, the last value returned by evaluating the method f with (annotated) argument w. Transitions will have the general shape

$$\langle c, \sigma, s\rangle \xrightarrow{\alpha} \langle c', \sigma', s'\rangle$$

where α can be of the form $v := f\ w$, representing the invocation of method f with argument w and return value v.

6.2 Dependency Specifications

A dependency rule such as (1) will have the general shape

$$f\ e \leftarrow x_1 := f_1\, e_1,\ \ldots,\ x_n := f_n\, e_n\ \mathit{when}\ \psi$$

where ψ is a boolean condition and the e_i are expressions.

A *dependency specification* $P = \langle \mathcal{H}, \mathcal{A}\rangle$ puts together a set of dependency rules $\mathcal{A}$ and a set $\mathcal{H}$ of secret entry points (e.g. $\mathbf{receive}\ LOCALSKEY$). The dependency rules in an specification limit

$$
\begin{array}{lcl}
\mathbf{enc}([na; a], pkB) & \leftarrow & b := \mathbf{receive}\ LOCAL\,,\ pkB := \mathbf{key}\ b\,,\ na := \mathbf{mkNonce}() \\
\mathbf{send}([a; b; c], d) & \leftarrow & c := \mathbf{enc}([na, a], pkB) \\
\mathbf{dec}(x, y) & \leftarrow & y := \mathbf{receive}\ LOCALSKEY \\
\mathbf{enc}([nb], pkB) & \leftarrow & b := \mathbf{receive}\ LOCAL\,,\ pkB := \mathbf{key}\ b\,,\ na := \mathbf{mkNonce}()\,, \\
 & & m := \mathbf{dec}(x, y)\,,\ na' := m[0]\,,\ nb := m[1]\,,\ b' := m[2] \\
 & & when\ na = na' \wedge b = b' \\
\mathbf{send}([a; b; e], d) & \leftarrow & e := \mathbf{enc}([nb], pkB)
\end{array}
$$

Figure 5: NSLPK – Dependency Specification for the Initiator

how the secret information is to flow through any implementation, from the moment it is input, till it becomes observable by the execution of an API function.

A rule in the specification declares an action $\alpha = (v := f\ e)$ to be admissible if the conditions to the right of the arrow are satisfied. Informally, conjuncts of the form $x_i := f_i\ e_i$ are satisfied if variable x_i matches the last result returned by the API invocation $f_i\, e_i$. The boolean expression ψ is used to introduce conditions such as "$state = STATE\_INIT$." Its purpose is to allow for limited observable branching in the processing of secrets. We will then write

$$P, s \vdash \alpha\ \mathsf{ok}$$

where s is the context recording the history of API invocations.

Since in *NSLPK* authentication relies upon the confidentiality of nonces and secret keys (cf. Section 3), $\mathcal{H}$ must include both $\mathbf{mkNonce}()$ and $\mathbf{receive}\ LOCALSKEY$. The dependency specification for the Initiator is completed with the rules in Fig. 5. Notice how the fourth rule makes explicit the expected nonce check.

Two questions of interest arise: (1) Are all direct flows through some given applet admissible? And (2), are there any inadmissible indirect flows? The first property is easily formalized as a notion of *flow compatibility*, meaning that if there is a derivation $\langle q_0, \sigma_0, s_0 = \bot\rangle \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_n} \langle c_n, \sigma_n, s_n\rangle \xrightarrow{\alpha_{n+1}} \langle c_{n+1}, \sigma_{n+1}, s_{n+1}\rangle$ then $P, s_n \vdash \alpha_{n+1}$ ok. A flow compatible program should manipulate secrets just as stipulated by the protocol. To see why this is needed, consider a variation of the code in Fig. 4 where we have added the line

```
7.5:  r := enc([nonceA; mySK; b], MYPK);
```

and where MYPK is the Initiator's public key. Given that the value of r can only be decrypted using the initiator's secret key, the execution of this statement may be consider innocuous. But further examination shows that it results in the full leakage of the secret key. Fortunately,

$$P, s \not\vdash \mathbf{enc}([n_0 : \mathbf{mkNonce}(); k_0 : \mathbf{receive}\ LOCALSKEY; \ldots], MYPK)\ \mathsf{ok}$$

meaning that the modified implementation is not flow compatible.

6.3 Secret Permuters

To get at both direct and indirect flows the idea is to use dependency specifications to isolate a class of "system pertubators", permutations of secrets, which can be applied at the external interface to internally process an annotated secret, say n, as if it really had some other value $\not{n}$. If the pertubators adhere to the constraints given by the dependency specification, and if care is taken to undo only those secret permutations for actions at the external interface that are admissible, no change in observable behavior ought to ensue. That is, a change of observable behavior in the *perturbed system* indicates a direct or indirect flow of information that does not adhere to the dependency specification.

Technically, a *secret permuter* for a given dependency specification will be a mapping g of annotated values to annotated values, satisfying a number of constraints derived from the dependency specification at hand. Details of these constraints are given in the accompanying paper [15]. The secret permuter is extended to stores and contexts in the obvious way, and to actions using a condition along the lines

$$g(s, \alpha = v := f\ w) = \begin{cases} v' := f\ g(w) & \text{if } P, s \vdash \alpha\ \mathsf{ok} \\ \alpha & \text{otherwise} \end{cases}$$

That is, the action α can be permuted into an action α' which is almost identical to α except that, in case α is actually admissible in the current context (and only in that case) the choices of values for secrets (and values depending on secrets) may be changed.

6.4 Admissible Information Flow

Using secret permuters we can define a form of *perturbed command* $c[g]$, and the criterion for admissibility for a guarded command c will then be that

$$\langle c, \sigma, s\rangle \sim \langle g(c)[g^{-1}], g(\sigma), g(s)\rangle$$

where $\sim$ is the standard Park-Milner strong bisimulation equivalence.

This idea of characterizing absence of information flow in terms of invariance under perturbation, is actually quite standard in information flow theory, and is related, among others, to approaches based on process equivalences [10] and PER's [29].

6.5 Local Verification Conditions

Applying the definition of admissibility out of the box can be quite cumbersome, since it is tantamount to searching for, and checking, a bisimulation relation. In case the observational content of control branching statements (e.g. conditionals and exception throws) is allowed by the dependency rules, we can do much better, since only data-related properties need to be checked.

As before, we associate a set of secret permuters to a dependency specification. Each secret permuter g in this set can be extended naturally to an execution trace permuter mapping trace $\alpha_1\alpha_2\ldots$ to trace $g(\alpha_1)g(\alpha_2)\ldots$. If for each secret permuter in the set, every possible execution trace of a

program is mapped into a permuted trace visiting the same program points, in the same order, we say that the control flow of the program is not affected by admissible secret permutations, and that the program is *stable*.

Our main verification result states that if a program is stable and *flow compatible* (i.e. it only produces admissible actions), then it is admissible. The result is important because it provides justification to the verification strategy that was sketched for our case study in Section 5. Value declassifications, like that of the output buffer (*outBuf*) before invoking method *sendMessage*, correspond to the verification of *flow compatibility*; whereas program counter declassifications, due to branching or exceptions, are accounted by *stability*.

7 Verification of Admissibility in Practice

This section reports on our efforts to develop an automatic tool for the verification of Admissibility properties of a simple object-oriented language annotated with loop invariants and method specifications. In the following, we describe the syntax of the model language together with its non-standard axiomatic semantics. The main feature of the semantics is that it records and informs on the way each secret value was obtained. With this important aid, we are able to provide a first-order encoding of a set of sufficient conditions for Admissibility.

7.1 OGCL: An Object-Oriented Guarded-Command Language

We illustrate our verification technique using a simple but rather complete implementation language which we call OGCL. It extends the sequential imperative language IMP used in the previous section with guarded-commands and object-oriented features. OGCL is also a variant of ECSTATIC [20], a simple object-oriented language with an axiomatic semantics. This choice has allowed us to reuse and benefit from the approach to verification proposed by Leino et al. This approach has proved successful not only for ECSTATIC, but also in the implementation of verification tools like ESC/Modula-3 and ESC/Java [22].

An OGCL program is a set of declarations for types, data fields, methods and methods implementations. Table 2 shows a simple example program written in OGCL. In what follows, we give a quick description of these constructs, referring the interested reader to [20] for a more detailed explanation.

Types are divided into simple types ($\mathbf{bool}$, $\mathbf{int}$ and $\mathbf{nat}$) and object types. An object is either $\mathbf{nil}$ or a reference to a set of data fields and methods. For this reason, object equality is understood just as reference equality. OGCL programs define simple subtyping relations as orderings over type names. The primitive object type $\mathbf{obj}$ is a supertype of all types, and $\mathbf{nat}$ is a subtype of $\mathbf{int}$. The statement "$\mathbf{type}\ T <: U$" declares T as a subtype of (previously declared) type U. If U is not present, like in "$\mathbf{type}\ T$", then T becomes a direct subtype of $\mathbf{obj}$. OGCL's static type system guarantees that whenever an expression has static type T, then every value it may evaluate to at run-time is either $\mathbf{nil}$ or an object whose allocated type is a subtype of T.

A data field declaration "$\mathbf{field}\ x{:}T \to U$" introduces x as a field of all objects of type T. The

values stored in such a field are of type U and are accessed by means of a "selector expression" $x[t]$ where t evaluates to a non-nil object of type T. For those familiar with OO-languages like C++ and Java, a selector expression $x[t]$ corresponds to the expression $t.x$.

A method declaration has the form

method *formal-outs* $:= m($*formal-ins*$)$
requires Pre
modifies w
ensures $nPost$
except-ensures $xPost$

where *formal-ins* and *formal-outs* are sequences of bindings. Each binding $x : T$ declares a parameter x to be of type T. We say that this specification is given at type T_0 where T_0 is the type of the first binding in *formal-ins*. The method is specified using three predicate formulas, Pre, $nPost$ and $xPost$, and a list of selector expressions w. The free identifiers in $nPost$ and $xPost$ may include *initial-value fields*, which are marked with a 0 subscript and refer to the value of a field at the moment of method invocation. A method declaration lists the requirements placed over all its correct implementations: If any implementation of method m is invoked in a state satisfying Pre, then it either does not terminate, or it terminates normally in a state satisfying $nPost$, or exceptionally (i.e. with an unhandled exception) in a state satisfying $xPost$. Moreover, every correct implementation of m can only alter the contents of object fields listed in w, or of objects it creates afresh.

The fourth kind of declaration in OGCL corresponds to method implementations with the form

impl *formal-outs* $:= m($*formal-ins*$)$ **is** S

where S is a command (explained later). If $x : T_0$ is the first binding in *formal-ins* then this implementation corresponds to the specification of method m given at type U, where U is the closest supertype of T_0 with such a method.

7.1.1 Commands

The syntax of OGCL differs from ECSTATIC's in two main aspects. First, OGCL introduce **except-ensures** clauses allowing for the specification of exceptional behaviors. Then, OGCL uses a command set derived from Dijkstra's guarded command language (using partial commands, a generalization introduced by Nelson in [26])

The OGCL command set is the following:

Formulas	F	$::= \mathbf{true} \mid \mathbf{false} \mid p(e_1, \ldots, e_n) \mid e = e \mid e < e \mid e \mathbin{@=} e \mid$
		$F \wedge F \mid \neg F \mid \forall\, bindings.\, F \mid \exists\, bindings.\, F$
Expressions	E	$::= \mathbf{true} \mid \mathbf{false} \mid \mathbf{nil} \mid n \mid \mathbf{x} \mid E \wedge E \mid \neg E \mid E = E \mid E < E \mid$
		$E + E \mid E * E \mid E - E \mid x[E]$

Table 1: OGCL formulas and expressions

$x := E$	simple assignment
$x := \mathbf{new}(T)$	allocation
$x[E] := E'$	update
$\mathbf{assume}\ E$	partial command
$\mathbf{assert}\ F$	assertion
$\mathbf{raise}$	exception
$\mathbf{if}\ S_1 [\!] \ldots [\!] S_n\ \mathbf{fi}$	alternative statement
$S; S'$	composition
$\mathbf{do}_F\ S\ \mathbf{od}$	loop
$\mathbf{try}\ S\ \mathbf{catch}\ S'$	exception handler
$\mathbf{var}\ bindings\ \mathbf{in}\ S\ \mathbf{end}$	block statement
var-list $:= m($*expr-list*$)$	method invocation

OGCL borrows, without changes, ECSTATIC's syntax for simple assignments, allocations, updates, compositions, block statements and method invocations. The partial command (see [26] for a thorough description) blocks for as long as E does not evaluate to $\mathbf{true}$. Together with the alternative statement, partial commands replace ECSTATIC's conditionals. Moreover, they can easily express Dijkstra's guarded commands too (the guarded command "$c_1 \rightarrow S_1 [\!] c_2 \rightarrow S_2$" is written "$\mathbf{if}\ \mathbf{assume}\ c_1; S_1 \ [\!]\ \mathbf{assume}\ c_2; S_2\ \mathbf{fi}$" in OGCL). The command $\mathbf{assert}\ F$ causes the program to terminate wrongly whenever formula F does not hold at this point. The execution of $\mathbf{raise}$ causes the program to raise an exception, which is always unnamed. An exception can also be raised implicitly, e.g. by attempting the evaluation of an undefined expression. Finally, the loop construct incorporates an invariant formula F to help automate the verification process.

7.1.2 Expressions and Formulas

Table 1 presents the rather standard syntax for OGCL expressions (E) and formulas (F). Only predicate $\mathbin{@=}$ deserves some explanation, which is given in the next section.

The evaluation at run-time of an expression can result in either a value or, if undefined, in an exception, and has no other side-effect. Formulas include first order quantifiers and, unlike ECSTATIC, also arbitrary function and predicate symbols. Since all language expressions could be used as terms, OGCL permits formulas like "$p(x > 0)$" as well as "$y > z[t]$" (notice that in the first case $>$ is interpreted as a function symbol, while in the latter case, it represents a predicate).

7.2 Axiomatic Semantics

In ECSTATIC (like in most other programming languages provided with an axiomatic semantics) the semantics uses a direct representation of the values manipulated by legal programs. A value stored in an integer variable is represented, in the semantics, just as an integer value. Thus, a integer variable x is mapped into a variable $\hat{x}$ in the (first order) logic used to describe states. This has important consequences. For example, the predicate $\hat{x} = \hat{y}$ describes those program states where precisely the same value is stored in variables x and y. However, there are situations where one would like the semantics to keep some extra information for each value computed by a program. The definition of Admissibility, for example, requires that we annotate values with their history of computation. In previous presentations, these annotated values were given a syntactic form making this extra information explicit. An annotated value "$5 : read(in)$" corresponded to an integer 5 read from input channel in using method $read$.

In our axiomatic semantics for OGCL, logical variables range over "annotated program values." However, instead of relying on the syntactic shape of the annotation, we use specific function and predicate symbols to describe the information in an annotated value. For example, we use function $val()$ to recover the program value contained within an annotated value. Since the annotations of a value may depend on the annotations of other values, we use predicates to relate them. In this way, we could represent the statement "program variable x contains annotated value $5 : read(in)$" as $val(\hat{x}) = 5 \wedge read(ct(in), \hat{x})$. We use function $ct()$ to mark a value as being a constant included in the program.

Why not use the simpler $\hat{x} = 5 \wedge read(in, \hat{x})$? Simply because if $\hat{x} = \hat{y}$ then we could conclude that $read(in, \hat{x})$, even if the value of variable y was obtained from completely different sources. Similar confusions would occur between constants included in the source code and computed values.

7.2.1 Transforming Expressions and Formulas

As a first step toward the definition of an axiomatic semantics, we embed OGCL expressions and formulas into first-order logic. This is achieved with the definition of functions tr and pr. Function tr converts an expression into a first-order term over annotated values. Function pr produces a first-order predicate out of an OGCL formula. Notice that a comparison predicate (like $>$ and $=$) is understood to compare unannotated values. In order to compare annotated values for equality, we use instead predicate $@=$.

Function tr is defined over OGCL expressions as follows.

- For any constant c (i.e. **false**, **true**, **nil** , or a numeric constant), $tr(c) = ct(c)$
- For any variable v, $tr(v) = v$
- For any select expression, $tr(x[E]) = select(x, tr(E))$
- Boolean, arithmetic and comparison operators are transformed using special functions (notice that, unlike their versions in [20], these special functions operate on annotated values. See Section 7.2.4 for an example of how these functions are defined within the logic):

$$
\begin{array}{lcl}
tr(E_1 \wedge E_2) & = & and(tr(E_1), tr(E_2)) \\
tr(\neg E) & = & not(tr(E)) \\
tr(E_1 = E_2) & = & equal(tr(E_1), tr(E_2)) \\
tr(E_1 < E_2) & = & less(tr(E_1), tr(E_2)) \\
tr(E_1 + E_2) & = & plus(tr(E_1), tr(E_2))
\end{array}
$$

Function pr is defined over OGCL expressions as follows.

- For any constant predicate p (i.e. $\mathbf{false}$, and $\mathbf{true}$), $pr(p) = p$
- pr distributes over the operators $\neg$ and $\wedge$
- $pr(\mathbf{Q}\ bindings.\ F) = (\mathbf{Q}\ vars.\ types \Rightarrow pr(F))$, where $\mathbf{Q}$ is a first-order quantifier, $vars = Vars(bindings)$ and $types = Types(bindings)$, as defined in [20]
- Comparison predicates are transformed into the corresponding first-order predicates (c.f. function tr):

  $$
  \begin{array}{lcl}
  pr(E_1 = E_2) & = & (val(tr(E_1)) = val(tr(E_2))) \\
  pr(E_1 < E_2) & = & (val(tr(E_1)) < val(tr(E_2)))
  \end{array}
  $$

- All other predicate symbols p are meant to describe annotated values, therefore

  $pr(p(E_1, \ldots, E_n)) = p(tr(E_1), \ldots, tr(E_n))$

  The pre-defined predicate $@=$ compares two annotated values for equality and therefore needs a special translation rule

  $pr(E_1 @= E_2) = (tr(E_1) = tr(E_2))$

7.2.2 Undefined Expressions

Occasionally, some expressions are built using operators that are not defined over all possible arguments. For example, $x[E]$ is not defined if E evaluates to $\mathbf{nil}$. Under these circumstances, we would expect that the evaluation of an undefined expression results in an exception. Moreover, we would like to identify the conditions under which this can happen.

Our approach treats an undefined application of an operator as an uninterpreted function of its subexpressions. Other approaches are equally possible. Although this one solves the issue of undefined expressions within the logic, it does not directly help us detect when an exception should be raised. We achieve this, instead, with the aid of a meta-function (i.e. a function defined outside the logic level). This meta-function, $Defined(E)$, returns a predicate that identifies when expression E can be evaluated without generating an exception. It is defined inductively over OGCL expressions. It is worth noting that whether a expression is defined or not does depends only on the unannotated values that it operates with. This is reflected in the use of function val to translate language expressions into first-order logic. For example:

$$Defined(x[E]) = Defined(E) \wedge val(tr(E)) \neq \mathbf{nil}$$

7.2.3 Verification Conditions: Commands

Following Leino [21], we define the weakest liberal precondition of command S and predicates (on the post-state of S) N and X, as the predicate $wlp.S.(N, X)$. This predicate holds for those initial states from which S either loops forever, terminates normally in a state satisfying N or terminates exceptionally in a state that satisfies X.

The definition of $wlp.S.(N, X)$ presents no major difficulties for most commands, with the important exception of the iterative command. The standard approach consists in the definition of a verification condition generator vc satisfying

$$vc.S.(N, X) \Rightarrow wlp.S.(N, X)$$

and, at the same time, making use of the extra information provided by loop invariants to avoid the difficulties in the computation of weakest preconditions of iterative commands.

For some commands, the definition is straightforward:

$$\begin{aligned} vc.\mathbf{skip}.(N, X) &\equiv N \\ vc.\mathbf{raise}.(N, X) &\equiv X \end{aligned}$$

which indicates that **skip** always terminates normally, without altering the state, and that a **raise** always terminates in an exceptional state.

Other simple cases include compositions, alternative statements, exception handlers, partial commands and assertions:

$$\begin{aligned} vc.(S; S').(N, X) &\equiv vc.S.(vc.S'.(N, X), X) \\ vc.\mathbf{if}\ S_1 [\!] \ldots [\!] S_n\ \mathbf{fi}.(N, X) &\equiv vc.S_1.(N, X) \wedge \ldots\ vc.S_n.(N, X) \\ vc.\mathbf{try}\ S\ \mathbf{catch}\ S'.(N, X) &\equiv vc.S.(N, vc.S'.(N, X)) \\ vc.\mathbf{assume}\ E.(N, X) &\equiv \mathit{if}\ \mathit{Defined}(E)\ \mathit{then}\ \mathit{val}(\mathit{tr}(e)) \Rightarrow N\ \mathit{else}\ X \\ vc.\mathbf{assert}\ F.(N, X) &\equiv \mathit{pr}(F) \wedge N \end{aligned}$$

Note the use of the predicate $(\mathit{if}\ x\ \mathit{then}\ y\ \mathit{else}\ z)$ as an abbreviation for the first-order predicate $(x \wedge y) \vee (\neg x \wedge z)$. The violation of an assertion stops the program at once, thus the precondition produced by vc excludes this possibility.

Simple Assignment For a simple assignment $x := E$, the verification condition states that variable x gets not only its value but also its annotations from E

$$vc.x := E.(N, X) \equiv \mathit{if}\ \mathit{Defined}(E)\ \mathit{then}\ N[e/x]\ \mathit{else}\ X$$

where $e = \mathit{tr}(E)$.

Update The update command requires more care. The state of a data field x is encoded using two functions as in [20]: $\mathit{select}(x, t)$, which returns the value of field x at object t; and $\mathit{store}(x, t, v)$,

which returns a new data field that agrees with x everywhere but possible at object t, where it hold value v. These two functions are related by the axioms:

$$\begin{array}{l} \forall x, t, t', v.\ val(t) = val(t') \Rightarrow select(store(x, t', v), t) = v \\ \forall x, t, t', v.\ val(t) \neq val(t') \Rightarrow select(store(x, t', v), t) = select(x, t) \end{array} \tag{2}$$

Note how only the value of the object reference (i.e. $val(t)$) is needed to recover the contents of the data field at that object. That is, the annotations of t are not of use here. However, the values stored in a data field carry with them all their annotations.

$$\begin{array}{rcl} vc.x[E] := E'.(N, X) & \equiv & \mathit{if}\ \mathit{Defined}(E) \wedge \mathit{Defined}(E') \wedge val(e) \neq \mathbf{nil}\ \mathit{then} \\ & & \quad N[store(x, e, e')/x] \\ & & \mathit{else}\ X \end{array}$$

where $e = tr(E)$ and $e' = tr(E')$.

Allocation The verification condition associated to an object allocation statement is almost that in [20], though with a minor change due to the introduction of annotations. We show its definition here for completeness, referring the reader to Leino's work for details on the various predicates it relies upon:

$$\begin{array}{l} vc.x := \mathbf{new}(T).(N, X) \equiv \\ \quad \forall x, alloc'.\ (typecode(x) = tc\$T \wedge \underline{val(x)} \neq \mathbf{nil}\ \wedge FieldReset(W, x) \wedge \\ \qquad \neg isDecl(x, alloc) \wedge succeeds(alloc, alloc') \wedge \\ \qquad (\forall t.\ isDecl(x, alloc') \equiv isDecl(x, alloc) \vee t = x)) \Rightarrow R[alloc'/alloc] \end{array}$$

where W is the list of all data fields whose index type is a supertype of T. The resulting verification conditions says that **new** registers a non-nil object reference (x) with allocated type T and all fields properly initialized ($FieldReset(W, x)$) into a new allocation state ($alloc'$).

Block Statement The verification condition associated to a block statement coincides with that in ECSTATIC:

$$vc.\mathbf{var}\ bindings\ \mathbf{in}\ S\ \mathbf{end}.(N, X) \quad \equiv \quad \forall\ vars.\ reset \Rightarrow vc.S.(N, X)$$

where $vars = Vars(bindings)$ and $reset = Reset(bindings)$. Function $Reset$ was defined in [20] to encode the initial value of local variables, and $Vars$ extracts the list of variable names from a list of bindings.

Method Invocation To illustrate the extraction of verification conditions for method invocation, consider the following schematic method specification. Method m takes two in- and two out-parameters.

$$\begin{array}{l}\textbf{method}\ u0 : U0,\ u1 : U1\ := m(t0 : T0,\ t1 : T1) \\ \quad \textbf{requires}\ Pre \\ \quad \textbf{modifies}\ w \\ \quad \textbf{ensures}\ nPost \\ \quad \textbf{except-ensures}\ xPost\end{array}$$

The verification condition for method m is:

$$\begin{array}{l} vc.(v0,\ v1\ := m(E0,\ E1)).(N, X) \equiv \\ \quad \mathit{if}\ \neg \mathit{Defined}(E0) \vee \neg \mathit{Defined}(E1)\ \mathit{then}\ X \\ \quad \mathit{else} \\ \qquad \forall\, t0, t1.\ t0 = tr(E0) \wedge t1 = tr(E1) \Rightarrow pr(Pre) \wedge \\ \qquad\qquad \mathit{if}\ \mathit{val}(t0) = \textbf{nil}\ \ \mathit{then}\ X \\ \qquad\qquad \mathit{else} \\ \qquad\qquad\quad (\forall\, u0,\ u1,\ W,\ \mathit{alloc}.\ \mathit{types} \wedge \mathit{fieldtypes} \wedge \mathit{succeeds}(\mathit{alloc}, \mathit{alloc}_0) \wedge Q \Rightarrow \\ \qquad\qquad\qquad\qquad (pr(nPost) \Rightarrow N[u0,\ u1/v0,\ v1]) \wedge \\ \qquad\qquad\qquad\qquad (pr(xPost) \Rightarrow X))[W, \mathit{alloc}/W_0, \mathit{alloc}_0] \end{array}$$

where W is $\mathit{Fields}(w)$, W_0 is W with every identifier initial-valued, $\mathit{types} = \mathit{Types}(u0 :\ U0,\ u1 : U1)$, $\mathit{fieldtypes} = \mathit{FieldTypes}(W)$, and $Q = \mathit{PostCondContrib}(W, w)$, as described in [20, p. 28].

Loops By means of the Invariance Theorem, which approximates the weakest liberal precondition for a loop provided with an invariant, we can immediately derive a verification condition for our loops:

$$vc.\mathbf{do}_F\ S\ \mathbf{od}.(N, X) \equiv (inv\ \ \wedge \forall\, W.\ inv \Rightarrow vc.S.(inv, X) \wedge (\neg grd(S) \Rightarrow N))$$

where $inv = pr(F)$, W is the list of all fields and variables modified in the loop body S, and $grd(c)$ is the guard of command c. Intuitively, a guard corresponds to the initial states from which a partial command can execute to completion. Formally, $grd(c)$ is defined as $\neg wp.c.(\mathit{false}, \mathit{false})$, where $wp.c.(N, X)$ is the set of initial states on which the execution of c terminates normally in a state satisfying N, or exceptionally in a state satisfying X. Note that there are straightforward syntactic restrictions on loop bodies that greatly simplify the computation of guards.

This completes the definition of vc for all commands in the language.

7.2.4 Verification Conditions: Method Implementations

The verification condition for method invocation avoids considering the actual implementation of the invoked method. It relies instead upon the assumption that all method implementations comply

with their corresponding specifications. As usual, OGCL merely adapts ECSTATIC's approach to the verification of method implementations.

Given the implementation

$$\textbf{impl}\ \textit{formal-outs} := m(\textit{formal-ins})\ \textbf{is}\ S$$

corresponding to specification

$$\begin{array}{l}\textbf{method}\ \textit{formal-outs} := m(\textit{formal-ins})\\ \quad\textbf{requires}\ Pre\ \ \textbf{modifies}\ w\\ \quad\textbf{ensures}\ nPost\ \ \textbf{except-ensures}\ xPost\end{array}$$

then its verification condition has the form

$$BackgroundPred \wedge pr(Pre') \wedge Y_0 = Y \wedge val(self) \neq \textbf{nil} \Rightarrow vc.S.(nPost \wedge Q, nPost \wedge Q)$$

The precise details are to be found in [20]. The formulas Pre', $nPost'$ and $xPost'$ are obtained from Pre, $nPost$ and $xPost$ replacing *formal-outs'* and *formal-ins'* by their respective names in *formal-outs* and *formal-ins*. Predicate Q is the postcondition contribution resulting from the modifies list, and Y is the union of all fields in w and all fields explicitly modified by command S. As before, $alloc$ is the allocation state of all objects; and $self$ is the first name in *formal-ins'*.

The background predicate, $BackgroundPred$, collects several axioms on the different functions and predicates used in the encoding. These include the two rules relating functions $store$ and $select$ (2), the encoding of the subtyping relation, criteria for the consistency of allocation states, etc. The main difference with respect to the background predicate used in ECSTATIC is in the axiomatization of functions operating on annotated terms. To illustrate this last item, consider function $equal$ which compares annotated values for equality.

For $equal$, the background predicate contains the axioms

$$\begin{array}{l}\forall\ c, d.\ val(equal(c, d)) = true \vee val(equal(c, d)) = false\\ \forall\ c, d.\ (val(equal(c, d)) = true) \equiv (val(c) = val(d))\end{array} \tag{3}$$

Similar axioms are given for the functions introduced in the translation of OGCL expressions (see definition of function tr, Section 7.2.1).

This concludes our overview of the axiomatic semantics for OGCL, which is strongly based on that of ECSTATIC, but differs from it in the necessary handling of a different command set and, most importantly, the notion of annotated terms. These terms represent language expressions instead of concrete values, adding a level of indirection to access the latter (using function val).

7.3 Verification Conditions for Admissibility

We have previously introduced the notion of *Admissibility* [6, 11, 14], an information flow property meant to determine whether a system preserves the confidentiality properties of its specification. In contrast with most information flow properties in the literature, Admissibility does not prevent

in general the leakage of information but forces all leaks to respect a confidentiality policy. Such a policy identifies the secrets handed to the system by its environment, and establishes precise conditions under which these secrets may be leaked.

In this section we show, by example, how to encode sufficient conditions for the verification of Admissibility over implementations written in OGCL. Consider this (rather artificial) example. A trusted computing base provides programmers with a few types and interface methods written in OGCL. All data is communicated to and from the environment through objects of type *Channel*. The data thus exchanged is of type *int* or *Data*. Objects of type *Data* contain two data fields: an integer field *info* containing the actual piece of data, and a boolean field *public* which is *true* whenever the data may be declassified. Type *Data* possesses only one method, namely *process*, which in some obscure manner processes the data, returning a new data object and deciding whether the processed data may be made public (i.e. declassified). Without further method specifications, this could be written in OGCL as

```
type Data
field info : Data -> int
field public : Data -> bool

method e : Data := process(d : Data)
  ensures fresh(e)
```

For type *Channel*, we only care at the moment about two methods: *Readsecret* reads a *Data* object from the channel which it marks as non-public; and *writeInt* just outputs an integer on the channel. This is encoded in OGCL as

```
type Channel

method d : Data := readSecret(ch : Channel)
  ensures fresh(d) and public[d] = false

method writeInt(ch : Channel, n : int)
```

A Confidentiality Policy We want to test implementations using this interface for illegal information leakages. These implementations are allowed to invoke the interface methods at will, provided they only output secret data (obtained using method *readSecret*) if it has been previously processed and deemed public (by method *process*). According to the definition of confidentiality policies given in [11], we define the set of secret entries as $\mathcal{E} = \{readSecret\}$ and the set of critical entries as $\mathcal{C} = \{process\}$. Then, our confidentiality policy has the only clause:

$$writeInt(c, n) \leftarrow x := readSecret\ c' \ \wedge\ y := process\ x \ \wedge\ z = true \ \wedge\ n = info[y]$$

Admissibility The task of verifying an Admissibility property can be divided into two subtasks. In the first one, we annotate secret values and use these annotations to track direct information

leaks. In the second one, we rely on relabeling functions and bisimulation relations to exclude any remaining indirect leaks. Fortunately, there exist sufficient conditions to complete the second subtask without engaging in complicated proofs of bisimulations. Essentially, it suffices to prove that the flow of control is not altered by permutations of the secrets.

Direct Leaks In order to track direct leaks of information, we make use of the enhanced axiomatic semantics of OGCL. Basically, we need to make sure that secrets get properly annotated at entry points ($\mathcal{E}$) and that these annotations are carried along during computation. We achieve by modifying the postconditions of every method in $\mathcal{E} \cup \mathcal{C}$. We add the conjunct

$$m(\textit{formal-outs, formal-outs}, w)$$

at the *ensures* clause of every method declaration

$$\begin{array}{l}\textbf{method}\ \textit{formal-outs} := m(\textit{formal-ins}) \\ \quad \textbf{requires}\ Pre\ \ \textbf{modifies}\ w \\ \quad \textbf{ensures}\ nPost\ \ \textbf{except-ensures}\ xPost\end{array}$$

(Obs: Similar conjuncts should be added to *xPost*, but at the time of writing, we have not yet worked out a proper shape for those conjuncts).

For our example, where methods raise no exceptions, this simply requires changing the specification of methods *process* and *readSecret*:

```
method e : Data := process(d : Data)
  ensures fresh(e) and process(d, e)

method d : Data := readSecret(ch : Channel)
  ensures fresh(d) and public[d] = false and readSecret(ch, d)
```

Now, given that the annotations (i.e. the predicates introduced in the previous step) are carried around together with the values, we only need to insert an extra requirement at every method that outputs information to the environment. What is required is that the method invocation be considered admissible. For this particular purpose, we introduce a specification predicate, $adm()$.

In the example, we just modify the *requires* clause of the only outputting method,

```
method writeInt(ch : Channel, n : int)
  requires adm(ch, n)
```

Finally, we have to give meaning to this predicate. This is done by encoding the confidentiality policy into the *background predicate* (see Section 7.2.4). Since there is only one clause in our example policy, we extend *BackgroundPred* with the axiom:

$$\begin{array}{l}\forall\, x, y, c, c', i_x, p_x, i_y, i_x.\ (readSecret(c', x, i_x, p_x) \wedge \\ \qquad\qquad process(x, y, i_y, p_y) \wedge \\ \qquad\qquad val(p_y) = true) \\ \qquad\qquad \Rightarrow adm(c, i_y)\end{array} \tag{4}$$

Indirect Leaks As expected, illegal direct leaks can be discovered rather straightforwardly using the annotated semantics. In order to verify the absence of indirect leaks, however, we cannot simply play around with method specifications. The verification has to make sure that the flow of information is stable under permutation of secrets. Although there is no need to actually permute secret inputs, we still have to insert checks at every potentially branching point.

The main idea here is to define an specification predicate, $stable(e)$ which determines whether the values (that language expression e may evaluate to) vary or not, whenever the secret inputs are permuted according to the confidentiality policy.

In our running example,

$$\forall\, x, y, c', i_x, p_x, i_y, i_x.\ (readSecret(c', x, i_x, p_x)\ \wedge process(x, y, i_y, p_y)) \Rightarrow stable(p_y) \tag{5}$$

tells us that we are interested in permutations of the secrets that preserve the decisions made by method *process* regarding whether to allow the leakage or not of the processed data.

Naturally, we will have to extend the definition of predicate *stable* with several other axioms to make it useful. In order to deem stable all constant expressions, we have the axiom

$$\forall x.\ stable(ct(x)) \tag{6}$$

It is also useful to relate a stable boolean expression to its negation.

$$\forall x.\ (stable(x) \equiv stable(\neg x)) \tag{7}$$

We turn our attention to a piece of code, written in OGCL, and how to enforce stability of branching conditions. Consider the following method declaration and implementation

```
method main(self : obj)

impl main(self : obj) is
  var
    d : Data,
    ch : Channel,
    n : int
  in
    ch := new(Channel);
    d := readSecret(ch);
    n := 0;
    d := process(d);
    if
      assume public[d];
      writeInt(ch, info[d])
    []
      assume not(public[d]);
      writeInt(ch, n)
   fi
  end
```

In this simple example, we just have to concentrate on the two partial commands for possible perturbations of control flow under secret permutation. For each **assume** E, we insert an assertion of the form **assume** $stable(tr(E))$. Some care must be taken, however, not to insert these new statements in places where they might affect the guards of commands. In the code above, we are forced to make these insertions just before the alternative command. The resulting code, with the suggested modifications, looks like:

```
impl main(self : obj) is
  var
    d : Data,
    ch : Channel,
    n : int
  in
    ch := new(Channel);
    d := readSecret(ch);
    n := 0;
    d := process(d);
    assert public[d];
    assert not(public[d]);
    if
      assume public[d];
      writeInt(ch, info[d])
    []
      assume not(public[d]);
      writeInt(ch, n)
   fi
  end
```

Notice that we could drop the second assertion, as it is derivable from the first one using axiom (7). But this is just an optimization. Moreover, the use of code transformations should be taken just as an aid for initial experimentation. (We understand that an efficient implementor may instead prefer to modify the verification condition predicate transformer for the partial command. The advantages of this other approach will be investigated in the near future). For comparison, Tables 2 and 3 show the code of our main example program before and after being transformed.

Two wrong implementations Our tool is able to verify Admissibility for the implementation of the *main* method above. It can also detect direct leaks like in this version

```
impl main(self : obj) is
  var
    d : Data,
    e : Data,
    ch : Channel
  in
```

```
    ch := new(Channel);
    d := readSecret(ch);
    e := process(d);
    if
      assume public[e];
      writeInt(ch, info[d])
    []
      assume not(public[e]);
      writeInt(ch, 0)
   fi
  end
```

where unprocessed secret values are output (although this only happens when the data, once processed, is considered declassifiable).

The tool handles indirect leaks too. For example, it rejects this implementation of the *main* method which leaks information about the unprocessed secret:

```
impl main(self : obj) is
  var
    d : Data,
    e : Data,
    ch : Channel
  in
    ch := new(Channel);
    d := readSecret(ch);
    e := process(d);
    if
      assume public[e] and (info[d] = 0);
      writeInt(ch, info[e])
    []
      assume not(public[e]);
      writeInt(ch, 0)
   fi
  end
```

Finally, observe that there are many other ways in which the flow of control could be affected by changes in secret inputs. Exceptions could be turned on and off, and loops could take more or less iterations to terminate (or not terminate at all), etc. In most cases, the needed checks for stability can be inserted without problems.

7.4 Implementation

We have implemented a verification condition generator for OGCL, using approximately 7000 lines of OCaml code (including comments and approximately 4500 lines generated automatically by

the OCaml lexer and parser generators). It produces output for the Simplify theorem prover [7] which, once fed with the encoding of the confidentiality policy, will try to validate the verification conditions for every method in the OGCL code. The tool, although still in a debugging stage, can already be used as an experimentation bench for different encodings of policies and tracking of secret data. In the near future, we plan to automatize the introduction of verification assertions for admissibility and experiment with implementations of increasing complexity. Performance is expected to become an important issue then. There are several optimizations we can apply, like code transformations tailored at out particular verification condition generator (c.f. [9]) and alternative ways to handle term substitutions in the logic (c.f. [1]).

Our experience has contributed arguments for the feasibility of a practical verifier of admissibility properties. Moreover, we have studied how to extend our prototype tool to the complete Java Card language. With this objective, we have constructed a decompiler based on the Dava decompiler, developed at McGill University, to decompile JavaCard Virtual Machine code into the guarded command language used as input to our verification condition generator [3]. This approach would permit the verification of smart card applets whose source code is not available.

8 Related Work

Most studies of security and, in particular, confidentiality have remained essentially theoretical (see [28] for a recent survey). Besides the already mentioned Java Information Flow (Jif) tool, few practical schemes have been developed and applied to the full Java language with the purpose of verifying information properties.

While we have restricted our study to single Java Card applets, Bieber et al. have analyzed the verification of standard non-interference properties of multi-applet interactions [2]. Their PACAP tool [27] operates at the level of byte code. It inputs code for different applications intended to cooperate within the same smart card, and produces an abstract model that can then be verified using the SMV model checker. The actual security property to be verified is expressed as a traditional flow restriction over an MLS lattice.

A different tool for the verification of inter-applet interactions is described in [4]. In this case, the control flow graph of a set of applets is extracted from their source (Java Card) code, and then translated into a pushdown system for which the model-checking problem for Linear Temporal Logic (LTL) is decidable. Reasoning at the level of the control flow graph makes it possible to specify properties (in LTL) that imply, although quite indirectly, useful confidentiality properties. The control-flow model abstracts data and therefore lacks the information needed to reason about declassifications.

9 Conclusions

During the course of the project we have studied the verification of confidentiality of real mobile code. As a test architecture we have used Java Card smart cards, a choice that has presented several advantages. First, the concerns regarding the secrecy properties of smart card applications are

definitively real, turning this study into much more than a simple exercise. The relatively small size of the Java Card architecture has permitted the construction of the necessary models of the Java Card API (moreover, in some cases the model was readily available). Finally although Java Card is a cut-down version of the Java language it preserves most of the challenges that characterize the verification of object-oriented programs.

We have evaluated the state-of-the-art in Java Card verification. Our case study was developed with a simple idea in mind: to set up a real verification situation, and see how far we could get with existing analysis tools. Currently available tools have proved quite appropriate to verify MLS models of confidentiality for Java Card applets, but it is known that these models have serious practical limitations. In our case study we have actually encountered the "label creeping" problem, and we have argued that, in practice, it is absolutely critical to be able to model and justify declassification.

Admissible interference is our response to this challenge, as a theoretical model capable of accommodating declassifications of data and control. Our simple verification tool for admissibility is a first step toward the application of this theory in practical situations. More work will be needed to scale it up to the applications and languages used in the described case study. However, we have suggested, by means of our prototype tool, that such an enterprise could start by combining two existing technologies, each quite successful in its own domain: a label model and a weakest-precondition semantics.

References

[1] A. Appel, K. Swadi, and R. Virga. Efficient substitution in hoare logic expressions. In *Fourth International Workshop on Higher-Order Operational Techniques in Semantics (HOOTS 2000)*, volume 41 of *Electronic Notes in Theoretical Computer Science*, September 2000.

[2] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In *ESORICS*, pages 1–16, 2000.

[3] J. Borgström. Translation of Smart Card Applications for Formal Verification – From bytecode to a guarded-command language. Master's thesis, Laboratory of Electronic and Computer Systems, Royal Institute of Technology, Electrum 229. SE- 164 40 Kista, Sweden, December 2002.

[4] G. Chugunov, L.-Å. Fredlund, and D. Gurov. Model Checking of Multi-Applet JavaCard Applications. In *Proceedings of CARDIS'02*, 2002.

[5] Dallas Semiconductor Corporation. PKCS#11 Implementation for the Java-Powered IButton. In *URL: www.ibutton.com/PKCS/*.

[6] M. Dam and P. Giambiagi. Confidentiality for mobile code: The case of a simple payment protocol. In *Proceedings of 13th IEEE Computer Security Foundations Workshop*, pages 233–244, Cambridge, England, July 2000. IEEE.

[7] D. Detlefs, G. Nelson, and J. Saxe. The Simplify Theorem Prover. In *URL: http://research.compaq.com/SRC/esc/Simplify.html*.

[8] A. Myers et al. Jif: Java + information flow. In *URL: www.cs.cornell.edu/jif*.

[9] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proceedings of the 28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 193–205, London, UK, January 2001.

[10] R. Focardi and R. Gorrieri. A classification of security properties for process algebras. *Journal of Computer Security*, 3(1):5–33, 1995.

[11] P. Giambiagi. Secrecy for mobile implementations of security protocols. Technical Report T2001-19, Swedish Institute of Computer Science, Box 1263. SE–164 29 Kista, Sweden, October 2001. Available from ftp://ftp.sics.se/pub/SICS-reports/Reports/SICS-T–2001–19–SE.ps.Z.

[12] P. Giambiagi. Verification Condition Generation for Admissibility. Draft, August 2002.

[13] P. Giambiagi. Verification Condition Generation for Admissible Information Flow, April 2003. Slides available from http://www.sics.se/fdt/projects/mcs/slides/vcgen.ppt.

[14] P. Giambiagi and M. Dam. On the Secure Implementation of Security Protocols. In P. Degano, editor, *Programming Languages and Systems, 12th European Symposium on Programming, ESOP 2003*, number 2618 in LNCS, pages 144–158. Springer, April 2003.

[15] P. Giambiagi and M. Dam. On the Secure Implementation of Security Protocols. *Science of Computer Programming*, 2004. To appear.

[16] J.A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, CA, 1982.

[17] E. Hubbers, M. Oostdijk, and E. Poll. From Finite State Machines to Provably Correct JavaCard Applets. In *Workshop of IFIP WG 11.2 - Small Systems Security, SEC'2003*, Athens, May 2003.

[18] E. Hubbers, M. Oostdijk, and E. Poll. Implementing a Formally Verifiable Security Protocol in Java Card. In *Proceedings of SPC'2003, First International Conference on Security in Pervasive Computing*, Boppard, Germany, March 2003.

[19] B. Jacobs, M. Oostdijk, and M. Warnier. Source Code Verification of a Secure Payment Applet. *Journal of Logic and Algebraic Programming, Special Issue on Smart Cards*, 2003. To appear.

[20] K. M. R. Leino. Ecstatic: An object-oriented programming language with axiomatic semantics. In B. Pierce, editor, *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997.

[21] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, January 1995.

[22] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. Technical Report 1999–002, Compaq Systems Research Center, May 1999.

[23] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Margaria and Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *LNCS*, pages 147–166. Springer, 1996. Also in Software Concepts and Tools, 17:93-102, 1996.

[24] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 228–241, San Antonio, TX, January 1999. ACM.

[25] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 186–197, Oakland, CA, May 1998.

[26] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[27] P.Bieber, J.Cazin, A. El Marouani, P.Girard, J.-L.Lanet, V.Wiels, and G.Zanon. The PACAP prototype: a tool for detecting Java Card illegal flow. In *Java Card Forum 2000*, Cannes, France, September 2000.

[28] A. Sabelfeld and A. C. Myers. Language-Based Information-Flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[29] A. Sabelfeld and D. Sands. A PER model of secure information flow in sequential programs. In *Proceedings of the 8th European Symposium on Programming*, volume 1576 of *LNCS*, pages 40–58, Amsterdam, March 1999. Springer.

[30] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, number 2031 in LNCS, pages 299–312. Springer, 2001.

A Project Achievements

In the course of the project the following achievements have been made:

- We have proposed a new approach, admissible interference, to non-interference in the presence of information downgrading and cryptographic operations which is suitable for mobile code applications. In Pablo Giambiagi's lic. tech. thesis [11], we examine the basics of admissible interference, using a special-purpose process algebra. In the thesis, an unwinding theorem is obtained, and links to a The thesis also investigates an infrastructure showing how admissibility can be used in practice to ensure confidentiality aspects of mobile code. This reports on our experiences with Java, web browsers and proof-carrying code, and addresses subtleties in the definition of an adequate user interface.

- In the paper [14] (full version [15] selected for a special journal issue) extends the work of Giambiagi's licentiate thesis by showing how admissibility can be applied to a simple imperative scripting-like language based on Dijkstra's guarded commands. The language is given special annotated semantics to which admissibility and the unwinding theorem can be applied. A number of examples have been studied including a rudimentary payment protocol, a declassifier authorising clients to release confidential information, and a security protocol.

- A prototype verification condition generator has been developed which enforces admissibility for programs written in the guarded command language. This is reported in the note [12].

- Admissibility have been applied to JavaCard in several ways. We have applied admissibility to prove noninterference properties of a PKCS#11 public key signature applet publicly available from Dallas Semiconductors [5].

- We have constructed a decompiler based on the Dava decompiler, developed at McGill University, to decompile JavaCard Virtual Machine code into the guarded command language used as input to our verification condition generator [3]. The main case study studied within this activity has been the JavaCard implementation of the Needham-Schroeder-Lowe public key protocol also addressed in this final report.

- Influenced by the PKCS#11 case study it has emerged that the policy specification language introduced in [14] is not quite expressive enough to handle realistic applets. We are therefore working to replace the confidentiality policies introduced in [14] with more general *flow automata*, a variation on Schneider's security automata. The modifications to the verifier, corresponding to this new approach, are described in [13]. item In collaboration with Dr. Martin Strecker at Technical University Munich we have started work toward a static analysis for admissibility taking the form of a JCVM type system. At a first stage, we have investigated the use of automata to specify multilevel security policies (i.e. noninterference) and their relation with well-established type systems for noninterference.

- We have examined the general prospects for supporting confidentiality for JavaCard applets in practice using admissibility and other state of the art tools , in particular the Jif tool from Cornell and the ESC/Java tool. This is reported in the present final report.

B Source Code of the NSLClient Applet

```
import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

public class NSLClient extends Applet  {

    final static byte STATE_INIT = (byte) 0;
    final static byte STATE_WAITING = (byte) 1;
    final static byte STATE_CHALLENGING = (byte) 2;


    final static short KEY_LENGTH_BITS = KeyBuilder.LENGTH_RSA_512;

    final static short KEY_LENGTH = (short)(KEY_LENGTH_BITS/(short)8);

    // n byte nonces.
    // Should be 8, 16 or 24 to allow use as DES key.
    // KEY_LENGTH > 2*NONCE_LENGTH is required.
    final static short NONCE_LENGTH = (short)16;

    final static short MESSAGE_LENGTH = (short)((short)2 + KEY_LENGTH);


    private byte state;
    private byte[] inBuf;
    private byte[] outBuf, nonce;

    private byte appletID;
    private byte serverID;

    private final byte[] secret = {(byte)'S',
                                   (byte)'E',
                                   (byte)'C',
                                   (byte)'R',
                                   (byte)'E',
                                   (byte)'T',
                                   (byte)'.'};

    private Cipher enc;
    private Cipher dec, sCipher;

    private RSAPublicKey server_public_key;
    private DESKey temp_key;
    private RandomData rand;

    private RSAPrivateKey my_private_key;


    // Constructor, sets initial state.
    protected NSLClient(byte[] bArray, short bOffset, byte bLength)
        throws SystemException, CryptoException
    {
```

```
        state     = STATE_INIT;

        appletID = bArray[bOffset];
        serverID = bArray[bOffset+1];

        inBuf   = JCSystem.makeTransientByteArray
                        (MESSAGE_LENGTH, JCSystem.CLEAR_ON_DESELECT);
        outBuf = JCSystem.makeTransientByteArray
                    (MESSAGE_LENGTH, JCSystem.CLEAR_ON_DESELECT);
        nonce   = JCSystem.makeTransientByteArray
                    (NONCE_LENGTH, JCSystem.CLEAR_ON_DESELECT);

        my_private_key       = (RSAPrivateKey)KeyBuilder.buildKey
                                    (KeyBuilder.TYPE_RSA_PRIVATE,
                                     KEY_LENGTH_BITS, false);

        server_public_key = (RSAPublicKey)
                                  KeyBuilder.buildKey
                                     (KeyBuilder.TYPE_RSA_PUBLIC,
                                      KEY_LENGTH_BITS, false);

        // MISSING: KEY INITIALIZATION FOR RSA KEYS!

        temp_key = (DESKey)KeyBuilder.buildKey
             (KeyBuilder.TYPE_DES_TRANSIENT_DESELECT,
              KeyBuilder.LENGTH_DES3_2KEY, false);


        rand      = RandomData.getInstance
                        (RandomData.ALG_SECURE_RANDOM);

        dec = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        enc = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        sCipher = Cipher.getInstance(Cipher.ALG_DES_CBC_PKCS5,
                                          false);

        enc.init(server_public_key, Cipher.MODE_ENCRYPT);
        dec.init(my_private_key,          Cipher.MODE_DECRYPT);

    }

    // Installation routine
    public static void install(byte[] bArray, short bOffset, byte bLength)
        throws ISOException, SystemException
    {
        try{
            (new NSLClient(bArray, bOffset, bLength)).register();
        }
        catch (CryptoException e) {
            ISOException.throwIt(ISO7816.SW_UNKNOWN);
        }
    }

    // Reset on each new selection
    public boolean select() {
```

```
    state = STATE_INIT;
    return true;
}


// Main call-in point.
public void process(APDU apdu)
    throws ISOException, APDUException, CryptoException
{
    switch(state) {
    case STATE_INIT: sendId(apdu); break;
    case STATE_WAITING: sendChallenge(apdu); break;
    case STATE_CHALLENGING: sendSecret(apdu); break;
    default: state = STATE_INIT;
             ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
    }
}


// A -> S: A
private void sendId(APDU apdu)
    throws APDUException
{
    apdu.getBuffer()[0] = appletID;
    apdu.setOutgoingAndSend((short)0,(short)1);
    state = STATE_WAITING;
}


// S -> A: (S, A, {n, S}_Pub(A))
// A -> S: (A, S, {n, m, A}_Pub(S))
private void sendChallenge(APDU apdu)
    throws ISOException, CryptoException, APDUException
{
    state = STATE_INIT;

    getMessage(apdu);           // Get the challenge.

    // Check S and A in the clear
    if(inBuf[0] != serverID || inBuf[1] != appletID)
        ISOException.throwIt
            (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

    // Decrypt the encrypted part.
    dec.doFinal(inBuf, (short)2, KEY_LENGTH, outBuf, (short)2);

    // Check S in the encrypted part.
    if(outBuf[NONCE_LENGTH+2] != serverID)
        ISOException.throwIt
            (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

    // Build the encrypted part of response.
    rand.generateData(nonce, (short)0, NONCE_LENGTH);
    Util.arrayCopyNonAtomic(nonce, (short)0,
                            outBuf,
```

```
                                (short)(NONCE_LENGTH+(short)2),
                                NONCE_LENGTH);
        outBuf[NONCE_LENGTH*2+2] = outBuf[1];

        // Encrypt the encrypted part of response.
        enc.doFinal(outBuf, (short)2,
                    (short)(((short)2)*NONCE_LENGTH + (short)1),
                    outBuf, (short)2);

        // Fill in clear text part of response.
        outBuf[0] = appletID;
        outBuf[1] = serverID;

        // Send the response.
        sendMessage(apdu, outBuf, (short)0, MESSAGE_LENGTH);
        state = STATE_CHALLENGING;
    }


// S -> A:    (S, A, {m}_Pub(A))
// A -> S: {secret}_m
private void sendSecret(APDU apdu)
        throws ISOException, CryptoException, APDUException
    {
        state = STATE_INIT;
        getMessage(apdu);           // Get the challenge.

        // Check S and A in the clear
        if(inBuf[0] != serverID || inBuf[1] != appletID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Decrypt the encrypted part.
        dec.doFinal(inBuf, (short)2, KEY_LENGTH, outBuf, (short)0);

        // Compare the nonces.
        for(short i = 0; i < NONCE_LENGTH; i++)
            if(nonce[i] != outBuf[i])
                ISOException.throwIt
                    (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);


        // OK, we've authenticated.
        // Now send secret, encrypted under our nonce.
        temp_key.setKey(nonce, (short)0);
        sCipher.init(temp_key, Cipher.MODE_ENCRYPT);
        sCipher.doFinal(secret, (short)0, (short)secret.length,
                        outBuf, (short)2);

        // Fill in clear text part of response.
        outBuf[0] = appletID;
        outBuf[1] = serverID;

        // Send the secret.
        sendMessage(apdu, outBuf, (short)0, MESSAGE_LENGTH);
```

```
        state = STATE_INIT;
    }

    private void getMessage(APDU apdu)
        throws ISOException, APDUException
    {
        short nRead     = 0;
        short nReadNow  = apdu.setIncomingAndReceive();

        while (nReadNow > 0 &&
               nRead + nReadNow <= MESSAGE_LENGTH) {
            Util.arrayCopyNonAtomic(apdu.getBuffer(),
                                    ISO7816.OFFSET_CDATA,
                                    inBuf, nRead, nReadNow);
            nRead   += nReadNow;
            nReadNow = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
        }

        if (nRead + nReadNow != MESSAGE_LENGTH)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

    }

    private void sendMessage(APDU apdu, byte[] buf, short offset,
                             short len)
        throws APDUException
    {
        apdu.setOutgoing();
        apdu.setOutgoingLength(len);
        apdu.sendBytesLong(buf, offset, len);
    }
}
```

C Jif Annotation of the NSLClient Applet

```
import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

public class NSLClient[principal P] extends Applet[P] authority(P) {

    final static byte STATE_INIT = (byte) 0;
    final static byte STATE_WAITING = (byte) 1;
    final static byte STATE_CHALLENGING = (byte) 2;


    final static short KEY_LENGTH_BITS = KeyBuilder.LENGTH_RSA_512;

    /*final*/ static short KEY_LENGTH; //= (short)(KEY_LENGTH_BITS/(short)8);
    /* KEY_LENGTH should be final, but the initializer may thrown an
       Arithmetic expression, so it has to be given within the
       constructor.      However, when put in the constructor, JIF says
       that the initialization should be done before invoking the
       constructor, and that point of the implicit superclass
       constructor invocation is at the very beginning (because, it says,
       the superclass, javacard.framework.Applet is NOT trusted). Therefore,
       the initialization of this field cannot be put here, nor nowhere in
       the constructor. What's up?
    */

                        // n byte nonces.
                        // Should be 8, 16 or 24 to allow use as DES key.
                        // KEY_LENGTH > 2*NONCE_LENGTH is required.

    final static short NONCE_LENGTH = (short)16;

    final short MESSAGE_LENGTH = (short)((short)2 + KEY_LENGTH);


    private byte state;
    private byte[] inBuf;
    private byte{P:}[]{P:} outBuf, nonce;

    private byte    appletID;
    private byte    serverID;

    private final byte{P:}[]{P:} secret = {(byte)'S',
                                           (byte)'E',
                                           (byte)'C',
                                           (byte)'R',
                                           (byte)'E',
                                           (byte)'T',
                                           (byte)'.'};

    private Cipher enc;
    private Cipher{P:} dec, sCipher;
```

```
private RSAPrivateKey{P:} my_private_key;
private RSAPublicKey server_public_key;
private DESKey{P:} temp_key;

private RandomData{P:} rand;

// Constructor, sets initial state.
protected NSLClient{}(byte[]{} bArray, short bOffset, byte bLength):{P:}
    throws (SystemException, CryptoException)
{
    try {
        KEY_LENGTH = (short)(KEY_LENGTH_BITS/(short)8);
        state    = STATE_INIT;

        appletID = bArray[0];
        serverID = bArray[1];

        inBuf   = JCSystem.makeTransientByteArray
                    (MESSAGE_LENGTH, JCSystem.CLEAR_ON_DESELECT);
        outBuf = JCSystem.makeTransientByteArray
                    (MESSAGE_LENGTH, JCSystem.CLEAR_ON_DESELECT);
        nonce   = JCSystem.makeTransientByteArray
                    (NONCE_LENGTH, JCSystem.CLEAR_ON_DESELECT);


        my_private_key = (RSAPrivateKey)
            KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
                                KEY_LENGTH_BITS, false);

        server_public_key = (RSAPublicKey)
            KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,
                                KEY_LENGTH_BITS, false);

        // MISSING: KEY INITIALIZATION FOR RSA KEYS!

        temp_key = (DESKey)KeyBuilder.buildKey
            (KeyBuilder.TYPE_DES_TRANSIENT_DESELECT,
             KeyBuilder.LENGTH_DES3_2KEY, false);


        rand = RandomData.getInstance
                (RandomData.ALG_SECURE_RANDOM);

        dec = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        enc = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        sCipher = Cipher.getInstance(Cipher.ALG_DES_CBC_PKCS5,
                                     false);

        enc.init(server_public_key, Cipher.MODE_ENCRYPT);
        dec.init(my_private_key,          Cipher.MODE_DECRYPT);

    }
    catch (NullPointerException e) {/*@ unreachable */}
    catch (ArithmeticException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
```

```
    catch (ClassCastException e) {/*@ unreachable */}

}


// Installation routine
public static void install{}(byte{}[]{} bArray, short bOffset,
                             byte bLength):{P:}
    throws (ISOException, SystemException)
    where authority(P)
{
    try{
        (new NSLClient[P](bArray, bOffset, bLength)).register();
    }
    catch (CryptoException e) {
        ISOException.throwIt(ISO7816.SW_UNKNOWN);
    }
    catch (NullPointerException e){/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e){/*@ unreachable */}

}


// Reset on each new selection
public boolean select{}() {
    state = STATE_INIT;
    return true;
}


// Main call-in point.
public void process{}(APDU{} apdu):{P:}
    throws (APDUException, CryptoException, ISOException)
    where authority(P)
{
    switch(state) {
    case STATE_INIT: sendId(apdu); break;
    case STATE_WAITING: sendChallenge(apdu); break;
    case STATE_CHALLENGING: sendSecret(apdu); break;
    default: state = STATE_INIT;
             ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
    }
}


// A -> S: A
private void sendId{}(APDU{} apdu)
    throws (APDUException)
{
    try {
        apdu.getBuffer()[0] = appletID;
        apdu.setOutgoingAndSend((short)0,(short)1);
        state = STATE_WAITING;
    }
    catch (NullPointerException e) {/*@ unreachable */}
```

```
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
}

// S -> A: (S, A, {n, S}_Pub(A))
// A -> S: (A, S, {n, m, A}_Pub(S))
private void sendChallenge{}(APDU{} apdu):{P:}
    throws (CryptoException, APDUException, ISOException)
    where authority(P)
{
    try {
        state = STATE_INIT;

        getMessage(apdu);       // Get the challenge.

        // Check S and A in the clear
        if(inBuf[0] != serverID || inBuf[1] != appletID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Decrypt the encrypted part.
        dec.doFinal(inBuf, (short)2, KEY_LENGTH, outBuf, (short)2);

        // Check S in the encrypted part.
        if(outBuf[NONCE_LENGTH+2] != serverID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Build the encrypted part of response.
        rand.generateData(nonce, (short)0, NONCE_LENGTH);
        Util.arrayCopyNonAtomic(nonce, (short)0,
                                outBuf,
                                (short)(NONCE_LENGTH+(short)2),
                                NONCE_LENGTH);
        outBuf[NONCE_LENGTH*2+2] = outBuf[1];

        // Encrypt the encrypted part of response.
        enc.doFinal(outBuf, (short)2,
                    (short)(((short)2)*NONCE_LENGTH + (short)1),
                    outBuf, (short)2);

        // Fill in clear text part of response.
        outBuf[0] = appletID;
        outBuf[1] = serverID;

        // Send the response.
        declassify({}){
            sendMessage(apdu, declassify(outBuf, {}),
                        (short)0, MESSAGE_LENGTH);

            state = STATE_CHALLENGING;
        }
    }
    catch (NullPointerException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
}
```

```
// S -> A:   (S, A, {m}_Pub(A))
// A -> S: {secret}_m
private void sendSecret{}(APDU{} apdu):{P:}
    throws (CryptoException, APDUException, ISOException)
    where authority(P)
{
    try {
        state = STATE_INIT;
        getMessage(apdu);      // Get the challenge.

        // Check S and A in the clear
        if(inBuf[0] != serverID || inBuf[1] != appletID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Decrypt the encrypted part.
        dec.doFinal(inBuf, (short)2, KEY_LENGTH, outBuf, (short)0);

        // Compare the nonces.
        for(short i = 0; i < NONCE_LENGTH; i++)
            if(nonce[i] != outBuf[i])
                ISOException.throwIt
                    (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);


        // OK, we've authenticated.
        // Now send secret, encrypted under our nonce.
        temp_key.setKey(nonce, (short)0);
        sCipher.init(temp_key, Cipher.MODE_ENCRYPT);
        sCipher.doFinal(secret, (short)0, (short)secret.length,
                        outBuf, (short)2);

        // Fill in clear text part of response.
        outBuf[0] = appletID;
        outBuf[1] = serverID;

        // Send the secret.
        declassify({}){
            sendMessage(apdu, declassify(outBuf, {}),
                    (short)0, MESSAGE_LENGTH);
            state = STATE_INIT;
        }
    }
    catch (NullPointerException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
}

private void getMessage{}(APDU{} apdu)
    throws (APDUException, ISOException)
{
    try {
        short nRead      = 0;
        short nReadNow   = apdu.setIncomingAndReceive();
```

```
            while (nReadNow > 0 &&
                   nRead + nReadNow <= MESSAGE_LENGTH) {
                Util.arrayCopyNonAtomic(apdu.getBuffer(),
                                        ISO7816.OFFSET_CDATA,
                                        inBuf, nRead, nReadNow);
                nRead    += nReadNow;
                nReadNow = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
            }

            if (nRead + nReadNow != MESSAGE_LENGTH)
                ISOException.throwIt
                    (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
        catch (NullPointerException e) {/*@ unreachable */}
        catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
    }

    private void sendMessage{}(APDU{} apdu, byte{}[]{} buf, short{} offset,
                               short{} len)
        throws (APDUException)
    {
        try {
            apdu.setOutgoing();
            apdu.setOutgoingLength(len);
            apdu.sendBytesLong(buf, offset, len);
        }
        catch (NullPointerException e) {/*@ unreachable */}
    }
}
```

D ESC/Java Annotation of the NSLClient Applet

```
import javacard.framework.*;
import javacard.security.*;
import javacardx.crypto.*;

public class NSLClient extends Applet    {

    final static byte STATE_INIT = (byte) 0;
    final static byte STATE_WAITING = (byte) 1;
    final static byte STATE_CHALLENGING = (byte) 2;

    final static short KEY_LENGTH_BITS = KeyBuilder.LENGTH_RSA_512;

    final static short KEY_LENGTH = (short)(KEY_LENGTH_BITS/(short)8);

    // n byte nonces.
    // Should be 8, 16 or 24 to allow use as DES key.
    // KEY_LENGTH > 2*NONCE_LENGTH is required.
    final static short NONCE_LENGTH = (short)16;

    final static short MESSAGE_LENGTH = (short)((short)2 + KEY_LENGTH);


    private /*@ spec_public */ byte       state;
    private byte[] inBuf;
    private byte[] outBuf, nonce;

    private byte appletID;
    private byte serverID;

    private final byte[] secret = {(byte)'S',
                                   (byte)'E',
                                   (byte)'C',
                                   (byte)'R',
                                   (byte)'E',
                                   (byte)'T',
                                   (byte)'.'};

    private Cipher enc;
    private Cipher dec, sCipher;

    private RSAPublicKey server_public_key;
    private DESKey temp_key;
    private RandomData rand;

    private RSAPrivateKey my_private_key;


    /*@ invariant
      @       (state == STATE_INIT || state == STATE_WAITING ||
      @        state == STATE_CHALLENGING)
      @   && inBuf != null && inBuf.length == MESSAGE_LENGTH
      @   && outBuf != null && outBuf.length == MESSAGE_LENGTH
```

```
  @   && nonce != null && nonce.length == NONCE_LENGTH
  @   && dec != null
  @   && enc != null
  @   && rand != null
  @*/

// Constructor, sets initial state.
/*@ requires
  @       0 <= bOffset && bLength >= 2 && bArray != null
  @    && bOffset + bLength < bArray.length;
  @*/
protected NSLClient(byte[] bArray, short bOffset, byte bLength)
    throws SystemException, CryptoException
{
    try {
        state    = STATE_INIT;

        appletID = bArray[bOffset];
        serverID = bArray[bOffset+1];

        inBuf    = JCSystem.makeTransientByteArray
                     (MESSAGE_LENGTH, JCSystem.CLEAR_ON_DESELECT);
        outBuf   = JCSystem.makeTransientByteArray
                     (MESSAGE_LENGTH, JCSystem.CLEAR_ON_DESELECT);
        nonce    = JCSystem.makeTransientByteArray
                     (NONCE_LENGTH, JCSystem.CLEAR_ON_DESELECT);

        my_private_key = /*@ nowarn Cast */ (RSAPrivateKey)
            KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PRIVATE,
                                KEY_LENGTH_BITS, false);

        server_public_key = /*@ nowarn Cast */ (RSAPublicKey)
            KeyBuilder.buildKey(KeyBuilder.TYPE_RSA_PUBLIC,
                                KEY_LENGTH_BITS, false);

        // MISSING: KEY INITIALIZATION FOR RSA KEYS!

        temp_key = /*@ nowarn Cast */ (DESKey)
            KeyBuilder.buildKey
              (KeyBuilder.TYPE_DES_TRANSIENT_DESELECT,
               KeyBuilder.LENGTH_DES3_2KEY, false);


        rand = RandomData.getInstance
                 (RandomData.ALG_SECURE_RANDOM);

        dec = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        enc = Cipher.getInstance(Cipher.ALG_RSA_PKCS1, false);
        sCipher = Cipher.getInstance(Cipher.ALG_DES_CBC_PKCS5,
                                     false);

        enc.init(server_public_key, Cipher.MODE_ENCRYPT);
        dec.init(my_private_key,          Cipher.MODE_DECRYPT);
    }
    catch (NullPointerException e) {/*@ unreachable */}
```

```
    catch (ArithmeticException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
    catch (ClassCastException e) {/*@ unreachable */}
}

// Installation routine
/*@ requires
  @       0 <= bOffset && bLength >= 2 && bArray != null
  @    && bOffset + bLength < bArray.length
  @*/
public static void install(byte[] bArray, short bOffset, byte bLength)
    throws ISOException, SystemException
{
    try{
        (new NSLClient(bArray, bOffset, bLength)).register();
    }
    catch (CryptoException e) {
        ISOException.throwIt(ISO7816.SW_UNKNOWN);
    }
    catch (NullPointerException e){/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e){/*@ unreachable */}
}

// Reset on each new selection
/*@ also_ensures state == STATE_INIT;
  @ modifies state;
  @*/
public boolean select() {
    state = STATE_INIT;
    return true;
}

// Main call-in point.
/*@ requires apdu != null && apdu._APDU_state == 1;
  @ modifies
  @    state, apdu.buffer[*], inBuf[*], outBuf[*], nonce[*];
  @ exsures (Exception) state == STATE_INIT;
  @*/
public void process(APDU apdu)
    throws ISOException, APDUException, CryptoException
{
    switch(state) {
    case STATE_INIT: sendId(apdu); break;
    case STATE_WAITING: sendChallenge(apdu); break;
    case STATE_CHALLENGING: sendSecret(apdu); break;
    default: state = STATE_INIT;
             ISOException.throwIt(ISO7816.SW_FUNC_NOT_SUPPORTED);
    }
}

// A -> S: A
/*@ requires
  @    apdu != null && apdu._APDU_state == 1
  @    && state == STATE_INIT;
  @ modifies
```

```
  @     state, apdu.buffer[*];
  @  ensures
  @     state == STATE_WAITING;
  @  exsures (Exception) state == STATE_INIT;
  @*/
private void sendId(APDU apdu)
    throws APDUException
{
    try {
        apdu.getBuffer()[0] = appletID;
        apdu.setOutgoingAndSend((short)0,(short)1);
        state = STATE_WAITING;
    }
    catch (NullPointerException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
}

// S -> A: (S, A, {n, S}_Pub(A))
// A -> S: (A, S, {n, m, A}_Pub(S))
/*@ requires
  @     apdu != null && apdu._APDU_state == 1
  @     && state == STATE_WAITING;
  @  modifies
  @     state, apdu.buffer[*], inBuf[*], outBuf[*], nonce[*];
  @  ensures
  @     state == STATE_CHALLENGING;
  @  exsures (Exception) state == STATE_INIT;
  @*/
private void sendChallenge(APDU apdu)
    throws ISOException, CryptoException, APDUException
{
    try {
        state = STATE_INIT;

        getMessage(apdu);       // Get the challenge.

        // Check S and A in the clear
        if(inBuf[0] != serverID || inBuf[1] != appletID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Decrypt the encrypted part.
        dec.doFinal(inBuf, (short)2, KEY_LENGTH, outBuf, (short)2);

        // Check S in the encrypted part.
        if(outBuf[NONCE_LENGTH+2] != serverID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Build the encrypted part of response.
        rand.generateData(nonce, (short)0, NONCE_LENGTH);
        Util.arrayCopyNonAtomic(nonce, (short)0,
                                outBuf,
                                (short)(NONCE_LENGTH+(short)2),
                                NONCE_LENGTH);
```

```
        outBuf[NONCE_LENGTH*2+2] = outBuf[1];

        // Encrypt the encrypted part of response.
        enc.doFinal(outBuf, (short)2,
                    (short)(((short)2)*NONCE_LENGTH + (short)1),
                    outBuf, (short)2);

        // Fill in clear text part of response.
        outBuf[0] = appletID;
        outBuf[1] = serverID;

        // Send the response.
        //declassify({}){
        sendMessage(apdu, /*declassify(*/outBuf/*, {})*/,
                          (short)0, MESSAGE_LENGTH);

            state = STATE_CHALLENGING;
        //}
    }
    catch (NullPointerException e) {/*@ unreachable */}
    catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
}


// S -> A:    (S, A, {m}_Pub(A))
// A -> S: {secret}_m

/*@ requires
  @    apdu != null && apdu._APDU_state == 1
  @    && state == STATE_CHALLENGING;
  @ modifies
  @    state, outBuf[*], apdu.buffer[*], inBuf[*];
  @  ensures
  @     state == STATE_INIT;
  @  exsures (Exception) state == STATE_INIT;
  @*/
private void sendSecret(APDU apdu)
    throws ISOException, CryptoException, APDUException
{
    try {
        state = STATE_INIT;
        getMessage(apdu);      // Get the challenge.

        // Check S and A in the clear
        if(inBuf[0] != serverID || inBuf[1] != appletID)
            ISOException.throwIt
                (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

        // Decrypt the encrypted part.
        dec.doFinal(inBuf, (short)2, KEY_LENGTH, outBuf, (short)0);

        // Compare the nonces.
        for(short i = 0; i < NONCE_LENGTH; i++)
            if(nonce[i] != outBuf[i])
                ISOException.throwIt
```

```
                    (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);

            // OK, we've authenticated.
            // Now send secret, encrypted under our nonce.
            temp_key.setKey(nonce, (short)0);
            sCipher.init(temp_key, Cipher.MODE_ENCRYPT);
            sCipher.doFinal(secret, (short)0, (short)secret.length,
                            outBuf, (short)2);

            // Fill in clear text part of response.
            outBuf[0] = appletID;
            outBuf[1] = serverID;

            // Send the secret.
            //declassify({}){
            sendMessage(apdu, /*declassify(*/outBuf/*, {})*/,
                        (short)0, MESSAGE_LENGTH);
                state = STATE_INIT;
            //}
        }
        catch (NullPointerException e) {/*@ unreachable */}
        catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
    }

    /*@ requires
      @    apdu != null && apdu._APDU_state == 1;
      @ modifies
      @    inBuf[*], apdu.buffer[*]
      @*/
    private void getMessage(APDU apdu)
        throws ISOException, APDUException
    {
        try {
            short nRead     = 0;
            short nReadNow  = apdu.setIncomingAndReceive();

            while (nReadNow > 0 &&
                   nRead + nReadNow <= MESSAGE_LENGTH) {
                Util.arrayCopyNonAtomic(apdu.getBuffer(),
                                        ISO7816.OFFSET_CDATA,
                                        inBuf, nRead, nReadNow);
                nRead    += nReadNow;
                nReadNow = apdu.receiveBytes(ISO7816.OFFSET_CDATA);
            }

            if (nRead + nReadNow != MESSAGE_LENGTH)
                ISOException.throwIt
                    (ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
        }
        catch (NullPointerException e) {/*@ unreachable */}
        catch (ArrayIndexOutOfBoundsException e) {/*@ unreachable */}
    }

    /*@ requires
      @    apdu != null && apdu._APDU_state == 1
```

```
    @     && 0 <= len && len < 256 && 0 <= offset && buf != null
    @     && offset + len <= buf.length;
    @  modifies
    @     apdu.buffer[*]
    @*/
  private void sendMessage(APDU apdu, byte[] buf, short offset,
                           short len)
    throws APDUException
  {
    try {
      apdu.setOutgoing();
      apdu.setOutgoingLength(len);
      apdu.sendBytesLong(buf, offset, len);
    }
    catch (NullPointerException e) {/*@ unreachable */}
  }
}
```

```
type Data
field info : Data -> int
field public : Data -> bool

method e : Data := process(d : Data)
  ensures fresh(e)


/***/
type Channel

method d : Data := readSecret(ch : Channel)
  ensures fresh(d) and public[d] = false

method writeInt(ch : Channel, n : int)

/***/
method main(self : obj)

impl main(self : obj) is
  var
    d : Data,
    ch : Channel,
    n : int
  in
    ch := new(Channel);
    d := readSecret(ch);
    n := 0;
    d := process(d);
    if
      assume public[d];
      writeInt(ch, info[d])
    []
      assume not(public[d]);
      writeInt(ch, n)
   fi
  end
```

Table 2: OCGL example program

```
type Data
field info : Data -> int
field public : Data -> bool

method e : Data := process(d : Data)
  ensures fresh(e) and process(d, e)

/***/
type Channel

method d : Data := readSecret(ch : Channel)
  ensures fresh(d) and public[d] = false and readSecret(ch, d)

method writeInt(ch : Channel, n : int)
  requires adm(ch, n)

/***/
method main(self : obj)

impl main(self : obj) is
  var
    d : Data,
    ch : Channel,
    n : int
  in
    ch := new(Channel);
    d := readSecret(ch);
    n := 0;
    d := process(d);
    assert stable(public[d]);
    /* Obs: ``assert stable(not public[d])''
            is derivable using axiom (7) */
    if
      assume public[d];
      writeInt(ch, info[d])
    []
      assume not(public[d]);
      writeInt(ch, n)
   fi
  end
```

Table 3: OCGL example program adapted to the verification of Admissibility