A Hardest Attacker for Leaking References

René Rydhof Hansen

Informatics and Mathematical Modelling
Technical University of Denmark
DK-2800 Kongens Lyngby, Denmark
rrh@imm.dtu.dk

Abstract. Java Card is a variant of Java designed for use in smart cards and other systems with limited resources. Applets running on a smart card are protected from each other by the *applet firewall*, allowing communication only through *shared objects*. Security can be breached if a reference to a shared object is leaked to a hostile applet.
In this paper we develop a Control Flow Analysis for a small language based on Java Card, which will guarantee that sensitive object references can not be leaked to a particular (attack) applet. The analysis is used as a basis for formulating a *hardest attacker* that will expand the guarantee to cover *all possible* attackers.

1 Introduction

The Java Card platform is a multi-applet platform, meaning that a given Java Card may contain and execute several different applets from several different, possibly competing, vendors. In order for an applet to protect sensitive data from other, malicious, applets the Java Card platform implements an *applet firewall* to separate applets from each other by disallowing all communication, e.g., method invocation or field access, between applets. However, for certain applications, e.g., loyalty applets (applets implementing a customer loyalty scheme), it is desirable to allow (limited) communication between applets. Such communication is possible through the use of *shared objects* which allows an object with a reference to a shared object to invoke methods in the shared object. The problem with this approach is that it allows *any* object with such a reference to access the shared object. Thus it is necessary to ensure that certain objects references are not leaked to untrusted (attacker) applets. In [1] a static analysis is presented that for a given set of applets determines if a reference has been leaked. Such an approach works well provided the entire program is known *a priori*, which is not always possible because the Java Card framework allows dynamic loading of applets and therefore it is, in general, impossible to know the context in which an applet may run. This is a fundamental problem of any solution based on whole-program analysis. In this paper we solve the problem by developing a so-called *hardest attacker*, based on a control flow analysis [2] and inspired by the work on firewall validation in the ambient calculus [3]. By analysing a program in conjunction with the hardest attacker we can guarantee that object references are never leaked to *any* attacker, even if downloaded dynamically.

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 310–324, 2004.

For presentation purposes we develop our solution for a small bytecode language based on the Java Card Virtual Machine Language (JCVML). We have chosen to work at the level of the virtual machine because it is at this level applets are actually loaded onto the card. It also has the added advantage that there is no need to trust a compiler to not insert malicious code. This helps reducing the size of the *trusted computing base*. The remainder of the paper is structured as follows. Section 2 introduces the target language, Section 3 formalises the notion of leaking references. Section 4 presents the underlying control flow analysis and discusses a number of theoretical properties of the analysis. Finally in Section 5 the hardest attacker is described and discussed. Section 6 concludes and discusses future work.

2 The Carmel$_0$ Language

Carmel$_0$ is an abstraction of the Java Card Virtual Machine Language (JCVML). It abstracts away some of the implementation details, e.g., the constant pool, and all the features that are not essential to our development, e.g., static fields and static methods. The language is a subset of Carmel which is itself a rational reconstruction of the JCVML that retains the full expressive power of JCVML. See [4] for a specification and discussion of the full Carmel language.

The instruction set for Carmel$_0$ includes instructions for stack manipulation, local variables, object generation, field access, a simple conditional, and method invocation and return:

$$\begin{aligned}\mathsf{Instr} ::= {} & \texttt{push}\ c \mid \texttt{pop}\ n \mid \texttt{numop}\ op \mid \texttt{load}\ x \mid \texttt{store}\ x \mid \texttt{new}\ \sigma \mid \texttt{return} \\ \mid {} & \texttt{getfield}\ f \mid \texttt{putfield}\ f \mid \texttt{if}\ cmpOp\ \texttt{goto}\ pc_0 \mid \texttt{invokevirtual}\ m\end{aligned}$$

A Carmel$_0$ program, $P \in \mathsf{Program}$, is then defined to be the set of classes it defines: $P.classes$. Each class, $\sigma \in \mathsf{Class}$, contains a set of methods, $\sigma.methods$, and each method, $m \in \mathsf{Method}$, comprises a number of instance fields, $m.fields \subseteq \mathcal{P}(\mathsf{Field})$, and an instruction for each program counter, $pc \in \mathbb{N}_0$ in the method, $m.instructionAt(pc) \in \mathsf{Instr}$. For two programs P and Q such that $P.classes \cap Q.classes = \emptyset$ we shall write $P \mid Q$ for the obvious composition (concatenation) of the two programs.

We shall use a JCVML-like syntax for our examples, as shown in Figure 1. The example is a simplified version of a typical situation in Java Card applications: two applets (classes) wish to communicate (in the form of a method invocation from `Alice` to `Bob`). The method invocation is set up in `Alice.m_Alice` by first loading a reference to the object of the method to be invoked (line 0) which has been passed as a parameter to `m_Alice` into local variable number 1, and then the actual parameters are loaded from local variable 1 (line 1), in this case it is a (self-)reference to `Alice`, which by convention can be found in local variable 0. Finally the method is invoked (line 2). Note that the method to be invoked is identified by its class name, method name, and type signature (hence the `Object` in line 2 of `Alice.m_Alice`). The semantics is formalised in Section 2.1.

In real JCVML applets method invocation is the only form of communication that is allowed to cross the firewall boundary and only if the right object reference is known (in the example `Alice` must have a reference to `Bob`). As mentioned earlier, this can lead to problems if references are leaked. In the example program `Alice` wishes to communicate with `Bob` but also wants to make sure that `Bob` does not leak the `Alice`-reference to anyone else. In Section 3 we formalise the notion of leaking references.

```
class Alice {
  void m_Alice(Bob) {
    0: load 1
    1: load 0
    2: invokevirtual Bob.update(Object)
    3: return
  }
  /* ... */
}
```

```
class Bob {
  Object cache;

  void m_Bob() {
    0: return
  }
  void update(Object) {
    0: load 0
    1: load 1
    2: putfield Bob.cache
    /* ... */
    3: return
  }
}
```

Fig. 1. Example Carmel$_0$ Program P_{AB}

2.1 Semantics for Carmel$_0$

The semantics for Carmel$_0$ is a straightforward small step semantics, based on the corresponding semantics for full Carmel [4]. We briefly discuss the semantic domains before specifying the reduction rules.

A value is either a number or an object reference (a location): $\mathsf{Val} = \mathsf{Num} + \mathsf{ObjRef}$. The (global) heap is then a map from object references to objects: $\mathsf{Heap} = \mathsf{ObjRef} \to \mathsf{Object}$, where objects are simply maps from field names to values: $\mathsf{Object} = \mathsf{Field} \to \mathsf{Val}$. The operand stack is modelled as a sequence of values: $\mathsf{Stack} = \mathsf{Val}^*$ and the local heap (for a methods local variables) is then modelled as a map from (local) variables to values: $\mathsf{LocHeap} = \mathsf{Var} \to \mathsf{Val}$. Stack frames record the current method and program counter along with a local heap and an operand stack: $\mathsf{Frames} = \mathsf{Method} \times \mathbb{N}_0 \times \mathsf{LocHeap} \times \mathsf{Stack}$

With the semantics domains we can now specify the semantic configurations and the reduction rules. The configurations are on the form: $\langle H, F :: SF \rangle$ where $H \in \mathsf{Heap}$, $SF \in \mathsf{Frames}^*$, and $F = \langle m, pc, L, S \rangle \in \mathsf{Frames}$, meaning that the program is currently executing the instruction at program counter pc in method m with local heap L and operand stack S. This leads to reduction rules of

the following form for a given program, $P \in \mathsf{Program}$: $P \vdash \langle H, F :: SF\rangle \Longrightarrow \langle H', F' :: SF'\rangle$. In Figure 2 a few reduction rules are shown; for lack of space we only show the most interesting rules, the remaining rules are either trivial or obvious. In the reduction rule for `invokevirtual` (method invocation) we must take care to handle dynamic dispatch correctly. This is done as in JCVML by using a function, *methodLookup*, to represent the class hierarchy. It takes a method identifier, m', and a class, $o.class$, as parameters and returns the method, m_v, that implements the body of m', i.e., the latest definition of m' in the class hierarchy. In the same rule, note that a pointer to the object is passed to the object itself in local variable number 0.

$$\frac{m.instructionAt(pc) = \texttt{push}\ c}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L, c :: S\rangle :: SF\rangle}$$

$$\frac{m.instructionAt(pc) = \texttt{load}\ x}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L, L(x) :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{new}\ \sigma \wedge \\ loc \notin \mathrm{dom}(H) \wedge H' = H[loc \mapsto o] \wedge o = newObject(\sigma) \end{array}}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H', \langle m, pc+1, L, loc :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{putfield}\ f \wedge \\ H' = H[loc \mapsto o'] \wedge o' = H(loc)[f \mapsto v] \end{array}}{P \vdash \langle H, \langle m, pc, L, v :: loc :: S\rangle :: SF\rangle \Longrightarrow \langle H', \langle m, pc+1, L, S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.instructionAt(pc) = \texttt{invokevirtual}\ m' \wedge \\ S = v_1 :: \cdots :: v_{|m'|} :: loc :: S_0 \wedge m_v = methodLookup(m', o.class) \wedge \\ o = H(loc) \wedge L' = L[0 \mapsto loc, 1 \mapsto v_1, \ldots, |m'| \mapsto v_{|m'|}] \end{array}}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m_v, 0, L', \epsilon\rangle :: \langle m, pc, L, S\rangle :: SF\rangle}$$

Fig. 2. Semantic rules (excerpt)

In order to complete our discussion of the semantics, we need to define the initial configurations for a given program. In the JCVML applet execution is initiated by the run-time environment when the host sends the appropriate commands for installing and selecting an applet. The run-time environment then sets up an initial configuration with the appropriate method, applet instance, and parameters. We simplify this model by assuming that for a given program, $P \in \mathsf{Program}$, there exists an instance for each of the programs classes, $\sigma \in P.classes$, with a corresponding object reference, loc_σ, pointing to that instance, and a single entry point, m_σ.

In JCVML communication across the firewall boundary can only take place through so-called *shared objects*: When a JCVML applet wishes to communicate with another applet, it first has to obtain an object reference to such a shared object (if one exists) set up by the applet it wishes to communicate with. The

shared objects can be thought of as an *access control* mechanism where holders of a reference to a shared object are allowed to communicate with that object. References to shared objects are obtained by executing a trivial, if tedious, protocol involving the run-time system to make the actual transfer across the firewall. While there are no real technical difficulties in modelling this, it does not add any particular insights and instead we choose a simpler model: Any allowed sharing is set up at initialisation time. Thus, when defining a program we must also define what sharing is allowed to take place in that program. Formally we must define a function, $\mathit{sharing} : \mathsf{Program} \times \mathsf{Class} \to \mathsf{Class}^*$, that for each class in a program returns a list of classes (applets) it is allowed to access. During initialisation the shared references of a class are then passed to the corresponding entry point as in a method invocation (including a self-reference to the class in question) to the local variables of the entry point. Taking all of the above into account we can now define initial configurations for Carmel_0 programs:

Definition 1 (Initial Configurations). *If* $P \in \mathsf{Program}$ *then* C *is an* initial configuration *if and only if* $\sigma \in P.\mathit{classes}$, $P.\mathit{sharing}(\sigma) = \sigma_1 :: \cdots :: \sigma_n$, $\sigma_i \in P.\mathit{classes}$, *and* $C = \langle H, \langle m_\sigma, 0, [0 \mapsto \mathit{loc}_\sigma, 1 \mapsto \mathit{loc}_{\sigma_1}, \ldots, n \mapsto \mathit{loc}_{\sigma_n}], \epsilon\rangle :: \epsilon\rangle$

3 Leaking References

Intuitively we say that a class, τ, has been "leaked" to another class, σ, if there is an object of class σ or a method invocation in an object of class σ that contains, either in an instance field or in the local heap or on the operand stack, a reference to τ. In order to formalise this intuition, we shall write $v \Subset S$ for operand stacks $S \in \mathsf{Stack}$ if $S = v_1 :: \cdots :: v_n$ and $\exists i : v = v_i$. A similar notation is adopted for local heaps, $L \in \mathsf{LocHeap}$: we write $v \Subset L$ if $\exists x \in \mathrm{dom}(L) : L(x) = v$. We are now ready to formally define when a class has been leaked:

Definition 2 (Leaked). *Given a configuration,* $\langle H, SF\rangle$ *a class* τ *is said to be* leaked to class σ in $\langle H, SF\rangle$, *written* $\langle H, SF\rangle \vdash \tau \rightsquigarrow \sigma$, *iff* $\exists \mathit{loc}_\tau \exists \mathit{loc}_\sigma \exists f$: $(H(\mathit{loc}_\tau).\mathit{class} = \tau \ \wedge \ H(\mathit{loc}_\sigma).\mathit{class} = \sigma \ \wedge \ H(\mathit{loc}_\sigma).f = \mathit{loc}_\tau)$ *or* $\exists \mathit{loc}_\tau$ $\exists \langle m, pc, L, S\rangle \in SF$: $H(\mathit{loc}_\tau).\mathit{class} = \tau \wedge m.\mathit{class} = \sigma \wedge (\mathit{loc}_\tau \Subset S \vee \mathit{loc}_\tau \Subset L)$.

In order to be really useful, it is not enough to consider only single configurations. We therefore extend the notion of "leaked" to cover entire programs:

Definition 3 (Leaked). *A class* τ *is* leaked to σ in program P, *written* $P \vdash \tau \rightsquigarrow \sigma$, *if and only if there exists* C_0 *and* C *such that* C_0 *is an initial configuration of* P *and* $P \vdash C_0 \Longrightarrow^* C$ *such that* $C \vdash \tau \rightsquigarrow \sigma$.

We shall write $P \vdash \tau \not\rightsquigarrow \sigma$ to mean that class τ is *not* leaked to class σ in P.

As an example, consider the program in Figure 1 extended with the program in Figure 3, i.e., $P_{AB} \,|\, P_M$, and let P_{ABM} denote the entire program. Let us further assume that both `Alice` and `Mallet` are allowed to communicate with `Bob`, i.e., $P_{ABM}.\mathit{sharing}(\texttt{Alice}) = P_{ABM}.\mathit{sharing}(\texttt{Mallet}) = \{\texttt{Bob}\}$ Then, by executing the program it is easy to see that $P_{ABM} \vdash \texttt{Alice} \rightsquigarrow \texttt{Mallet}$, and

```
class Mallet {
  void m_Mallet(Bob) {
    0: load 1
    1: getfield Bob.cache
    2: return
  }
}
```

Fig. 3. "Malicious" program P_M.

```
class Charlie {
  void m_Charlie(Bob) {
    0: load 1
    1: invokevirtual Bob.m_Bob()
    2: return
  }
}
```

Fig. 4. "Innocuous" program P_C.

therefore `Alice` can be attacked by `Mallet`, simply because `Bob` "caches" the reference from `Alice`. Consider on the other hand the program $P_{ABC} = P_{AB} \mid P_C$ obtained by extending program P_{AB} with the program in Figure 4. It should be intuitively clear that `Alice` is not leaked to `Charlie` in P_{ABC}, because `Charlie` does not access fields in `Bob` that it is not supposed to. However in order to prove that we must try *all possible* executions of the program. In the following section we develop a control flow analysis capable of computing a conservative estimate of all possible executions. This is then used to prove that $P_{ABC} \vdash$ `Alice` $\not\rightsquigarrow$ `Charlie`. In [1] a conceptually similar approach is taken, for a slightly different subset of JCVML, with a special focus on the initial sharing and the firewall; to simplify presentation and analysis a three-address representation/conversion of JCVML is used and relevant API calls, in particular those for exchanging references to shared objects, are modelled as instructions. The approach in [1] is essentially a whole-program approach and thus the problem of dynamically downloaded applets is not considered.

4 Control Flow Analysis for Carmel$_0$

In this section we describe a control flow analysis for Carmel$_0$. The analysis is quite similar, in spirit, to the analysis developed in [5]. However, in anticipation of our intended use of the analysis, we develop a somewhat simpler (less precise) analysis, although with the added feature that it is parameterised on two equivalence relations: one on (object) classes, $\equiv_C \subseteq \mathsf{Class} \times \mathsf{Class}$, and one on methods, $\equiv_M \subseteq \mathsf{Method} \times \mathsf{Method}$. We let $[\cdot]_C : \mathsf{Class} \to \mathsf{Class}_{/\equiv_C}$ and $[\cdot]_M : \mathsf{Method} \to \mathsf{Method}_{/\equiv_M}$ denote the corresponding characteristic maps, that map a class or method respectively to its equivalence class. The reason for introducing these equivalence relations into the analysis is mainly that they allow us to partition the infinite number of names (for classes and methods) into a finite number of partitions and thereby enabling us to generate a finite Hardest Attacker (cf. Section 5 for more details). Furthermore, the partitioning can be used to "fine tune" the precision of the analysis: by choosing fewer equivalence classes the less precise (and less costly) the analysis and vice versa. Next we define the abstract domains for the analysis.

4.1 Abstract Domains

The abstract domains are simplified versions of the concrete domains used in the semantics. The simplification consists of removing information that is not pertinent to the analysis. Our abstract domain for representing object references in the analysis is similar to the notion of *class object graphs* from [6], thus object references should be modelled as classes. However, we have to take the equivalence relation over classes into consideration and therefore define an abstract object reference to be an equivalence class: $\widehat{\mathsf{ObjRef}} = \mathsf{Class}_{/\equiv_C}$ In order to enhance readability we write $[\text{Ref } \sigma]_C$, rather than just $[\sigma]_C$, for abstract object references. Following the semantics values are taken to be either numerical values or object references: $\mathsf{AbsVal} = \{\mathsf{INT}\} + \widehat{\mathsf{ObjRef}}$ and abstract values to be sets of such values: $\widehat{\mathsf{Val}} = \mathcal{P}(\mathsf{AbsVal})$. Since we are only interested in control flow analysis we do not try to track actual values (data flow analysis). Objects are modelled simply as an abstract value representing the (union of the values of all the) fields of the object: $\widehat{\mathsf{Object}} = \widehat{\mathsf{Val}}$. Addresses consist of a (fully qualified) method and a program counter, making them unique in a program: $\mathsf{Addr} = \mathsf{Method} \times \mathbb{N}_0$. The local heap tracks the values contained in the local variables of a given method: $\widehat{\mathsf{LocHeap}} = \mathsf{Method}_{/\equiv_M} \to \widehat{\mathsf{Val}}$. The operand stack is represented in a similar manner: for each method we track the set of values possibly on the stack: $\widehat{\mathsf{Stack}} = \mathsf{Method}_{/\equiv_M} \to \widehat{\mathsf{Val}}$. Finally we model the global heap as a map from object references to objects: $\widehat{\mathsf{Heap}} = \widehat{\mathsf{ObjRef}} \to \widehat{\mathsf{Object}}$.

4.2 Flow Logic Specification

An analysis is specified in the Flow Logic framework [7] by defining what it means for a proposed analysis result to be correct with respect to the analysed program. Thereby separating the specification of the analysis from the implementation of the analysis, making it easy to construct new analyses.

The judgements of our Flow Logic specification will have the following general form: $(\hat{H}, \hat{L}, \hat{S}) \models \texttt{addr} : \texttt{instr}$ where $\hat{H} \in \widehat{\mathsf{Heap}}$, $\hat{L} \in \widehat{\mathsf{LocHeap}}$, $\hat{S} \in \widehat{\mathsf{Stack}}$, $\texttt{addr} \in \mathsf{Addr}$, and $\texttt{instr} \in \mathsf{Instr}$. Intuitively the above judgement can be read as: $(\hat{H}, \hat{L}, \hat{S})$ is a *correct* analysis of the instruction `instr` found at address `addr`.

The full Flow Logic specification is given in Figure 5. We shall only explain a few of the judgements in detail. First the judgement for the `new` instruction. In the semantics two things happen: a new location is allocated in the heap, and a reference to the newly created object is placed on top of the stack. In the analysis only the last step is modelled: $\{[\text{Ref } \sigma]_C\} \subseteq \hat{S}([m_0]_M)$. This is because in the analysis objects are abstracted into the union of values stored in the objects instance fields, and since a newly created object has no values stored in its fields (we do not model static initialisation) it is essentially empty. Modelling that in the analysis we would write: $\emptyset \subseteq \hat{H}([\text{Ref } \sigma]_C)$ which trivially holds and thus does not contribute to the analysis. Note that the specification for the `pop`-instruction is trivially true. This is because the operand stack is (over-)approximated simply as the set of values that could possibly be on the stack

$$
\begin{array}{lll}
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{new}\ \sigma & \text{iff} & \{[\text{Ref}\ \sigma]_C\} \subseteq \hat{S}([m_0]_M) \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{load}\ x & \text{iff} & \hat{L}([m_0]_M) \subseteq \hat{S}([m_0]_M) \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{store}\ x & \text{iff} & \hat{S}([m_0]_M) \subseteq \hat{L}([m_0]_M) \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{push}\ c & \text{iff} & \{\mathsf{INT}\} \subseteq \hat{S}([m_0]_M) \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{pop}\ n & \text{iff} & \mathit{true} \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{numop} & \text{iff} & \mathit{true} \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{if}\ \mathit{cmpOp}\ \texttt{goto}\ pc & \text{iff} & \mathit{true} \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{getfield}\ f & \text{iff} & \forall [\text{Ref}\ \sigma]_C \in \hat{S}([m_0]_M) : \\
 & & \quad \hat{H}([\text{Ref}\ \sigma]_C) \subseteq \hat{S}([m_0]_M) \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{putfield}\ f & \text{iff} & \forall [\text{Ref}\ \sigma]_C \in \hat{S}([m_0]_M) : \\
 & & \quad \hat{S}([m_0]_M) \subseteq \hat{H}([\text{Ref}\ \sigma]_C) \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{return} & \text{iff} & \mathit{true} \\
(\hat{H},\hat{L},\hat{S}) \models (m_0,pc_0) : \texttt{invokevirtual}\ m & & \\
\quad \text{iff} \quad \forall [\text{Ref}\ \sigma]_C \in \hat{S}([m_0]_M) : & & \\
\quad\quad \forall [m_v]_M \in \mathit{methodLookup}_{/\equiv}(m, [\text{Ref}\ \sigma]_C) & & \\
\quad\quad\quad \hat{S}([m_0]_M) \subseteq \hat{L}([m_v]_M) & & \\
\quad\quad\quad [m_v]_M.\mathit{returnVal} \Rightarrow \hat{S}([m_v]_M) \subseteq \hat{S}([m_0]_M) & &
\end{array}
$$

Fig. 5. Flow Logic specification

anytime during execution of the program and therefore `pop` has no effect in the analysis. Similarly for the `if`, `numop`, and `return` instructions. A prerequisite for analysing the `invokevirtual` instruction is a model of the dynamic dispatch, i.e., we need an abstract version of the *methodLookup* function, taking the equivalence classes on methods and classes into account: $\mathit{methodLookup}_{/\equiv}(m, [\text{Ref}\ \sigma]_C) = \{[m_v]_M \,|\, m_v = \mathit{methodLookup}(m, \text{Ref}\ \sigma'), \text{Ref}\ \sigma' \in [\text{Ref}\ \sigma]_C\}$. This is not the only possible choice, but fits the purpose well and is trivially sound. However, this definition introduces a minor problem with methods that return a value: In order to determine if a given method returns a value it is enough to check if the methods return type is different from `void` ($m.\mathit{returnType} \neq \texttt{void}$). Taking equivalence classes into account, we have to approximate this by saying that the equivalence class of a method returns a value if any one of the methods in the equivalence class returns a value, thus $[m]_M.\mathit{returnVal} = \mathit{true}$ if $\exists m' \in [m]_M : m.\mathit{returnType} \neq \texttt{void}$ and $[m]_M.\mathit{returnVal} = \mathit{false}$ otherwise. The predicate defines a sound approximation, since it will compute an over-approximation of the possible flows. With these auxiliary predicates in place, the Flow Logic judgement for the `invokevirtual` instruction is straightforward. We first locate any object references on top of the stack: $\forall [\text{Ref}\ \sigma]_C \in \hat{S}([m_0]_M)$. The object references found are used to lookup the method to execute, modelled by a set of method equivalence classes: $\forall [m_v]_M \in \mathit{methodLookup}_{/\equiv}(m, [\text{Ref}\ \sigma]_C)$. The next step is then to transfer any arguments from the operand stack of the invoking method to the local variables of the invoked method: $\hat{S}([m_0]_M) \subseteq \hat{L}([m_v]_M)$.

Finally, if the method returns a value we make sure to copy that back to the invoking method: $[m_v]_M.returnVal \Rightarrow \hat{S}([m_v]_M) \subseteq \hat{S}([m_0]_M)$. The remainder of the judgements are created by following similar reasoning. Having specified how to analyse individual instructions, we lift this to cover entire programs in a straightforward way based on initial configurations in the semantics:

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}) \models P \qquad \text{iff} \\
\quad \forall (m, pc) \in P.addresses : \\
\qquad m.instructionAt(pc) = \texttt{instr} \Rightarrow (\hat{H}, \hat{L}, \hat{S}) \models (m, pc) : \texttt{instr} \\
\quad \forall \sigma \in P.classes : [\text{Ref } \sigma]_C \in \hat{L}([m_\sigma]_M) \\
\quad P.sharing(\sigma) = \sigma_1 :: \cdots :: \sigma_n \Rightarrow \{[\text{Ref } \sigma_1]_C, \ldots, [\text{Ref } \sigma_n]_C\} \subseteq \hat{L}([m_\sigma]_M)
\end{array}
$$

4.3 Theoretical Properties

In order to show the semantic soundness of the control flow analysis, we follow the approach of [8,5] and define *representation functions* for all the concrete domains. Intuitively a representation function maps a concrete semantics object, e.g., an integer, to its abstract representation in the corresponding abstract domain. In the case of integers this abstract representation is simply a token, INT, leading to the following representation function: $\beta_{\text{Num}}(n) = \{\mathsf{INT}\}$.

The notion of a location or an object reference only makes sense relative to a given heap, thus the representation function for locations is parameterised on the particular heap to use: $\beta^H_{\text{ObjRef}}(loc) = \{[\text{Ref } \sigma]_C\}$ if $H(loc).class = \sigma$. Combining the above two representation functions, that cover all of the basic values in a Carmel$_0$ program, we obtain a representation function for basic values: $\beta^H_{\text{Val}}(v) = \beta_{\text{Num}}(v)$ if $v \in \mathsf{Num}$ and $\beta^H_{\text{Val}}(v) = \beta_{\text{ObjRef}}(v)$ if $v \in \mathsf{ObjRef}$. Objects are represented simply as the "union" of all the values in all the objects fields, giving rise to the following representation function $\beta^H_{\text{Object}}(o) = \bigcup_{f \in \text{dom}(o)} \beta^H_{\text{Val}}(o.f)$. Finally we can define a representation function for heaps such that all objects of the same class are merged:

$$
\beta_{\text{Heap}}(H)([\text{Ref } \sigma]_C) = \bigcup_{\substack{loc \in \text{dom}(H) \\ \beta^H_{\text{Val}}(loc) = [\text{Ref } \sigma]_C}} \beta^H_{\text{Object}}(H(loc))
$$

The last step is to relate concrete semantic configurations to their abstract equivalents: $\langle H, SF\rangle \; \hat{\mathcal{R}}_{\text{Conf}} \; (\hat{H}, \hat{L}, \hat{S})$ iff $\beta_{\text{Heap}}(H) \sqsubseteq \hat{H} \wedge SF \; \hat{\mathcal{R}}^H_{\text{Frames}} \; (\hat{L}, \hat{S})$ where $(\langle m_1, pc_1, L_1, S_1\rangle :: \cdots :: \langle m_n, pc_n, L_n, S_n\rangle) \; \hat{\mathcal{R}}^H_{\text{Frames}} \; (\hat{L}, \hat{S})$ iff $\forall i \in \{1, \ldots, n\} : \beta_{\text{LocHeap}}(L_i) \sqsubseteq \hat{L}([m_i]_M) \wedge \beta_{\text{Stack}}(S_i) \sqsubseteq \hat{S}([m_i]_M)$. We are now in a position to state and prove that the control flow analysis is semantically sound. Following the tradition of Flow Logic specifications this is done by establishing a *subject reduction* property for the analysis, i.e., that the analysis is invariant under semantic reduction; this is very similar to the approach taken for type and effect systems. Below we take $\Longrightarrow^*$ to mean the reflexive and transitive closure of $\Longrightarrow$:

Theorem 1 (Soundness). *If* $P \in \mathsf{Program}$*,* C_0 *is an initial configuration of* P*,* $C_0 \Longrightarrow^* C$ *and* $(\hat{H}, \hat{L}, \hat{S}) \models P$*, then* $C\ \hat{\mathcal{R}}_{\mathrm{Conf}}\ (\hat{H}, \hat{L}, \hat{S}) \wedge (C \Longrightarrow C') \Rightarrow C'\ \hat{\mathcal{R}}_{\mathrm{Conf}}\ (\hat{H}, \hat{L}, \hat{S})$

Proof. By case-inspection using a technical lemma establishing that the call-stack is well-formed. ∎

While the above theorem establishes the semantic correctness for the analysis, it may not be entirely obvious how this is useful in the current setting. We therefore state a corollary below that follows as a trivial consequence of the above Theorem, showing that it is sufficient to check the analysis result in order to guarantee that no object references can be leaked:

Corollary 1. *Let* $P \in \mathsf{Program}$*,* $(\hat{H}, \hat{L}, \hat{S}) \models P$*. Assuming that* $[\text{Ref } \tau]_C \notin \hat{H}([\text{Ref } \sigma]_C)$ *and for all* $[m_v]_M$ *s.t.* $[\text{Ref } \sigma]_C \in [m_v]_M.\mathit{class}$ *implies* $[\text{Ref } \tau]_C \notin (\hat{S}([m_v]_M) \cup \hat{L}([m_v]_M))$ *then* $P \vdash \tau \not\rightsquigarrow \sigma$*.*

Note that the requirements above follow those of Definition 3 quite closely. This gives us a very convenient way to verify that there are no leaks in a given program by simply analysing the program and applying Corollary 1. The problem with this approach is of course that it requires access to the entire program which is not realistic in situations where users and third-party vendors are allowed to download applets onto a Java Card after it has been issued, as is the case for instance with newer mobile phones. In Section 5 we show how the analysis can be used to give strong security guarantees in precisely such situations.

Returning to the example programs P_{ABM} and P_{ABC} we wish to use the analysis to examine if `Alice` is leaked to `Mallet` and/or `Charlie` respectively. For simplicity we define the equivalence relations such that each class and method is in an equivalence class by itself. Analysing the programs, as described in Section 1, we find $(\hat{H}_{ABM}, \hat{L}_{ABM}, \hat{S}_{ABM})$ and $(\hat{H}_{ABC}, \hat{L}_{ABC}, \hat{S}_{ABC})$ such that $(\hat{H}_{ABM}, \hat{L}_{ABM}, \hat{S}_{ABM}) \models P_{ABM}$ and also $(\hat{H}_{ABC}, \hat{L}_{ABC}, \hat{S}_{ABC}) \models P_{ABC}$ respectively. Below the analysis results for the classes `Mallet` and `Charlie` are shown. We elide the corresponding results for `Alice` and `Bob` since they are of little consequence here.

$$\begin{aligned}
\hat{S}_{ABM}([\texttt{Mallet.m_Mallet}]_M) &= \{[\text{Ref } \texttt{Alice}]_C, [\text{Ref } \texttt{Bob}]_C, [\text{Ref } \texttt{Mallet}]_C\} \\
\hat{L}_{ABM}([\texttt{Mallet.m_Mallet}]_M) &= \{[\text{Ref } \texttt{Bob}]_C, [\text{Ref } \texttt{Mallet}]_C\} \\
\hat{H}_{ABM}([\text{Ref } \texttt{Mallet}]_C) &= \emptyset \\
\\
\hat{S}_{ABC}([\texttt{Charlie.m_Charlie}]_M) &= \{[\text{Ref } \texttt{Bob}]_C, [\text{Ref } \texttt{Charlie}]_C\} \\
\hat{L}_{ABC}([\texttt{Charlie.m_Charlie}]_M) &= \{[\text{Ref } \texttt{Bob}]_C, [\text{Ref } \texttt{Charlie}]_C\} \\
\hat{H}_{ABC}([\text{Ref } \texttt{Charlie}]_C) &= \emptyset
\end{aligned}$$

As can be seen from the results: $[\text{Ref } \texttt{Alice}]_C \in \hat{S}_{ABM}([\texttt{Mallet.m\_Mallet}]_M)$ thus we can deduce that *possibly* $P_{ABM} \vdash \texttt{Alice} \rightsquigarrow \texttt{Mallet}$ which is consistent with our earlier observations. On the other hand, using the results for P_{ABC} with Corollary 1, we conclude that $P_{ABC} \vdash \texttt{Alice} \not\rightsquigarrow \texttt{Charlie}$.

4.4 From Specification to Constraint Generator

While the high-level Flow Logic style, used above for the control flow logic, is very useful for specification of analyses and for proving correctness of analyses, it may be less obvious how to actually implement such a specification. Below we show how the Flow Logic specification can be turned into a *constraint generator*, generating constraints over the Alternation-free Least Fixed-Point (ALFP) logic, cf. [9]. This enables us to use the *Succinct Solver* to solve the generated constraints efficiently, cf. [9]. Furthermore, as we shall see in Section 5, the formulation of the analysis in terms of ALFP constraints is essential in achieving the goal of this paper: the development of a "Hardest Attacker" for verifying that programs do not leak object references.

It is actually trivial to convert the Flow Logic specification into a constraint generator, because the formulae in the specification are really just ALFP formulae, written in a slightly different notation for legibility (cf. [9]), over the universe formed by our semantic domains. Thus the translation amounts to a simple change of notation and below we therefore show only one example clause:

$$\mathcal{G}[\![(m_0, pc_0) : \texttt{putfield } f]\!] \;=\; \forall [\text{Ref } \sigma]_C \in \hat{S}([m_0]_M) : \\ \hat{S}([m_0]_M) \subseteq \hat{H}([\text{Ref } \sigma]_C)$$

The following lifts the constraint generator to cover programs:

$$\begin{aligned} \mathcal{G}[\![P]\!] = & \bigwedge_{\substack{(m,pc) \in P.addresses, \\ m.instructionAt(pc) = \texttt{instr}}} \mathcal{G}[\![(m, pc) : \texttt{instr}]\!] \\ \wedge & \bigwedge_{\sigma \in P.classes} [\text{Ref } \sigma]_C \in \hat{L}([m_\sigma]_M) \\ \wedge & \bigwedge_{P.sharing(\sigma) = \sigma_1 :: \cdots :: \sigma_n} \{[\text{Ref } \sigma_1]_C, \ldots, [\text{Ref } \sigma_n]_C\} \subseteq \hat{L}([m_\sigma]_M) \end{aligned}$$

The next lemma establishes the correctness of the constraint generator

Lemma 1. *Let* $P \in \mathsf{Program}$*, then* $(\hat{H}, \hat{L}, \hat{S}) \models P$ *iff* $(\hat{H}, \hat{L}, \hat{S}) \models_{\text{ALFP}} \mathcal{G}[\![P]\!]$.

Finally, we can show that the solutions to the analysis constitute a *Moore family*, i.e., that the intersection of a set of acceptable analyses is also an acceptable analysis. This actually follows from Proposition 1 in [9] which establishes a general Moore family result for solutions to ALFP formulae[1]. Using that result in combination with the above we obtain the following

Corollary 2. *The set of acceptable analyses for a given program,* P*, is a* Moore family*:* $\forall X : X \subseteq \{(\hat{H}, \hat{L}, \hat{S}) \mid (\hat{H}, \hat{L}, \hat{S}) \models P\} \Rightarrow (\sqcap X) \models P$.

The implication of the Moore family result is that for every program there exists an acceptable analysis, $\sqcap\emptyset$, and there always exists a smallest, i.e., best, acceptable analysis, $\sqcap\{(\hat{H}, \hat{L}, \hat{S}) \mid (\hat{H}, \hat{L}, \hat{S}) \models P\}$. The Succinct Solver [9] will efficiently

[1] Since we do not use negation at all in our ALFP constraints, the stratification conditions on negation for Proposition 1 in [9] hold vacuously.

compute the smallest solution for a given set of ALFP constraints. A naïve and unoptimised implementation of the analysis and constraint generator presented in this paper gives rise to a worst case time complexity on the order of $\mathcal{O}(n^5)$, where n is the size of the program being analysed. This can be brought down to $\mathcal{O}(n^4)$ in a fairly straightforward manner and we conjecture that through careful analysis and optimisation this can even be lowered to $\mathcal{O}(n^3)$.

5 The Hardest Attacker

In this section we present a solution to the problem posed in the previous section: How to guarantee that a given applet does not leak object references, regardless of what other (malicious) applets run in the same environment. In the previous section, we showed that a simple control flow analysis can be used to guarantee that a specific program in a specific environment does not leak references. However, this is not enough to deal with the problem of applets loaded *after* the analysis is done. Therefore a different approach is needed. We follow the approach of [3,10] and identify a *hardest attacker* with respect to the control flow analysis specified earlier. A hardest attacker is an attacker with the property that it is no less "effective" than any other attacker and therefore, if the hardest attacker cannot execute a successful attack then *no other attacker* can. The key to making it work is that we only need to find a hardest attacker as seen from the perspective of the control flow analysis, rather than try and give a finite characterisation of all the infinitely many attackers, which may not be possible in general.

The idea behind our particular hardest attacker is that (modulo names) only finitely many "types" of constraints are generated, all specifically determined by the Flow Logic. Therefore it is possible to specify a finite set of constraints, in such a way that any constraints generated by another program will be contained within the constraints of the hardest attacker. Here we rely on the equivalence relations on classes and methods in order to deal with the (possibly infinitely many) names an attacker may use. Since the constraints generated depend only on the *equivalence classes* of names and not on the actual names used, we can simply select equivalence relations that partition the names into a *finite* number of equivalence classes.

Given a program, P, that we wish to protect, we shall call the classes and methods defined in P that should not be shared for *private* classes and methods; classes that are allowed to be shared with other programs are called *sharable* ($P.sharable$ denotes the set of sharable classes in P). Any other class or method is called *public*. We then define equivalence relations, called the *discrete* equivalence relations for P, on classes (methods) that map all private and sharable classes (methods) to an equivalence class of its own and all public classes (methods) to $\bullet_C$ ($\bullet_M$). A program that only defines and creates public classes, objects, and methods is called a *public* program. The set of public programs is denoted $\mathsf{Program}_\bullet$. For convenience we parameterise the set of public programs on the (sharable) classes a public program has initial knowledge of (through sharing): let $\mathcal{I} \subseteq \mathsf{Class}$, we then write $\mathsf{Program}_\bullet^{\mathcal{I}}$ to denote the set of public programs with

access to the classes in $\mathcal{I}$, i.e., $\forall Q \in \mathsf{Program}^{\mathcal{I}}_{\bullet} : \forall \sigma \in Q.classes : Q.sharing(\sigma) \subseteq \mathcal{I}$.

Finally, we need to abstract the method names used in method invocation instructions. In general this can be done by using *static inheritance* instead of dynamic dispatch, i.e., syntactically duplicating inherited methods. For our use, however, it is sufficient to compute a conservative approximation of the abstract method lookup, denoted $\widehat{methodLookup}_{/\equiv}$, used in the control flow analysis:

$$\widehat{methodLookup}_{/\equiv}([\text{Ref } \sigma]_C) = \begin{cases} \bigcup_{m \in P.methods} methodLookup_{/\equiv}(m, [\text{Ref } \sigma]_C) & \text{if } [\text{Ref } \sigma]_C \neq \bullet_C \\ \{\bullet_M\} & \text{otherwise} \end{cases} \quad (1)$$

The above is sound and sufficient for the case where public programs are downloaded to run with our private program. Thus the public program can inherit from the private program, but not the other way around. Note that inheritance is not, as such, related to leaking references and is only mentioned here because we wish to model dynamic dispatch rather than use static inheritance.

We can now specify the hardest attacker (HA) as a constraint that contains all the constraints that the analysis can possibly give rise to:

Definition 4 (Hardest Attacker). *The* Hardest Attacker *with respect to the discrete equivalence relation for P, written $\mathcal{H}_P$, is defined as the following constraint*

$$\begin{aligned} \mathcal{H}_P = {} & \{\bullet_C\} \subseteq \hat{S}(\bullet_M) \wedge \hat{L}(\bullet_M) \subseteq \hat{S}(\bullet_M) \wedge \hat{S}(\bullet_M) \subseteq \hat{L}(\bullet_M) \wedge \{\mathsf{INT}\} \subseteq \hat{S}(\bullet_M) \wedge \\ & \forall [\text{Ref } \sigma]_C \in \hat{S}(\bullet_M) : \hat{H}([\text{Ref } \sigma]_C) \subseteq \hat{S}(\bullet_M) \wedge \\ & \forall [\text{Ref } \sigma]_C \in \hat{S}(\bullet_M) : \hat{S}(\bullet_M) \subseteq \hat{H}([\text{Ref } \sigma]_C) \wedge \\ & \forall \sigma \in P.sharable : \{[\text{Ref } \sigma]_C\} \subseteq \hat{L}(\bullet_M) \wedge \\ & \forall [\text{Ref } \sigma]_C \in \hat{S}(\bullet_M) : \\ & \quad \forall [m_v]_M \in \widehat{methodLookup}_{/\equiv}([\text{Ref } \sigma]_C) \\ & \quad\quad \hat{S}(\bullet_M) \subseteq \hat{L}([m_v]_M) \\ & \quad\quad [m_v]_M.returnVal \Rightarrow \hat{S}([m_v]_M) \subseteq \hat{S}(\bullet_M) \end{aligned}$$

The Lemma below is the formal statement of the fact that the HA as defined actually generates (or contains) all the constraints that can possibly be generated from a program with initial knowledge of public classes and methods only:

Lemma 2. *Let $P \in \mathsf{Program}$ and $\mathcal{I} \subseteq P.sharable$ then $\forall Q \in \mathsf{Program}^{\mathcal{I}}_{\bullet}$: $(\hat{H}, \hat{L}, \hat{S}) \models_{\text{ALFP}} \mathcal{H}_P \Rightarrow (\hat{H}, \hat{L}, \hat{S}) \models_{\text{ALFP}} \mathcal{G}[\![Q]\!]$.*

This leads to the following observation:

Lemma 3. *Let $P \in \mathsf{Program}$, $\mathcal{I} \subseteq P.sharable$, and assume $(\hat{H}, \hat{L}, \hat{S}) \models_{\text{ALFP}} \mathcal{G}[\![P]\!] \wedge \mathcal{H}_P$, then $\forall Q \in \mathsf{Program}^{\mathcal{I}}_{\bullet}$: $(\hat{H}, \hat{L}, \hat{S}) \models_{\text{ALFP}} \mathcal{G}[\![P]\!] \wedge \mathcal{G}[\![Q]\!]$.*

We can now state and prove the main Theorem for hardest attackers:

Theorem 2. *For $P \in \mathsf{Program}$ and $\mathcal{I} \subseteq P.sharable$ assume $(\hat{H}, \hat{L}, \hat{S}) \models_{\text{ALFP}} \mathcal{G}[\![P]\!] \wedge \mathcal{H}_P$, then $\forall Q \in \mathsf{Program}^{\mathcal{I}}_{\bullet}$: $(\hat{H}, \hat{L}, \hat{S}) \models (P \,|\, Q)$.*

Proof. Follows from Lemmas 1 and 3, and a technical lemma regarding program composition. ∎

Below we state a corollary showing explicitly how this can be used to validate that a program does not leak any private knowledge. However, the above theorem is of a more general nature since *any* property that can be validated using only an analysis result is amenable to the Hardest Attacker approach.

Corollary 3. *Let* $P \in \mathsf{Program}$*,* $\mathcal{I} \subseteq P.sharable$*, such that* $(\hat{H}, \hat{L}, \hat{S}) \models_{\mathrm{ALFP}} \mathcal{G}[\![P]\!] \wedge \mathcal{H}_P$ *and assume that* $[\mathrm{Ref}\ \tau]_C \notin \hat{H}([\mathrm{Ref}\ \sigma]_C)$ *and for all* $[m_v]_M$ *s.t.* $[\mathrm{Ref}\ \sigma]_C \in [m_v]_M.class$ *implies* $[\mathrm{Ref}\ \tau]_C \notin (\hat{S}([m_v]_M) \cup \hat{L}([m_v]_M))$ *then* $\forall Q \in \mathsf{Program}_\bullet^{\mathcal{I}} : (P \mid Q) \vdash \tau \not\rightsquigarrow \sigma$

In other words: If no leaks can be detected in the result of analysing a program, P, in conjunction with the Hardest Attacker, then P will *never* leak to any public program Q.

In the following we apply the Hardest Attacker approach to the examples shown earlier. For program P_{AB} the discrete equivalence relations are easily computed: the classes `Alice` and `Bob` are in their own respective equivalence classes, denoted $[\mathrm{Ref}\ \texttt{Alice}]_C$ and $[\mathrm{Ref}\ \texttt{Bob}]_C$. Similarly for methods: every method defined in `Alice` of `Bob` is mapped to its own equivalence class.

The abstract method lookup is computed as specified by (1):

$$\widehat{methodLookup}_{/\equiv}(x) = \begin{cases} \{[\texttt{Alice.m\_Alice}]_M\} & \text{if } x = [\mathrm{Ref}\ \texttt{Alice}]_C \\ \{[\texttt{Bob.m\_Bob}]_M, [\texttt{Bob.update}]_M\} & \text{if } x = [\mathrm{Ref}\ \texttt{Bob}]_C \\ \{\bullet_M\} & \text{otherwise} \end{cases}$$

The next step is to generate constraints according to Theorem 2 and solving them: $\mathcal{G}[\![P_{AB}]\!] \wedge \mathcal{H}_{P_{AB}}$. From the small excerpt of the result, shown below, it follows that the requirements of Corollary 1 do not hold, since a reference to `Alice` potentially *can* be leaked, and thus the program is (potentially) not secure: $\{[\mathrm{Ref}\ \texttt{Alice}]_C\} \subset \hat{S}(\bullet_M)$, $\{[\mathrm{Ref}\ \texttt{Alice}]_C\} \subset \hat{L}(\bullet_M)$, $\{[\mathrm{Ref}\ \texttt{Alice}]_C\} \subset \hat{H}(\bullet_C)$. Figure 6 shows a version of the program where the security flaw has been removed. Applying the same techniques as above, we obtain the following result: $(\hat{S}(\bullet_M) \cup \hat{L}(\bullet_M) \cup \hat{H}(\bullet_C)) \cap \{[\mathrm{Ref}\ \texttt{Alice}]_C\} = \emptyset$. Thus, by Corollary 3, we conclude that for all $\mathcal{I} \subseteq P_{AB'}.sharable$: $\forall Q \in \mathsf{Program}_\bullet^{\mathcal{I}}$: $\forall \sigma \in$

```
class Alice {                                  class Bob {
  void m_Alice(Bob) {                        void m_Bob() {
    0: load 1                                    0: return
    1: load 0                                  }
    2: invokevirtual Bob.update(Object)        void update(Object) {
    3: return                                    /* ... */
  }                          0: return
  /* ... */                                   }
}                                            }
```

Fig. 6. Program $P_{AB'}$: a corrected version of P_{AB}

Q.classes: $(P_{AB'} \mid Q) \vdash \texttt{Alice} \not\rightsquigarrow \sigma$ and therefore that program $P_{AB'}$ does not leak a reference to `Alice` to *any* public program.

6 Conclusions and Future Work

In this paper we have presented a method for verifying that applets do not leak sensitive object references to *any* attack-applet. The hardest attacker approach makes it possible to assure security of sensitive applets even when dynamic loading of applets is allowed. Our approach is not confined to checking for leaking references. Any property that can be verified by using the analysis result alone are amenable to the hardest attacker as described in Section 5. Investigating precisely what properties can be expressed this way is left for future work; as is extending the analysis to include more advanced features.

Acknowledgements. The author would like to thank Flemming Nielson, Mikael Buchholtz, and Andrei Sabelfeld for reading early drafts of this paper, and also the anonymous referees for helpful suggestions.

References

1. Élouard, M., Jensen, T.: Secure object flow analysis for Java Card. In: Proc. of Smart Card Research and Advanced Application Conference, Cardis'02. (2002)
2. Nielson, H.R., Nielson, F.: Hardest Attackers. In: Workshop on Issues in the Theory of Security, WITS'00. (2000)
3. Nielson, F., Nielson, H.R., Hansen, R.R., Jensen, J.G.: Validating Firewalls in Mobile Ambients. In: Proc. of conference on Concurrency Theory, CONCUR'99. Volume 1664 of Lecture Notes in Computer Science., Springer Verlag (1999) 463–477
4. Siveroni, I., Hankin, C.: A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2. Available from [11] (2001)
5. Hansen, R.R.: Flow Logic for Carmel. SECSAFE-IMM-001-1.5. Available from [11] (2002)
6. Vitek, J., Horspool, R.N., Uhl, J.S.: Compile-Time Analysis of Object-Oriented Programs. In: Proc. International Conference on Compiler Construction, CC'92. Volume 641 of Lecture Notes in Computer Science., Springer Verlag (1992)
7. Nielson, H.R., Nielson, F.: Flow Logic: a multi-paradigmatic approach to static analysis. In: The Essence of Computation: Complexity, Analysis, Transformation. Volume 2566 of Lecture Notes in Computer Science. Springer Verlag (2002) 223–244
8. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer Verlag (1999)
9. Nielson, F., Nielson, H.R., Seidl, H.: A Succinct Solver for ALFP. Nordic Journal of Computing **2002** (2002) 335–372
10. Nielson, F., Nielson, H.R., Hansen, R.R.: Validating Firewalls using Flow Logics. Theoretical Computer Science **283** (2002) 381–418
11. Siveroni, I.: SecSafe. Web page: `http://www.doc.ic.ac.uk/siveroni/secsafe/` (2003)