Confidentiality for Mobile Code: The Case of a Simple Payment Protocol

Mads Dam Pablo Giambiagi
Dept. of Teleinformatics, KTH/IT
Electrum 204, S-164 40 Kista, Sweden; {mfd,pablo}@sics.se

Abstract

We propose an approach to support confidentiality for mobile implementations of security-sensitive protocols using Java/JVM. An applet which receives and passes on confidential information onto a public network has a rich set of direct and indirect channels available to it. The problem is to constrain applet behaviour to prevent those leakages that are unintended while preserving those that are specified in the protocol. We use an approach based on the idea of correlating changes in observable behaviour with changes in input. In the special case where no changes in (low) behaviour are possible we retrieve a version of noninterference. Mapping our approach to JVM a number of particular concerns need to be addressed, including the use of object libraries for IO, the use of labelling to track input/output of secrets, and the choice of proof strategy. We use the bisimulation proof technique. To provide user feedback we employ a variant of proof-carrying code to instrument a security assistant which will let users of an applet inquire about its security properties such as the destination of data input into different fields.

1. Introduction

In this paper we report on some recent experiments to support confidentiality for mobile implementations of security-sensitive protocols. The targeted applications are simple Java applets expected to implement protocol agents for e-commerce, information management, or, e.g., voting. Working on the assumption that the protocol itself is correct, we ask under what conditions an implementation of such a protocol can be relied upon to respect confidentiality of some particular item of information, possibly acquired through user input. General access control mechanisms such as those found in current security architectures for Java are inadequate to protect against the many types of covert channel which a hostile applet possesses to communicate confidential information to the outside world. So, some extra machinery is required to provide this protection.

One option is to submit the protocol implementation to a complete verification, to ensure that exactly and only the actions required by the protocol at any specific time are possible. Enough information would then be transmitted at run-time to allow receiving hosts to validate the verification, for instance using proof-carrying code [13]. Even if such a scheme could be implemented efficiently, which is unclear, such an approach seems unduly rigid and heavyhanded. The experiments we report on here aim to show that it is possible to strike a balance such that the critical confidentiality properties are handled adequately (to plug, as comprehensively as possible, covert channels) while leaving many implementation aspects which are independent of those confidentiality concerns to be decided upon by the implementor.

The basic idea is to accompany the applet with a specification which in formal and verifiable terms specifies the confidentiality properties enjoyed by the applet. Once the applet has been seen, by the receiving host, to satisfy the confidentiality specification, the specification is used to instantiate a security assistant which lets users inquire about security properties such as the destination of data which is input into the different fields.

Our intention with this paper is to show how such an approach can be realized for the case of a non-trivial but quite simple applet which is supposed to implement an e-payment protocol in the style of 1KP [2]. This requires attention to a rather wide variety of issues, ranging from basic principles (how is confidententiality specified, modelled, and verified), implementation aspects (how are the basic principles mapped to actual JVM bytecode, how are confidentiality properties processed by the participating entities), to usability aspects (how can the confidentiality properties be presented to users to empower them to make informed decisions). Evidently we are not able to cover all this ground comprehensively in the context of a single paper. Rather, our aim is to outline a promising new approach to confidentiality for mobile code, and to show that the array of problems lying between basic principles and realization us-

ing today's technology may well have sound and practical solutions.

The first stumbling block concerns the nature of confidentiality itself. The primary means of realizing confidentiality is encryption. But, in this context, encryption also poses some fundamental problems. One difficulty is information leakage in the sense that changes in plaintext cause changes in ciphertext (cf. [19]). Since ciphertexts are visible to intruders as bit strings, this bit structure can easily be exploited by a hostile applet to create a covert channel. Moreover, this can often be done without trusted parties in the exchange ever learning that a leak has taken place. Similar channels can be established using timing leaks (cf. [8]).

However, encryption is not the only source of information leakage. A security-sensitive protocol may explicitly require that certain secret information be made public, entirely or in part, by some of its agents. An example of this is the 1-bit payment clearance notification channel from a payment acquirer to a merchant, as described in Section 2 below. The value of the notification message depends on several confidential parameters (account number, PIN code, account balance, credit limit,...), so it represents an information leak. However, this leak is intended, indeed it is inherent in the application.

To address this issue we introduce a notion of *admissible information flow* which is based on the idea, very roughly, of correlating changes in system behaviour with changes in input. For instance, we may correlate changes in some secret parameter, say an account number, with corresponding changes in the actions involved in inputting the account number to an applet, and in forwarding encrypted information containing that account number to a merchant. As another example, in an MLS model, input stimuli may be in the form of high or low level actions, and low-level actions are required to be completely uncorrelated with high-level ones. Cast in this way, the resulting notion of admissibility turns out to coincide with a variant of Goguen-Meseguer noninterference [5] due to Focardi and Gorrieri [4].

Admissibility is cast, first, in terms of general labelled transition systems. Mapping this to JVM amounts to an operational semantics. While this subject is getting fairly well understood in general terms (cf. [15, 3]) we face two particular complications:

1. The use of library method calls, to initialize and use window objects such as textfields (from package java.awt), to encrypt/decrypt (from javax.crypto), to open and close streams (from java.io), and to write to sockets (from java.net).

2. The need for some form of object annotation to identify the precise entry points of secrets, and to track their passage through the heap as the applet computes.

We then turn to the proof strategy. Many simple applets in the style of the purchasing applet can be assumed to satisfy a data independence property, in the sense that there will be no branching of control flow dependent on confidential information. In Section 6 we show how the bisimulation proof technique [10] turns out to be convenient for exploiting this property.

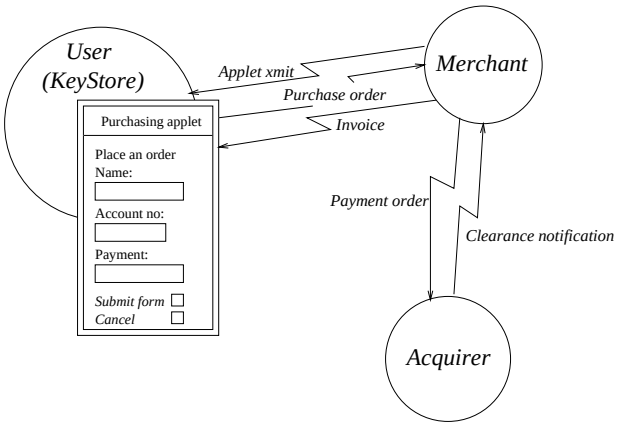


Figure 1. A purchasing applet

We also consider implementation and usability. Our experiments rely on the use of proof-carrying code (PCC). The idea in this context is to communicate, along with the confidentiality specification, a machine checkable proof that the bytecode adheres to this specification. This information can be validated at applet load time relying only on local information. We have built an experimental platform for PCC based on Sun's Java Plug-in, Netscape Navigator 4.5, and the Isabelle theorem prover [14]. In Section 7 we report very briefly on the status of these experiments and the role of the security assistant.

Finally, in the conclusion, we discuss related work and future directions. One problem with our approach as it applies to the case study is that timing leaks are not very adequately addressed. In the conclusion we discuss, however, how our modelling approach can be easily adapted to prevent most of them.

2. Confidentiality for a purchasing applet

Our running example is a simple purchasing applet based on the 1KP protocol [2], shown in Figure 1. An applet originating from *Merchant* is executed by *User* at her local site. The applet requests several items of information from *User*, including

1. the name *acq* of a payment acquirer,

2. ordering information *order* (item, price agreed, delivery, date and time, etc.), and

3. accounting information *acc* (account number, expiry date, PIN code), intended for *Acquirer*.

The user, or rather, the user's local system, possesses a *KeyStore* associating public keys to their principals. The applet requests from the *KeyStore* the key K corresponding to *acq*, and can then produce a purchase order

$$(acq, order, \{order, acc\}_K)$$

to pass to *Merchant*. The rest of the protocol is fairly self-evident from the figure: *Merchant* uses *acq* to route $\{order, acc\}_K$ (the payment order) to *Acquirer*, *Acquirer* returns a notification to *Merchant* telling whether or not the payment was cleared, and finally *Merchant* passes the goods along with an invoice to *User*.

Several items in this system may need protection:

(i) User's private information as known to the local host

(ii) Account information provided by *User* to the applet upon form submission

(iii) Order information similar to (ii)

(iv) The mere fact that *User* is engaging in a transaction with *Merchant*

(v) Secrets concerning *User*'s account possessed by *Acquirer* such as account balance or credit limit

Item (i) concerns applets gaining access to files or memory locations by some illicit means, by exploiting some shortcoming in *User*'s local system. This type of problem is outside the scope of this paper. Item (iv) has a nature which is somewhat different from the others. Simple applets which communicate using encryption over public channels in a straightforward fashion will in fact not meet this property. Nonetheless the confidentiality concern expressed, quite roughly, by item (iv) is a legitimate one which our techniques should at least be able to specify.

Our work here focuses on the confidentiality of item (ii), *User*'s account input. Intuitively it is fairly clear how an applet should behave in order not to breach the confidentiality of this information: It just has to follow the intentions of the protocol. Any substantial deviation from this behaviour can be exploited by a malicious applet to introduce covert channels. For instance:

1. The applet should not be allowed to retransmit in a manner which may depend on the value of *acc*.
2. The values of *order* and *acq* should not be modified in ways that may depend on the value of *acc*.
3. Correlation should be enforced between the value of K received from *User* and the value of K used for encryption.
4. The applet should not be permitted to e.g. encrypt information which depends on the value of *acc* other than as prescribed by the protocol.

Observe that constraints such as (2) and (3) are not required for integrity in this context. Integrity may be desirable, for sure, but that is a different matter. Rather, they are needed since a malicious applet could easily establish a covert channel by violating one of these constraints. Constraints such as (4) are required to prevent timing leaks with other active applets.

On the other hand, the applet *can* be allowed any other behaviour (like: authenticating K with *Merchant*, or downloading some fancy piece of graphics from somewhere) as long as that behaviour does not depend on the value of *acc*.

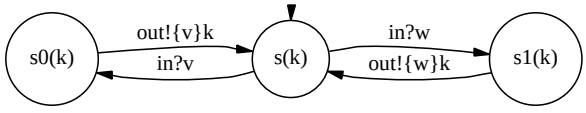
This type of confidentiality property is different and far more intensional than the MLS properties traditionally considered in the information flow literature (cf. [9]). There seems to be no useful way in which protocol entities or their actions can be assigned security levels. Neither is there a useful sense in which the role of the applet can be regarded as an information downgrader, for instance in the sense of intransitive information flow (cf. [17, 16]). Rather, the confidentiality properties of the applet rely on very intensional properties of applet behaviour which seem inherently to go beyond the MLS model.

3. Admissible information flow

We have found a simple way of modelling confidentiality, in the sense of (ii)–(v), in terms of permitted and prohibited dependencies. In this section we introduce and motivate this notion, using models based on general state transition systems.

A state transition system consists of *states* (processes), s, of which some are initial, *actions* (events, transition labels) $\alpha, \beta \in Act$, and *labelled state transitions* $s_1 \xrightarrow{\alpha} s_2$. The set *Act* may be structured in various ways. In particular we assume a "silent", or unobservable, action ε, and actions distinct from ε are often assumed to have the shape either $a!v$ or $a?v$ where, in $a!v$ (resp. $a?v$), v is a value sent to (resp. received by) the process named a.

Example 1 (Encrypter) *The following diagram corresponds to a system that reads a value (of some 2-element domain) from input channel* in*, and transmits it encrypted with key* k *along channel* out*:*

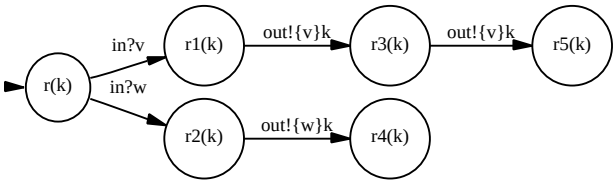


where the arrow indicates initial states (one for each value of k*).*

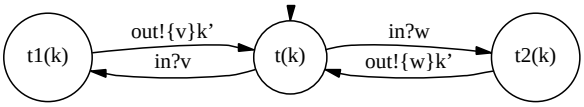
The previous system leaks information about its input only by encrypting it. Under the assumption of perfect cryptography, an observer of channel *out* cannot discover which value was input, provided that it is not able to decrypt $\{x\}_k$. The following two examples illustrate what a malicious implementation of the encrypter can do to leak extra information about the input.

Example 2 (Bad Encrypters)

Bad Encrypter #1: *This system encrypts its input before transmitting it, but an observer can still learn the input by counting the number of output messages:*



Bad Encrypter #2: *Another simple way of defeating confidentiality would be to use a wrong encryption key k':*



In all three versions of the *Encrypter*, output values or behaviours are affected by the values in the input. In Example 1, the information flow between *in* and *out* is the intended (admitted) one. Instead, in Example 2, we say that the systems contain non-admissible information flows.

Our intention is to capture admissible information flow by correlating changes in system behaviour with changes in input. For instance, in the case of noninterference based information flow properties (cf. [5, 6]) we require that system behaviour, as seen by a low-level observer, is unaffected by changes in high-level input. On the other hand, for the purchasing applet example, changes in the confidential parameter, say *acc*, received as part of a message

$$\mathit{applet}?(\mathit{acq}, \mathit{order}, \mathit{acc}) \tag{1}$$

should give rise to "proportional" changes in applet output messages

$$\mathit{merchant}!(\mathit{acq}, \mathit{order}, \{\mathit{order}, \mathit{acc}\}_K), \tag{2}$$

but affect applet behaviour in no other way. To make this idea precise the following ingredients are needed:

1. a notion of behaviour equivalence, and
2. a mechanism to model changes in values and their effects, as changes in actions.

Several equivalences have been considered in the information flow theory literature, including trace equivalence [5, 6], failures equivalence [16], and bisimulation equivalence [4]. The work reported here is to a large extent independent of the specific choice of equivalence, so it suffices for now to just assume the existence of *some* equivalence relation on states, $\sim$, reflecting the idea of behaviour identity, or indistinguishability by an external observer.

To model action changes we use the notion of a *relabelling* as a mapping $f : \mathit{Act} \to \mathit{Act}$. Relabellings are used to enforce the correlation between received and transmitted values required for confidentiality. A relabelling f applied to a transition system S determines a relabelled transition system $S[f]$ with states of the (syntactical) shape $s[f]$. The transition relation is determined by the condition: $s_1[f] \stackrel{\alpha'}{\to} s_2[f]$ iff for some α, $\alpha' = f(\alpha)$, and $s_1 \stackrel{\alpha}{\to} s_2$. The intention is that $S[f]$ maintains all the computational structure of S except for relabelling the transitions.

As an example in the context of the *Encrypter* systems, consider any relabelling $f : \mathit{Act} \to \mathit{Act}$ satisfying

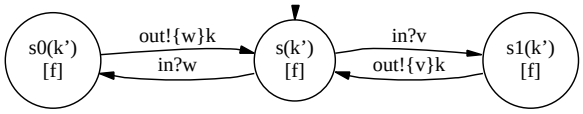
$$\begin{array}{lcl} f(in?v) & = & in?w \\ f(in?w) & = & in?v \\ f(out!\{v\}_k) & = & out!\{w\}_{k'} \\ f(out!\{w\}_k) & = & out!\{v\}_{k'} \\ f(\alpha) & = & \alpha, \text{ for all other actions.} \end{array}$$

where k and k' are some fixed key values.

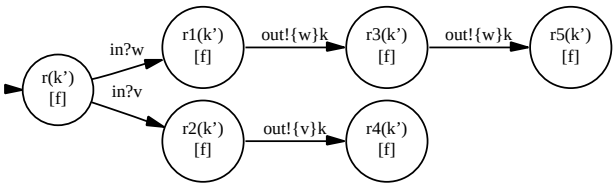
Notice how f permutes all inputs, but only the outputs that any good implementation of an *Encrypter* is expected to perform.

Example 3 (Relabelled systems)

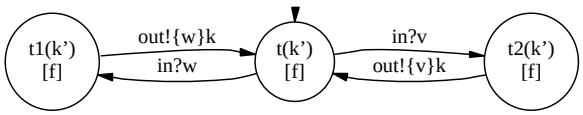
Encrypter: *If $s(k')$ is like in Example 1, we get the following transition system for $s(k')[f]$*



Bad Encrypter #1: *By considering $r(k')$ like in Example 2, the relabelled transition system $r(k')[f]$ is*



Bad Encrypter #2: *When $t(k')$ is like in Example 2, the relabelled transition system $t(k')[f]$ is*



By applying a given notion of behaviour equivalence, each Encrypter system can be compared against its relabelled version. We get the following results:

Encrypter:	$s(k')[f] \sim s(k)$
Bad Encrypter #1:	$r(k')[f] \not\sim r(k)$
Bad Encrypter #2:	$t(k')[f] \not\sim t(k)$

We can conclude that relabelling f, which intuitively permutes all inputs and only admissible outputs, lets us detect the presence of non-admissible information flows in both *Bad Encrypter* examples.

The proposal is therefore to cast confidentiality as invariance, up to state equivalence, under a set $\mathcal{F}$ of relabellings.

Definition 4 (Admissibility) Let $\mathcal{F}$ be a set of relabellings. The state s is *admissible for* $\mathcal{F}$ if $s[f] \sim s$ whenever $f \in \mathcal{F}$. The state transition system S is *admissible for* $\mathcal{F}$ if all its initial states are.

Choosing an appropriate set $\mathcal{F}$ is a matter of engineering, which will depend upon the particular protocol and confidentiality properties to be modelled. We consider the purchasing applet example, confidentiality of user account input (property (ii) of (i)–(v) in Section 2). The dependency of applet behaviour upon changes in the confidential parameter (in this case *acc*) is controlled by changing (relabelling) the action in which the confidential parameter is allowed to appear. So the input fetch event (1) may be relabelled to $\mathit{applet}?(\mathit{acq}, \mathit{order}, \mathit{acc}')$ for some specific pair of account number values *acc* and *acc*′. Correspondingly, the merchant output event (2) must be relabelled by replacing *acc* by *acc*′, and these will be the only action relabellings affecting the value of *acc*. With this choice of relabelling we are guaranteed some rather strong properties. For instance it is easy to show, from definition 4, that the only value *acc* which can appear as part of a merchant output event as in (2) is the corresponding value input as part of an applet input event (1).

This is, however, not yet the full story. We also need to correlate uses of K and *acq* with their points of definition, as otherwise a malicious applet might be able to encrypt using a key outside the control of the user. There is, however, a simple way of enforcing this correlation using relabellings, simply by systematically changing occurrences of acquirer names *acq* and public keys K, at their corresponding input actions, with other values *acq*′ and K'. This identifies explicitly the only admitted points of definition of K and *acq*. Since these values are not confidential, the relabellings should apply the same changes to all actions. As a result it will be possible for any well-behaved applet to e.g. authenticate K with *Merchant*.

We illustrate these ideas in Figure 2: It depicts, very schematically, part of the labelled transition system of some arbitrary purchasing applet. Each wiggling arrow stands for a sequence of transitions and is labelled by a single action of interest in the sequence, f is some relabelling function in $\mathcal{F}$, and the dotted lines correlate K and *acq* with their points of definition. Finally, $\mathit{leak}!(\mathit{acq}, K)$ indicates the presence of output actions depending on either *acq* or K.

For this particular confidentiality property we may thus define the set $\mathcal{F}$ as the set of all relabellings $f_{\vec{g}} : \mathit{Act} \to \mathit{Act}$ adhering to the scheme in Table 1 where $\vec{g} = (g_1, g_2, g_3)$ is a vector of endofunctions on, respectively, account information (the "account permuter", g_1), public keys (the "key permuter", g_2), and acquirer names (the "payment acquirer permuter", g_3). In the scheme, α matches all actions not listed before, meaning that each relabelling $f_{\vec{g}}$ changes the values of K and *acq* everywhere. Finally, since it is required that each function g_i in $\vec{g}$ satisfies $g_i = g_i^{-1}$, every relabelling in $\mathcal{F}$ is bijective and satisfies $f_{\vec{g}} = f_{\vec{g}}^{-1}$.

Notice that, for this definition of $\mathcal{F}$ to make sense we are really suppressing a lot of extra information to determine exactly which values *acc*, *acq*, and K to substitute. As we pass on to the JVM model in section 5, these annotations are made explicit.

4. Admissibility as confidentiality

With $\mathcal{F}$ as in Table 1, admissibility provides very tight control over the ways an applet is free to process the confidential parameter *acc*. For instance in the case where $\sim$ is strong bisimulation equivalence (ε is CCS τ [10]) there will, up to strong bisimulation equivalence, need to be a one-to-one correspondence between the states of an applet when receiving *acc* and when receiving *acc*′, for any pair $(\mathit{acc}, \mathit{acc}')$, when the appropriate relabelling function is applied. This leaves room for neither explicit, unauthorized leaks, implicit, say bitwise, leaks, or indeed of timing channels, since the internal computation steps of both applet instances need to be matched exactly. On the other hand, in the case of equivalences which abstract from internal computation (ε-steps), absence of timing channels can no longer be guaranteed.

A partial "purge" style characterization of admissibility (like in [5]) is easily obtained.

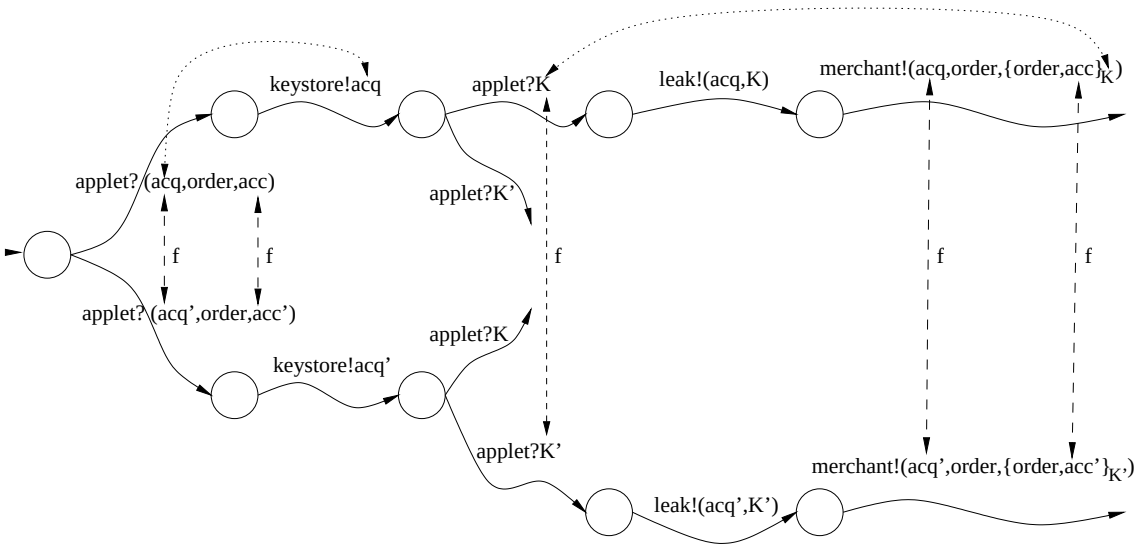


Figure 2. Schematic transition system for purchasing applet

Table 1. Denition of set $\mathcal{F}$ for purchasing applet

$applet?(acq, order, acc)$	$\leftrightarrow applet?(g_3(acq), order, g_1(acc))$
$merchant!(acq, order, \{order, acc\}_K)$	$\leftrightarrow merchant!(g_3(acq), order, \{order, g_1(acc)\}_{g_2(K)})$
$keystore!acq$	$\leftrightarrow keystore!g_3(acq)$
$applet?K$	$\leftrightarrow applet?g_2(K)$
α	$\leftrightarrow \alpha[g_2(K)/K, g_3(acq)/acq]$

Proposition 5 *Suppose that $\sim$ is finite trace equivalence and suppose that all members of $\mathcal{F}$ are bijections, and that their inverses also belong to $\mathcal{F}$. The state s is admissible for $\mathcal{F}$ iff for all $f \in \mathcal{F}$, if $\alpha_1\alpha_2\cdots\alpha_n$ is a trace of s then so is $f(\alpha_1)f(\alpha_2)\cdots f(\alpha_n)$.* □

In other words, given an admissible process for $\mathcal{F}$, permuting its inputs by $f \in \mathcal{F}$ makes its outputs also be permuted by f.

A similar observation is valid when we consider computations of applets in the context of arbitrary users and observers. We might model this using a slight variant of value-passing CCS, where we only need to recall the definitions of parallel composition and relabelling: The parallel composition $P \mid Q$ behaves like the interleaved behaviors of P and Q, allowing syncronization between dual actions, say, $a!v$ and $a?v$ and producing an internal action τ. The relabelled process $P[f]$ represents a relabelled state transition system exactly as defined in the previous section, with the additional requirement that f should map dual actions into dual actions, and τ into τ.

Using this notation our system might be modelled as a composite process

$$P = User \mid Applet \mid Merchant.$$

The best we can hope for, in view of the observability of encryption, seems to be that there is no observable difference between P and the version of P we would obtain by systematically relabelling the output of *User* and *Merchant* before feeding it into *Applet*, and then, subsequently, undoing this relabelling over all remaining communications, including all outputs by *Applet*. That is, we would want the following equation to hold:

$$P \sim (User[f_{\vec{g}}] \mid Applet \mid Merchant[f_{\vec{g}}])[f_{\vec{g}}] \qquad (3)$$

for any $\vec{g} = (g_1, g_2, g_3)$, where we have extended the definition of $f_{\vec{g}}$ so that it maps dual actions into dual actions. Notice that, since $f_{\vec{g}} = f_{\vec{g}}^{-1}$, applying a relabelling a second time undoes the effects of the first one. We would then have shown that the only possible ways *Applet* can react to

changes in its confidential input are undone by f. Hence *Applet* can not possess any unintended information leak.

By distributing the outermost relabelling [10, p. 80], the right hand side of equation (3) results structurally congruent to $(User[f_{\vec{g}}][f_{\vec{g}}] \mid Applet[f_{\vec{g}}] \mid Merchant[f_{\vec{g}}][f_{\vec{g}}])$. Since $f_{\vec{g}} \circ f_{\vec{g}} = id$, this is in turn congruent to $(User \mid Applet[f_{\vec{g}}] \mid Merchant)$ showing that equation (3) holds for arbitrary processes *User* and *Merchant*, and for an admissible *Applet* for all the three equivalences (traces, failures, bisimulation) mentioned above.

Other confidentiality properties among (ii)–(v) can be captured using admissibility as well. Property (iii), confidentiality of order information, is a straightforward analogue of (ii). Property (v) requires the notion of admissibility to be refined, to take into account both the existence of static secrets, and the 1-bit payment clearance leak. We do not consider this matter further in the present paper.

Property (iv), on the other hand, is interesting since it provides a link to more traditional information flow models. In this case we may, for simplicity, consider all actions of the protocol, Figure 1, as high, and we may assume that the protocol entities may be able to engage in other low actions which are not further specified. So we assume that the set of actions *Act* is partitioned into two subsets Act_H and Act_L of high-level and low-level actions respectively, and property (iv) then concerns the existence, or otherwise, of information flow from high to low in this model. We may, intuitively, expect to be able to cater for this idea using admissibility by requiring that no changes in behaviour at all be permitted by changes in high input, or input and output, depending on one's point of view. That is, we may consider admissibility for the set containing the single relabelling $f = \lambda\alpha \in Act_H.\varepsilon$ where the notion of equivalence is trace equivalence restricted to low-level actions. It turns out that this is just the notion of *strong nondeterministic noninterference*, a generalization of Goguen and Meseguer's notion of noninterference [5, 6] to nondeterministic systems by Gorrieri and Focardi [4] (similar generalizations have been defined for the CSP process algebra).

5. Modelling the Java Virtual Machine

Mapping admissibility to JVM amounts to giving it an operational semantics in terms of states and labelled state transitions. There is space only for an informal presentation here.

States are made up of the following three components:

1. *A Classfile Area* containing a definition for each class, its constant pool, fields and method codes.

2. *A Heap* containing a representation for each object or array.

3. *A Frame stack for each running thread* where each frame identifies the current executing method, the class it belongs to, the current value of the program counter, the values of local variables, and the operand stack.

To illustrate how the transitions of this state transition system are defined, consider a possible implementation of the purchasing applet: Table 2 delineates the part responsible for the applet side of the protocol where, for the sake of presentation, we have replaced the original JVM instructions by a less detailed and, hopefully, easier to read pseudocode. Each step involves a mix of primitive JVM instructions and library method calls. The library methods are implemented, in turn, by other methods and instructions, native or in bytecode form, belonging to JVM or the local operating system. Since our task is to analyze the confidentiality property of that part of the code which is mobile, it is natural to draw the trust borderline at the level of library method calls. Thus each library method invocation will, in the model, give rise to a "virtual instruction", a labelled transition representing the effect of the corresponding library method call. Two sorts of effects are involved. The effect on the external world (socket creation, input and output) is captured by transition labels (i.e., actions), and the effect on applet execution is captured by replacing the library objects by object "stubs" which maintain the required data structures. For example, a java.net.Socket stub contains a field of class java.net.InetAddress to keep track of the internet address the socket instance is connected to. Virtual instructions which have no externally observable effect are regarded as internal and labelled ε. Note though that by abstracting the execution of library methods (such as encryption), the model may not be able to detect some leaks which exploit side-effects and timing behaviours of the particular method implementations. The same can be said of leaks based on specific knowledge of the cryptographic algorithms used (we return to this issue in the conclusion). To avoid undeterminate states we assume that the applet has passed the bytecode verifier and all library methods are well-behaved, so that each instruction has a well defined successor state.

For instance, step 2, Table 2, is the input event $applet?(acq, order, acc)$ in the model of Section 3. An object of class NamedTextField is used to identify the entry point of each datum. Class NamedTextField, defined as a subclass of java.awt.TextField, associates a name to a textfield object. As an example, the name of the textfield intended for account number input may be, simply, "account number". Reading a string from a NamedTextField object is modelled as a virtual instruction labelled `GETTEXT textfield string`. This string is allocated in the heap and annotated with the name of the textfield, i.e. "account number". The string will keep its annotation during computation in the model, thus letting us identify where to apply the re-

Table 2. Partial pseudo-code for the purchasing applet

```
if (button ''Submit form'' was pressed) {                          (1)
    get (acq, order, acc) from applet window;                      (2)
    get K, the Acquirer's public key from local keystore;          (3)
    enc = encrypt {order, acc} with K;                             (4)
    data = convert (acq, order, enc) to proper type for output;    (5)
    create Socket connection with Merchant;                        (6)
    get OutputStream associated with Socket;                       (7)
    write data to OutputStream;                                    (8)
    close Socket;                                                  (9)
}
```

labelling (see end of Section 3), and calculate how value changes affect the state when performing the proof of admissibility (Section 6).

Similarly, in step 3, a keystore containing name-key pairs is accessed using some local method. This is represented as a virtual instruction labelled `GETKEY principal key`, with `key` the principal's key. This transition corresponds to events *keystore*$!$*acq* and *applet*$?$*K* in the model of Section 3.

Steps 4, 5 and 7 can be modelled as virtual instructions, but they do not involve any observable communication. Therefore, they are labelled with the silent action. On the other hand, step 6, socket creation, and step 9, socket closure, are clearly observable, so they are abstracted by appropriately labelled virtual instructions: `MAKESOCKET ipaddr portno` and `CLOSE socket`.

Step 8 is associated with virtual instruction `WRITE dest st`, and corresponds to the purchase order event $\mathit{merchant}!(\mathit{acq}, \mathit{order}, \{\mathit{order}, \mathit{acc}\}_K)$ in the model of the previous section. While the destination of the message (`dest`) can directly be recovered from the OutputStream object to which the method is applied, determining the value of `st` is more involved. The reason is that `st` should support a symbolic representation of data, sufficient to recover the values of *acq*, *order*, *acc* and *K*, as well as the operations performed on them to obtain the actual value which is transmitted to `dest`. This is done by annotating, in the model, every byte array in the heap, and extending the transitions that correspond to conversions from string, concatenation (pairing) and encryption so that the annotation is updated accordingly.

Table 3 contains a list of the virtual instructions used in each step of the pseudo-code, together with a reference to the corresponding event, if any, in the model of Section 3.

6. The proof technique

In order to apply admissibility to JVM applets, definition 4 has to be mapped to the semantics of the previous section. For the purchasing applet, the set of relabellings (Table 1) becomes the set of all $f_{\vec{g}}$ as defined in Table 4. A slight complication is that JVM applets define methods whose initial states might already contain secrets. This means that an initial state of $S[f]$ should be bisimilar to an initial state of S where the secrets appear permuted in a way that depends on f. In the following, $s(x, y, z)$ indicates state s with x substituted, in the heap, for every string object annotated "account number"", with y substituted for byte arrays annotated "key", and with z substituted for strings annotated "payment acquirer". Account numbers and acquirer names inherit their annotations from their textfield entry points. Keys, on the other hand, receive their annotation as result of being returned by a `GETKEY` virtual instruction.

The proof task, now, is to exhibit a proof of the equivalence

$$s_0(\mathit{acc}, K, \mathit{acq})[f_{\vec{g}}] \sim s_0(g_1(\mathit{acc}), g_2(K), g_3(\mathit{acq})) \quad (4)$$

where s_0 is any initial state of any method in the purchasing applet, and where $\vec{g}$ is any secret permuter vector. We use the bisimulation proof technique [10].

Definition 6 (Bisimulation Equivalence) *The binary relation R on states is a bisimulation relation if whenever $s_1 R s_2$ then*

1. *if $s_1 \xrightarrow{\alpha} s_1'$ then for some s_2', $s_2 \xrightarrow{\alpha} s_2'$ and $s_1' R s_2'$, and*
2. *vice versa, if $s_2 \xrightarrow{\alpha} s_2'$ then for some s_1', $s_1 \xrightarrow{\alpha} s_1'$ and $s_1' R s_2'$.*

A direct proof of (4), using the definition of $s[f_{\vec{g}}]$ for simplification, will thus have two components:

1. A specification of a relation $R_{\vec{g}} \in \mathit{jvm\_state} \times \mathit{jvm\_state}$ for each $\vec{g}$.
2. A proof that, for any $\vec{g}$, $R_{\vec{g}}$ satisfies the following conditions:
   (a) For every initial state s_0 of every method in the applet,

Table 3. Virtual instructions and corresponding events in the model of Section 3

Step	Virtual instruction label	Events in model
2	`GETTEXT textfield string`	*applet*?(*acq*,*order*,*acc*)
3	`GETKEY principal key`	*keystore*!*acq*, *applet*?*K*
4	ε	-
5	ε	-
6	`MAKESOCKET ipaddr portno`	-
7	ε	-
8	`WRITE dest st`	*merchant*!$(acq, order, \{order, acc\}_K)$
9	`CLOSE socket`	-

Table 4. Denition of $f_{\vec{g}}$ for a JVM purchasing applet

`GETTEXT 'account number'` acc	$\leftrightarrow$	`GETTEXT 'account number'` $g_1(acc)$
`GETTEXT 'payment acquirer'` acq	$\leftrightarrow$	`GETTEXT 'payment acquirer'` $g_3(acq)$
`WRITE merchant.com:4567` $(acq, order, \{order, acc\}_K)$	$\leftrightarrow$	`WRITE merchant.com:4567` $(g_3(acq), order, \{order, g_1(acc)\}_{g_2(K)})$
`GETKEY` acq K	$\leftrightarrow$	`GETKEY` $g_3(acq)$ $g_2(K)$
α	$\leftrightarrow$	$\alpha[g_2(K)/K, g_3(acq)/acq]$

$\forall acc, K, acq.$
$s_0(acc, K, acq)\ R_{\vec{g}}\ s_0(g_1(acc), g_2(K), g_3(acq))$

(b) $\forall r, t.\ r\ R_{\vec{g}}\ t \Rightarrow$
$(\forall r', \alpha.\ (r \xrightarrow{\alpha} r' \Rightarrow \exists t'.\ t \xrightarrow{f_{\vec{g}}(\alpha)} t' \wedge r'\ R_{\vec{g}}\ t') \wedge$
$\forall t', \beta.\ (t \xrightarrow{\beta} t' \Rightarrow \exists r', \alpha.\ \beta = f_{\vec{g}}(\alpha) \wedge r \xrightarrow{\alpha} r' \wedge$
$r'\ R_{\vec{g}}\ t'))$

There are several reasons why this is a convenient proof approach. First, bisimulation equivalence is finer than most other process equivalences around including trace, failures, and testing equivalence, so bisimulation equivalence is a sufficient condition for each of those. Secondly, the bisimulation technique requires reasoning only on the level of individual states and transitions, and, unlike these other equivalences, avoids quantification over entire computation histories, refusal sets, and so on. Thirdly, and most importantly, the bisimulation proof technique is convenient to exploit an important data independence property which we expect to be satisfied by most "reasonable" implementations of the purchasing applet. This property concerns the fact that, by avoiding inlining of basic operations on numbers and strings, the applet can be implemented without branching of control flow dependent on the value of confidential information. Under this condition a bisimulation relation $R_{\vec{g}}$, if it exists, for any fixed account-key-acquirer permuter triple $\vec{g}$ will have the shape

$$R_{\vec{g}} = \{(s(acc, K, acq), s(g_1(acc), g_2(K), g_3(acq))) \mid s(acc, K, acq) \in A\}$$

where A is some suitable invariant. We obtain the following result.

Proposition 7 *Suppose the following conditions hold:*

1. A is invariant: *Whenever* $s \in A$ *and* $s \xrightarrow{\alpha} s'$ *then* $s' \in A$.
2. A ensures data independence: *For all account-key-acquirer permuter triples* $\vec{g}$*, whenever* $s(acc, K, acq) \in A$ *and*
$$s(acc, K, acq) \xrightarrow{\alpha} s'(acc', K', acq')$$
then
$$s(g_1(acc), g_2(K), g_3(acq)) \xrightarrow{f_{\vec{g}}(\alpha)} s'(g_1(acc'), g_2(K'), g_3(acq')).$$
3. A respects annotations: *Whenever* $s(acc, K, acq) \in A$ *and*
$$s(acc, K, acq) \xrightarrow{\alpha} s'$$
for some s'*,* s' *can be written as* $s''(acc', K', acq')$ *for some* acc'*,* K'*, and* acq'.
4. A includes all initial states.

Then $R_{\vec{g}}$ *satisfies conditions 2.a and 2.b above.* □

Proposition 7 represents a considerable simplification over the direct proof strategy. The specific definition and check of a bisimulation relation is replaced by simple and local conditions which ensures the bisimulation property of some

fixed relation. Moreover, the quantification over account-key-acquirer permuter triples has been restricted to condition 7.2 which we are able, because of data independence, to establish independently of the choice of $\vec{g}$.

7. Experiments and user interface design

We have implemented an experimental platform for proof-carrying JVM applets based on Sun's Java Plug-in running inside Netscape Navigator 4.5. This platform permits us to experiment with concrete applets, equipped with proofs and specifications in the form of admissibility predicates. Proofs and specifications are produced using the Isabelle theorem prover [14], based on earlier work by Pusch [15]. The Isabelle formalization uses the ideas of Sections 5 and 6 rather directly. The formalization presents no essential problems. However, arriving at a good structure of the JVM specification and the proof which permits the checking speeds required for real applications is a matter of continued experimentation. For this reason we do not regard it very meaningful to report on performance aspects at this stage, but refer instead to the work of Necula and Lee (cf. [13, 12]) which has gone some way to indicate the practical realizability of the general PCC scheme.

A deeper problem, however, is how to present confidentiality information to users in a meaningful way. The point is that the nature of the confidentiality specifications considered here are quite different from the safety properties considered by Necula and Lee. There it is tacitly assumed that code producers and code consumers (here: users), have established agreement on which security policies to enforce in advance. This is inappropriate in cases such as this where the way the code gets hold of a piece of confidential information is not completely determined prior to loading time. We stress this point: In the case of the purchasing applet example, there is no apparent way to describe a security policy related to the flow of the PIN number other than in terms of how, where and when this piece of secret information enters and leaves the applet. In particular, no information asserting "this field is used for PIN number input" is to be trusted, unless this is adequately substantiated in terms of constraints on the way that information is used.

So instead of using PCC information as a code filter, weeding out from execution those applets that fail some statically determined property we use successful PCC checks to support a security assistant which will let applet users inquire about its security properties, such as the destination of data which is input into the different fields.

In order to realize this it is vital that the amount of specification information being communicated along with the applet is reduced to the greatest extent possible, by assuming prior agreement between code producers and code consumers of e.g. the formalization of JVM, and other concepts like bisimulation equivalence. In this way we can reduce specifications to the equivalents of (4) accompanied with Table 4 to define the relabelling set. Table 4 is used to initialize the security assistant.

If this table is simple enough, one could imagine an assistant providing help in a meaningful way by relying on some reasonable conventions as to the significance of the different actions appearing in Table 4. For instance, the assistant would know that `WRITE` actions represent output, that `GETTEXT` actions represent input, and that `GETKEY` is the action which locally associates a public key to a principal. Prompted by the user, for instance by pointing at a textfield named "account number", and maybe using some reserved mouse click sequence, the security assistant could then provide information of the following form:

> Information entered in this field is annotated "account number", and can only leave the system encrypted with the public key of "payment acquirer",

and when querying a textfield named "payment acquirer" the following response can be given:

> Information entered in this field is annotated "payment acquirer".

Our early experiments have indicated that this form of user support is both natural and helpful. However it does not scale well to complex confidentiality specifications. In those cases, one could consider an alternative assistant that would rely on an external authority to certify a high-level, human-readable description of the confidentiality property accompanying the applet. This authority would not have to certify the code itself, but just a description of the guarantees implied by the admissibility property. More work is needed, though, to determine what forms assistant output should take once we begin to address more complex applets and protocols.

8. Conclusion

We have introduced a new approach to confidentiality for mobile code which is capable of handling the sorts of explicit information leaks which arise in practical applications implementing protocol agents, for instance in e-commerce or voting. We have shown, furthermore, how our approach is mapped to real JVM code, and how an implementation using proof-carrying code can make use of the confidentiality specifications to support a security assistant which can help users determine, e.g., the destination of information which is input to some given text field. We report on our implementation work in a forthcoming paper.

The very intensional confidentiality properties that we focus on here go beyond other work on information flow

security which we have been able to identify. Myers [11] introduces a coarse-grained, but quite flexible approach to information flow control based on object labelling. However, no comprehensive protection against covert channels is provided. Abadi and Gordon [1] introduce a notion of bisimulation equivalence specifically geared to encryption. The approach, however, is very specific to the Spi calculus, and it is not clear how to extend it to other types of information leakage. Within the area of MLS security, the subject of intransitive noninterference, authorizing some distinguished downgrading process to leak information, has received some attention [6, 17, 16]. In the context of applet confidentiality, however, this approach seems to be of little help.

Our basic idea, of characterizing admissible and inadmissible flows by correlating changes in input to changes in behaviour, has clear precursors in earlier work on information flow security. The work of Abadi and Gordon clearly has this character, and we have already addressed (Section 4) noninterference and the work of Focardi and Gorrieri. For sequential languages, Sands and Sabelfeld [18] use partial equivalence relations for MLS information flow properties. Also here, though, there are fundamental difficulties in accomodating information leakages.

Avoiding the exploitation of timing channels is a difficult matter. An applet, for example, could use encryption or any operation which takes different times to compute on different inputs, and synchronization (e.g., with any other applet running concurrently in the system) to establish a covert timing channel. The approach presented here does not prevent these channels even if the synchronization with the concurrent applet is modelled. The reason is that most of the "virtual instructions" that can be used in these attacks are considered atomic and unobservable in the model. There is, however, a simple way of extending the model to plug most of these timing channels. It consists in making observable (though still atomic) all critical virtual instructions, like encryption. Then, by modifying accordingly the relabelling, an admissible applet will only be allowed to invoke those critical instructions with values which either do not depend on the secrets, or do so only as specified by the protocol. We believe this will provide a quite comprehensive protection against covert timing channels, though to quantify this will require a deeper analysis of the system primitives and of the cryptographic protocols involved.

Several issues need to be addressed in future work:

- Automation and semi-automation: In the paper we arrive at a simple and sufficient set of criteria for admissibility for data-independent applets. By exploiting an automated verification condition generator in the style of Necula and Lee [13] it is quite possible that the proof machinery can be simplified further. Moreover, by the use of static analysis techniques (like those in [20]), this machinery can possibly be eliminated altogether.
- Usability: The use of PCC information to provide interactive help seems promising. More experiments need to be done to evaluate what forms user feedback should take. Another issue is to determine good ways for programmers to define relabellings which reflect some desired confidentiality property.

Acknowledgements John Mullins, now at École Polytechnique de Montréal, participated actively in the initial development of this work. Stéphane Tao and Fabien Puig helped develop part of the experimental platform. Thanks are due to Jan Cederquist for useful comments and discussions, and also to Herbert Sander at Ericsson Utvecklings AB.

References

[1] M. Abadi and A. D. Gordon. A bisimulation method for cryptographic protocols. *Nordic Journal of Computing*, 5(4):267–303, 1998.

[2] M. Bellare, J. Garay, R. Hauser, A. Herzberg, H. Krawczyk, M. Steiner, G. Tsudik, and M. Waidner. iKP – a family of secure electronic payment protocols. In *First USENIX Workshop on Electronic Commerce*, May 1995.

[3] E. Börger and W. Schulte. A modular design for the Java Virtual Machine architecture. In E. Börger, editor, *Architecture Design and Validation Methods*. Springer, Dec. 1998.

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

[5] J. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the IEEE Symp. on Security and Privacy*, pages 11–20, Oakland, CA, 1982. IEEE Computer Society.

[6] J. Goguen and J. Meseguer. Inference control and unwinding. In *Proceedings of the IEEE Symp. on Security and Privacy*, pages 75–86, Oakland, CA, April 1984. IEEE Computer Society.

[7] IEEE. *Twelfth IEEE Computer Security Foundations Workshop*, Mordano, Italy, June 1999.

[8] P. C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In N. Koblitz, editor, *Advances in Cryptology – CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, 1996.

[9] J. McLean. The specification and modeling of computer security. *Computer*, 23(1):9–16, 1990.

[10] R. Milner. *Communication and concurrency*. Prentice-Hall, 1989.

[11] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of POPL'99*, pages 228–241. ACM, Jan. 1999.

[12] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *Proceedings of LICS'98*. IEEE, Computer Society Press, June 1998.

[13] G. C. Necula and P. Lee. Safe, untrusted agents using Proof-Carrying Code. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *LNCS*, pages 61–91. Springer, 1998.

[14] L. C. Paulson. Introduction to Isabelle. Technical report, Computer Laboratory, University of Cambridge, 1998.

[15] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.

[16] A. W. Roscoe and M. H. Goldsmith. What is intransitive noninterference? In *Proceedings of CSFW-12* [7], pages 228–238.

[17] J. Rushby. Noninterference, transitivity, and channel-control security policies. Technical Report CSL-92-2, Stanford Research Institute, 1992.

[18] 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, Mar. 1999. Springer.

[19] D. Volpano, M. Abadi, R. Focardi, C. Meadows, and J. Millen. Panel: Formalization and proof of secrecy properties. In *Proceedings of CSFW-12* [7].

[20] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, 1996.