Non-Interference and Erasure Policies for Java Card Bytecode

René Rydhof Hansen and Christian W. Probst

Informatics and Mathematical Modelling
Technical University of Denmark
E-mail: {rrh,probst}@imm.dtu.dk

Abstract. Non-interference is the property of a program not to leak any secret information. In this paper we propose a notion of *non-interference* for an abstract version of the Java Card bytecode language. Furthermore an *information-flow analysis* for verifying non-interference is developed and proved sound and correct with respect to the formal semantics of the language. The information-flow analysis can automatically verify the absence of leaks in a program, thus proving non-interference.
Based on the definition of non-interference we propose a notion of *simple erasure policies*. These allow to statically check that confidential information is unavailable after a certain point—and that this unavailability is enforced by te system. This is a crucial requirements for systems like e-commerce or e-voting.

1 Introduction

Smart cards have found widespread use in applications with stringent requirements on handling secret or private information in a safe and secure manner, e.g., electronic purses and GSM cards for mobile phones. To secure the secret and private information, it is important to ensure that programs intended to run on a smart card do not leak sensitive information, whether through accident or on purpose. Consequently, software handling confidential information on smart cards often is required to be formally certified in accordance with rigorous security standards, e.g., the Common Criteria [4], which mandate the use of formal methods and techniques (for assurance level EAL5 and up) to guarantee that the program under scrutiny does not have any leaks. However, verifying that a program does not have any leaks is hard, and doing it manually for even a modestly sized application is infeasible. In this paper we propose a fully automated solution to the problem, by developing an *information flow analysis* that can statically, i.e., at compile-time, verify that a program does not leak any secret information. This is formalised and formally proved by establishing that the analysis is correct with respect to the formal semantics of the language and that it guarantees a notion of *non-interference*.

Furthermore, we propose a notion of *simple erasure* policies based on [3]. In the context of smart cards erasure policies allow for example to formalise the security policy that confidential data must only be stored on the smart

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

Fig. 1. Carmel Core instruction set

card in encrypted form, except temporarily for communication or computation purposes, when the unencrypted form is needed. Such properties are hard to capture using traditional information-flow policies. By using erasure policies they can not only be expressed, but one can also analyse whether the system under scrutiny enforces these properties or not.

2 The Carmel Core Language

In order to simplify formal developments and presentation we use the Carmel Core language [11, 7]. The Carmel Core language is an abstraction of the Java Card bytecode language that facilitates the use of formal methods by abstracting away some of the implementation details, e.g., the constant pool, and by removing language constructs and features that are not essential for the purpose at hand, e.g., static fields and static methods. This results in a language that is powerful enough to incorporate key features of Java Card bytcode, such as classes and dynamic heap allocation. At the same time it is well-suited for formal methods and proofs. Carmel Core is a subset of the Carmel language, a rational reconstruction of Java Card bytecode with the full feature set of Java Card bytecode and the Java Card Run-time Environment. Mapping a Java Card bytecode program to an equivalent Carmel program is a matter of applying a trivial syntactic transformation, e.g., to translate the redundant `push` instructions in Java Card bytecode to the generic `push` instruction of Carmel. In this paper we concentrate on the Carmel Core subset of Carmel and leave for future work the extension to the full Carmel language. This section gives a brief introduction to the formal semantics of Carmel Core.

The instruction set for Carmel Core, shown in Figure 1, includes instructions for stack manipulation, local variables, object generation, field access, a simple conditional, and method invocation and return. A program $P \in \mathsf{Program}$ is defined to be the set of classes $P.classes$ it defines. Each class $\sigma \in \mathsf{Class}$ contains a set of methods $\sigma.methods \subseteq \mathsf{Method}$, and a number of instance fields $\sigma.fields \subseteq \mathsf{Field}$. Each method comprises an instruction for each program counter $pc \in \mathsf{PC}$ in the method $m.instructionAt(pc) \in \mathsf{Instr}$.

The semantics for Carmel Core is defined as a straightforward small-step semantics. The semantic domains are shown in Figure 2 and in Figure 3 an excerpt of the small step semantics is shown. The reduction relation of the semantics is defined over Conf, the domain of semantic configurations. A configuration is either a final configuration, containing only a final heap and a return

Val	$= \mathsf{Num} + \mathsf{Ref}$	Num	$= \mathbb{Z}$
PC	$= \mathbb{N}_0$	Addr	$= \mathsf{Method} \times \mathsf{PC}$
Ref	$= \mathsf{Location} \cup \{\mathtt{null}\}$	Heap	$= \mathsf{Ref} \to \mathsf{Object}$
Object	$= \mathsf{Field} \to \mathsf{Val}$	$\mathsf{LocHeap}$	$= \mathbb{N}_0 \to \mathsf{Val}$
Stack	$= \mathsf{Val}^*$	Frame	$= \mathsf{Method} \times \mathsf{PC} \times \mathsf{LocHeap} \times \mathsf{Stack}$
Conf	$= (\mathsf{Heap} \times \mathsf{Frame}^*) + \mathsf{Val}_\perp$		

Fig. 2. Semantic domains

value ($\langle H, \langle \mathsf{Ret}\ v\rangle\rangle$) or it is a running configuration that contains a heap and a call stack (a stack of frames). A frame consists of the current method, current program counter, local heap, and operand stack. The remaining domains are straightforward and we shall not go into further details here.

In order to complete our discussion of the semantics, we need to define the initial configurations for a given program. In Java Card bytecode, 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 classes in the program $\sigma \in P.classes$ with a corresponding object reference loc_σ pointing to that instance, and a single entry point m_σ. This is formalised in

Definition 1 (Initial Configurations). *If* $P \in \mathsf{Program}$ *then* C *is an* initial configuration *if and only if* $\sigma \in P.classes$ *and* $C = \langle H, \langle m_\sigma, 0, [0 \mapsto loc_\sigma], \epsilon\rangle :: \epsilon\rangle$

In Figure 4 an excerpt of an example program is shown. The program is discussed in more detail in the following section.

3 Non-Interference for Carmel

The notion of secure information flow defined in this paper is based on the observation that the information that must be protected on a smart card, e.g., a PIN code, typically resides in particular instance fields of objects in memory. Thus our notion of security should be flexible enough to allow applets to manipulate and temporarily store sensitive information, e.g. on the operand stack, yet strict enough that it catches and rejects any applet that tries to store high security information in a low security field. This is loosely inspired by the approach taken in [6, 10].

The example program shown in Figure 4 illustrates some of the security issues in Java Card bytecode. The example is an excerpt of an applet that models a situation where a customer (modelled by a `Customer` class) wishes to purchase something from a merchant (modelled by the `Merchant` class). Before allowing the customer to purchase anything, the merchant validates, using the bank (modelled by the `Bank` class), that the customer has enough money in his/her

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

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

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{if}\ \mathit{cmp}\ \texttt{goto}\ pc_0 \\ pc_1 = \begin{cases} pc_0 & \text{if } \mathit{cmp}(v_1, v_2) = \mathit{true} \\ pc+1 & \text{otherwise} \end{cases} \end{array}}{P \vdash \langle H, \langle m, pc, L, v_1 :: v_2 :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc_1, L, S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{getfield}\ f \\ \mathit{loc} \neq \texttt{null} \qquad o = H(\mathit{loc}) \qquad v = o.\mathit{fieldValue}(f) \end{array}}{P \vdash \langle H, \langle m, pc, L, \mathit{loc} :: S\rangle :: SF\rangle \Longrightarrow \langle H, \langle m, pc+1, L, v :: S\rangle :: SF\rangle}$$

$$\frac{m.\mathit{instructionAt}(pc) = \texttt{new}\ \sigma \qquad \sigma \in \mathsf{Class} \qquad (\mathit{loc}, H') = \mathit{newObject}(\sigma, H)}{P \vdash \langle H, \langle m, pc, L, S\rangle :: SF\rangle \Longrightarrow \langle H', \langle m, pc+1, L, \mathit{loc} :: S\rangle :: SF\rangle}$$

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{invokevirtual}\ m_0 \\ \mathit{loc} \neq \texttt{null} \qquad o = H(\mathit{loc}) \\ L_v = \mathit{loc} :: v_1 \cdots :: v_{|m_0|} \qquad m_v = \mathit{methodLookup}(m_0, o.\mathit{class}) \end{array}}{\begin{array}{l} P \vdash \langle H, \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: \mathit{loc} :: S\rangle :: SF\rangle \Longrightarrow \\ \qquad \langle H, \langle m_v, 0, L_v, \epsilon\rangle :: \langle m, pc, L, v_1 :: \cdots :: v_{|m_0|} :: \mathit{loc} :: S\rangle :: SF\rangle \end{array}}$$

$$\frac{\begin{array}{c} m.\mathit{instructionAt}(pc) = \texttt{return}\ t \\ S' = v'_1 :: \cdots :: v'_{|m|} :: \mathit{loc} :: S'' \end{array}}{\begin{array}{l} P \vdash \langle H, \langle m, pc, L, v :: S\rangle :: \langle m', pc', L', S'\rangle :: SF\rangle \Longrightarrow \\ \qquad\qquad \langle H, \langle m', pc'+1, L', v :: S''\rangle :: SF\rangle \end{array}}$$

$$\frac{m.\mathit{instructionAt}(pc) = \texttt{return}\ t}{P \vdash \langle H, \langle m, pc, L, v :: S\rangle :: \epsilon\rangle \Longrightarrow \langle H, \langle \mathsf{Ret}\ v\rangle\rangle}$$

Fig. 3. Small step semantics for Carmel Core (excerpt)

```
public class Customer
{
  int account_no;
  Merchant merchant;
  /* ... */
  public void buysomething(void)
  {
    0: load 0
    1: getfield merchant
    2: load 0
    3: getfield account_no
    4: push 42
    5: invokevirtual Merchant.purchase(int,int)
    6: return
  }
}
public class Bank
{
  /* ... */
  public short validate(int,int)
  {
    /* ... */
  }
}
```

```
public class Merchant
{
  Bank bank;
  int  stolen_acct;

  /* ... */

  public void purchase(int,int)
  {
     0: load 0
     1: getfield bank
     2: load 2
     3: load 1
     4: invokevirtual Bank.validate(int,int)
     5: if eq 0 goto 9
     6: load 0
     7: getfield bank
     8: invokevirtual Bank.execute(void)
     9: load 0
    10: load 2
    11: putfield stolen_acct
    12: return
  }
}
```

Fig. 4. Example program

account to purchase the item in question. This requires the customer to send his account number to the merchant who then forwards it to the bank. Obviously the customer does not want the merchant to keep the account number longer than strictly necessary, i.e., the merchant should "forget" the account number once the purchase has been validated and executed. However, in the example program the merchant actually steals the account number (lines 9–11 in the `purchase` method of class `Merchant`), presumably in order to (ab-)use it later for nefarious activities. While rather unsubtle, it does demonstrate the need for a flexible notion of secure information flow that can handle a situation as the one described above where secret information (the account number) actually is allowed to flow to a potentially malicious entity (the merchant). The key observation here of course is that the merchant is not allowed to *store* the confidential information or any information *derived* from the confidential data, i.e., the merchant should be absolutely oblivious to the actual account number since all checking/processing of the account number is done by the bank. The remainder of this section is devoted to describing a notion of *non-interference* suitable for such applications.

In order to formalise the above intuitions we define a *security policy* to be a map that assigns a *security level* to every instance field. We assume that the set of security levels forms a lattice, denoted $(\mathsf{Level}, \sqsubseteq)$:

Definition 2 (Security policy). *A* security policy *is a total function, level* : $\mathsf{Field} \to \mathsf{Level}$*, that assigns a security level to instance fields.*

To simplify presentation, we consider a simple security lattice with only two levels: low (L) and high (H) with the obvious ordering: $\mathsf{L} \sqsubseteq \mathsf{H}$; however, the developments in this paper are easily generalised to consider a lattice of many different security levels. For the example program in Figure 4 we require that $\texttt{Customer.account\_no}.\mathit{level} = \mathsf{H}$ and $\forall f \in \texttt{Merchant}.\mathit{fields} : f.\mathit{level} = \mathsf{L}$ to indicate that no information regarding the customers account number must be leaked to the merchant.

The ability to dynamically allocate objects on the heap and the subsequent handling of object references in an applet poses a particular challenge when defining non-interference for Carmel Core and languages with similar features. We overcome this by defining non-interference "up to" isomorphism on memory locations in the heap that contain objects with at least one field classified as *low*-security. In preparation of this we first introduce the following relation, called π-equivalence, for comparing values in a program up to the given π-mapping. The map is required to be bijective (on the subset of Loc on which it is defined); in the definition of heap equivalence (Definition 4) the map must be an isomorphism on the locations that point to objects containing fields with a low-security classification:

Definition 3 (π-equivalence). *Let* $v_1, v_2 \in \mathsf{Val}$ *and* $\pi : \mathsf{Loc} \to \mathsf{Loc}$ *be a bijective partial map and define*

$$v_1 \equiv_\pi v_2 \quad \textit{iff} \quad \begin{cases} v_1 = v_2 & \textit{if } v_1, v_2 \notin \mathsf{Loc} \\ v_2 = \pi(v_1) & \textit{if } v_1 \in \mathrm{dom}(\pi) \\ v_1 = \pi^{-1}(v_2) & \textit{if } v_2 \in \mathrm{codom}(\pi) \\ \textit{true} & \textit{if } v_1 \in \mathsf{Loc} \setminus \mathrm{dom}(\pi), v_2 \in \mathsf{Loc} \setminus \mathrm{codom}(\pi) \end{cases}$$

As already mentioned, the kind of non-interference of interest here must be able to prevent leaks from high-security instance fields to low-security instance fields. Local variables and stack contents are of no concern here since they are only used temporarily for storing secret information. Thus security, as defined here, is only concerned with the contents of the heap. The following definition formalises that two heaps are considered to be equivalent, as seen from a security perspective, when all fields that have a low security classification are equivalent (modulo the isomorphism on low heap locations):

Definition 4 (Heap-equivalence). *Let* $H_1, H_2 \in \mathsf{Heap}$*, then* $H_1 \approx_{\mathsf{L}} H_2$ *if and only if there exists a bijective partial map,* $\pi : \mathsf{Loc} \to \mathsf{Loc}$*, such that*

$$\begin{aligned} &\forall \mathit{loc}_1 \in \mathrm{dom}(H_1) : \forall f \in H_1(\mathit{loc}_1).\mathit{fields} : f.\mathit{level} \sqsubseteq \mathsf{L} \Rightarrow \\ &\quad H_1(\mathit{loc}_1).\mathit{class} = H_2(\pi(\mathit{loc}_1)).\mathit{class} \wedge H_1(\mathit{loc}_1).f \equiv_\pi H_2(\pi(\mathit{loc}_1)).f \end{aligned}$$

and

$$\begin{aligned}&\forall loc_2 \in \mathrm{dom}(H_2) : \forall f \in H_2(loc_2).\mathit{fields} : f.\mathit{level} \sqsubseteq \mathsf{L} \Rightarrow \\ &\quad H_1(\pi^{-1}(loc_2)).\mathit{class} = H_2(loc_2).\mathit{class} \wedge H_1(\pi^{-1}(loc_2)).f \equiv_\pi H_2(loc_2).f\end{aligned}$$

A mapping π that fulfils the above requirements is called a *low-isomorphism* on locations. Note that π is not defined for all locations, only those that point to objects that contain at least one field of a low security classification. Thus any object that is composed entirely of high-security fields is "invisible" to a low security observer. We can now define non-interference for Carmel:

Definition 5 (Non-Interference). *Let* $P \in \mathsf{Program}$, $H_1, H_2 \in \mathsf{Heap}$ *and let* $\langle H_i, \langle m_i, 0, L_i, \epsilon\rangle\rangle$ *for* $i = 1, 2$ *be initial configurations for* P *such that* $P \vdash \langle H_i, \langle m_i, 0, L_i, \epsilon\rangle\rangle \Longrightarrow^* \langle H'_i, \langle \mathsf{Ret}\ v_i\rangle\rangle$ *then* P *is said to be* non-interfering *if and only if* $H_1 \approx_{\mathsf{L}} H_2 \;\Rightarrow\; H'_1 \approx_{\mathsf{L}} H'_2$

Intuitively this interpretation of non-interference states that if a given program is started in two different initial configurations with equivalent heaps, then the program is non-interfering if both executions terminate and the heaps in the final configurations are equivalent. This guarantees that no information in a high-security field could, in any way, have been leaked to any low-security field.

The definition of non-interference has a number of noteworthy implications. First, it only applies to terminating programs and thus cannot prevent information leaks through termination or timing behaviour. However, since Java Card programs running on a smart card are expected to always terminate, timing attacks pose only a minor security risk. In [1] a program transformation that can eliminate such timing leaks is discussed. The second thing to note is that the definition of non-interference could trivially be extended to also cover return values and thus implement an aspect of input/output non-interference as well as heap-equivalence, however, the execution model of Java Card bytecode, and thus Carmel, makes this less interesting.

4 Control and Information Flow Analysis

In this section we develop an *information flow analysis* which can be used to statically determine if a program has the non-interference property of Definition 5 with respect to a given security policy. The analysis is developed as an extension of a previously specified control flow analysis; we shall not go into the control flow aspects of the analysis here, merely refer to [7] for the details.

One of the main problems to overcome for an information flow analysis of a low-level bytecode language is to take implicit flows into account and to ensure that they are handled correctly. The information flow analysis described in the following incorporates a special component specifically to track the implicit flow of a program. However, there is another problem related to the implicit flows: conditionals in Carmel, and other low level languages, are essentially conditional jumps and, in contrast to higher level languages, there is no program structure to indicate or even suggest the scope of a conditional statement. In order to

$$
\begin{array}{rlrl}
\widehat{\mathsf{Stack}}_{\mathsf{IFA}} = \overline{\mathsf{Addr}} \to ((\widehat{\mathsf{Val}} \times \mathsf{Security})^*)^\top & \widehat{\mathsf{LocHeap}}_{\mathsf{IFA}} = \overline{\mathsf{Addr}} \to \mathbb{N}_0 \to (\widehat{\mathsf{Val}} \times \mathsf{Security}) \\
\widehat{\mathsf{Object}}_{\mathsf{IFA}} = \mathsf{Field} \to (\widehat{\mathsf{Val}} \times \mathsf{Security}) & \widehat{\mathsf{Heap}}_{\mathsf{IFA}} = \overline{\mathsf{ObjRef}} \to \widehat{\mathsf{Object}}_{\mathsf{IFA}} \\
\widehat{\mathsf{Implicit}} = \overline{\mathsf{Addr}} \to \mathcal{P}(\mathsf{Security} \times \overline{\mathsf{Addr}}) & \mathsf{Dominators} = \overline{\mathsf{Addr}} \to \mathcal{P}(\mathsf{PC})
\end{array}
$$

Fig. 5. Abstract Domains for the Information Flow Analysis

recover (some of) that structure, the analysis computes the *post-dominators* or forward dominators for all program points; such post-dominators represent program points where every terminating execution from the corresponding conditional *must* pass through regardless of the branch taken. This is similar to the approach taken in [8, 2]. For a configuration $C = \langle H, \langle m, pc, L, S\rangle :: SF\rangle$ let $C.\text{address} = (m, pc)$. The post-dominator can then be formally defined in the current setting as follows:

Definition 6 (Post-dominator). *For $P \in \mathsf{Program}$ the program counter pc is a* post-dominator *for (m_1, pc_1), written $(m_1, pc_1) \curvearrowright pc'_1$, if for all reduction sequences with $C_1.\text{address} = (m_1, pc_1)$ and of the form: $P \vdash C_0 \Longrightarrow^* C_1 \Longrightarrow \cdots \Longrightarrow C_n \Longrightarrow \langle H, \langle \mathsf{Ret}\ v\rangle\rangle$ there exists an $i \in \{2, \ldots, n\}$ such that $C_i.\text{address} = (m_1, pc'_1)$.*

Later we show how this definition is instantiated for Carmel Core.

4.1 Abstract Domains

The abstract domains for the information flow analysis are mainly extensions of the abstract domains for the control flow analysis with security information. In addition abstract domains are needed to compute the post-dominators (the $\mathsf{Dominators}$ domain), as discussed above, and to track the implicit flow (the $\widehat{\mathsf{Implicit}}$ domain). The abstract domains are shown in Figure 5.

We can now show how to compute post-dominators for Carmel Core:

Definition 7. *Let $P \in \mathsf{Program}$ and $DOM \in \mathsf{Dominators}$, then DOM is the set of post-dominators for P, written $DOM(P)$, if and only if*

$$
\forall m \in P.\text{methods} : DOM(m, pc) = \begin{cases} \{pc\} \cup (DOM(m, pc+1) \cap DOM(m, pc_0)) & \\ & \textit{if } m.\textit{instructionAt}(pc) = \texttt{if}\ t\ \textit{cmp}\ \texttt{goto}\ pc_0 \\ \{\mathsf{END}_m\} & \textit{if } m.\textit{instructionAt}(pc) = \texttt{return} \\ \{pc\} \cup DOM(m, pc+1) & \textit{otherwise} \end{cases}
$$

That this definition indeed captures the notion of post-dominator is established by the following:

Lemma 1. *Let $P \in \mathsf{Program}$ such that $DOM(P)$ and such that*

$$P \vdash C_1 \Longrightarrow C_2 \Longrightarrow \cdots \Longrightarrow C_n \Longrightarrow \langle H, \langle \mathsf{Ret}\ v\rangle\rangle$$

where $C_1 = \langle H_1, \langle m_1, pc_1, L_1, S_1\rangle :: SF_1\rangle$ *then*

$$\forall pc_1' \in DOM(m_1, pc_1) \colon \exists i \in \{1, \ldots, n\} : C_i = \langle H_i, \langle m_i, pc_1', L_i, S_i\rangle :: SF_i\rangle$$

Proof. By induction in n, the length of the instruction sequence, and case analysis.

For a detailed proof see [7].

The least upper bound of the security levels of the possible implicit flows at an address, i.e., $\sqcup\{\ell' \,|\, (\ell', (m', pc')) \in \hat{C}(m, pc)\}$, is called the *security context* of that address and is written $\sqcup\hat{C}(m, pc)$. Tracking implicit flow requires keeping track of the security label of the implicit flow and also the origin of the implicit flow, i.e., the program point of the conditional or method invocation that gave rise to the implicit flow. Implicit flows originating at program point pc must be propagated throughout the program until a post-dominator for pc is encountered. This is formalised as follows for $\hat{C}_1, \hat{C}_2 \in \widehat{\mathsf{Implicit}}$ and $DOM \in \mathsf{Dominators}$:

$$\begin{array}{l} \hat{C}_1(m_1, pc_1) \sqsubseteq_{DOM} \hat{C}_2(m_2, pc_2) \quad \text{iff} \\ \quad \{(\ell, (m, pc)) \in \hat{C}_1(m_1, pc_1) \,|\, m_2 \neq m \vee pc_2 \notin DOM(m, pc)\} \subseteq \hat{C}_2(m_2, pc_2) \end{array}$$

Putting all of the above together results in the following abstract domain for the information flow analysis:

$$\widehat{\mathsf{Analysis}}_{\mathsf{IFA}} = \widehat{\mathsf{Heap}}_{\mathsf{IFA}} \times \widehat{\mathsf{LocHeap}}_{\mathsf{IFA}} \times \widehat{\mathsf{Stack}}_{\mathsf{IFA}} \times \widehat{\mathsf{Implicit}} \times \mathsf{Dominators}$$

Elements of the analysis domain are written $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM)$ where the semicolon serves as a reminder that the dominator component, DOM, is a parameter to the Flow Logic specification and is not, as such, part of the analysis.

4.2 Flow Logic Specification

The information flow analysis is specified using the Flow Logic framework, cf. [9], and is composed of three mostly independent components: a control flow analysis, tracking of implicit flows, and calculation of dominators. This gives Flow Logic judgements of the form: $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{instr}$ with the intended meaning that $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM)$ is a correct analysis of the instruction `instr` at program counter pc in method m, Figure 6 show an excerpt of the specification for the information flow analysis. Below the judgements for conditionals and method invocations are discussed in more detail. For a detailed discussion see [7].

In the analysis specification we use the notation $A_1 :: \cdots :: A_n :: X \triangleleft \hat{S}(m, pc)$ to mean that the abstract stack at instruction (m, pc) has at least n elements bound to the variables A_1 through A_n for later reference. The $X \triangleleft Y$ is generally also used to introduce X as a shorthand for Y in the analysis specification.

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{load}\ x \\
\quad \text{iff} \quad \ell \lhd \sqcup \hat{C}(m, pc) \sqcup \hat{L}_{\downarrow 2}(m, pc)(x) : \\
\qquad \hat{L}_{\downarrow 1}(m, pc)(x)^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1) \\
\qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \\
\qquad \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{if}\ cmp\ \texttt{goto}\ pc_0 \\
\quad \text{iff} \quad {A_1}^{\ell_1} :: {A_2}^{\ell_2} :: X \lhd \hat{S}(m, pc) : \\
\qquad \ell \lhd \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2 : \\
\qquad X \sqsubseteq \hat{S}(m, pc+1) \\
\qquad X \sqsubseteq \hat{S}(m, pc_0) \\
\qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \\
\qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0) \\
\qquad \{(\ell, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1) \\
\qquad \{(\ell, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc_0)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{new}\ \sigma \\
\quad \text{iff} \quad \ell \lhd \sqcup \hat{C}(m, pc) : \\
\qquad \{(\mathsf{Ref}\ \sigma)\}^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc+1) \\
\qquad \mathit{default}_{\ell}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma) \\
\qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \\
\qquad \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{getfield}\ f \\
\quad \text{iff} \quad B^{\ell_1} :: X \lhd \hat{S}(m, pc) : \\
\qquad \forall (\mathsf{Ref}\ \sigma) \in B : \\
\qquad\quad \ell \lhd \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) : \\
\qquad\quad \hat{H}_{\downarrow 1}(\mathsf{Ref}\ \sigma)(f)^{\ell} :: X \sqsubseteq \hat{S}(m, pc+1) \\
\qquad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \\
\qquad \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)
\end{array}
$$

$$
\begin{array}{l}
(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{invokevirtual}\ m_0 \\
\quad \text{iff} \quad {A_1}^{\ell_1} :: \cdots :: {A_{|m_0|}}^{\ell_{|m_0|}} :: B^{\ell_0} :: X \lhd \hat{S}(m, pc) : \\
\qquad \ell \lhd \sqcup \hat{C}(m, pc) : \\
\qquad \forall (\mathsf{Ref}\ \sigma) \in B\!: m_v \lhd \mathit{methodLookup}(m_0, \sigma) : \\
\qquad\quad \{(\mathsf{Ref}\ \sigma)\}^{\ell_0 \sqcup \ell} :: {A_1}^{\ell_1 \sqcup \ell} :: \cdots :: {A_{|m_0|}}^{\ell_{|m_0|} \sqcup \ell} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|] \\
\qquad\quad \{(\ell_0, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m_v, 0) \\
\qquad\quad m_0.\mathit{returnType} = \texttt{void} \Rightarrow X \sqsubseteq \hat{S}(m, pc+1) \\
\qquad\quad m_0.\mathit{returnType} \neq \texttt{void} \Rightarrow \\
\qquad\qquad A^{\ell_A} :: Y \lhd \hat{S}(m_v, \mathsf{END}) : \\
\qquad\qquad\quad A^{\ell_A \sqcup \ell} :: X \sqsubseteq \hat{S}(m, pc+1) \\
\qquad\quad \hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc+1) \\
\qquad\quad \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc+1)
\end{array}
$$

Fig. 6. Information Flow Analysis

Conditionals The security context for the current instruction is determined by the (security label of) the op two values on the operand stack and the implicit flows that may have reached the instruction. First we find the security labels of the top two stack values: ${A_1}^{\ell_1} :: {A_2}^{\ell_2} :: X \lhd \hat{S}(m, pc)$. Based on the security levels of the stack values and the implicit flows the security level for the current instruction is calculated: $\ell \lhd \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2$. Next the rest of the stack is pushed forward to the two possible jump destinations: $X \sqsubseteq \hat{S}(m, pc{+}1)$ and $X \sqsubseteq \hat{S}(m, pc_0)$. Similarly for the local heap: $\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$ and $\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc_0)$. Since conditionals give rise to new implicit flows that must be tracked, the current conditional is added to the set of tracked conditionals (and method invocations), all of which must also be copied forward: $\{(\ell, pc)\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$ and $\{(\ell, pc)\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc_0)$.

Method Invocation The information flow analysis for method invocation proceeds like the semantics fetching parameters from the stack along with a reference to the target object: ${A_1}^{\ell_1} :: \cdots :: {A_{|m_0|}}^{\ell_{|m_0|}} :: B^{\ell_0} :: X \lhd \hat{S}(m, pc)$:. The security levels are then used to calculate the current security context: $\ell \lhd \sqcup \hat{C}(m, pc)$:. Now all object references found on the stack are used for method lookup:

$$\forall (\mathsf{Ref}\ \sigma) \in B : \quad m_v = \mathit{methodLookup}(m_0, \sigma) \ldots$$

Next the parameters are transferred annotated with the updated security context:

$$\{(\mathsf{Ref}\ \sigma)\}^{\ell_0 \sqcup \ell} :: {A_1}^{\ell_1 \sqcup \ell} :: \cdots :: {A_{|m_0|}}^{\ell_{|m_0|} \sqcup \ell} \sqsubseteq \hat{L}(m_v, 0)[0..|m_0|]$$

and the implicit flows are also copied to the invoked method:

$$\{(\ell_0, (m, pc))\} \cup \hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m_v, 0)$$

Any return values from the method invocation are handled as in the control flow analysis updated with the security level of the current context:

$$A^{\ell_A} :: Y \lhd \hat{S}(m_v, \mathsf{END}) : \quad A^{\ell_A \sqcup \ell} :: X \sqsubseteq \hat{S}(m, pc + 1)$$

Then the local heaps and (local) implicit flows are copied forward:

$$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$$
$$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$$

4.3 Soundness and Non-Interference

Proving that the information flow analysis is semantically sound amounts to proving that it can be used to show that a program is non-interfering in the sense of Definition 5. We prove this by establishing that the anlaysis statically guarantees that a so-called $\mathcal{A}$-equivalence, parameterised on the analysis $\mathcal{A}$, holds between semantic configurations of the analysed program; this $\mathcal{A}$-equivalence is then shown to be sufficient to establish non-interference for the analysed program. Only the most important definitions and lemmas are stated here; see [7] for further details.

First we define an equivalence on individual stack frames. Taking the dynamic memory allocation into account the equivalence is defined only up to a given low-ismorphism:

Definition 8. *Let $F_i = \langle m_i, pc_0, L_i, S_i\rangle$ for $i = 1, 2$ be stack frames and let $\mathcal{A} = (\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$ then F_1 and F_2 are $\mathcal{A}$-equivalent, written $F_1 \approx^\pi_{\mathcal{A}} F_2$, if and only if $\pi : \mathsf{Loc} \to \mathsf{Loc}$ is a bijective partial map and the following conditions hold:*

1. $m_1 = m_2$
2. $pc_1 = pc_2$
3. $\forall x : \hat{L}_{\downarrow 2}(m_1, pc_1)(x) \sqsubseteq \mathsf{L} \Rightarrow L_1(x) \equiv_\pi L_2(x)$
4. $\forall i : \hat{S}_{\downarrow 2}(m_1, pc_1)|_i \sqsubseteq \mathsf{L} \Rightarrow S_1|_i \equiv_\pi S_2|_i$

This is trivially extended to call stacks: $SF_1 \approx^\pi_{\mathcal{A}} SF_2$ if and only if $\forall i : SF_1|_i \approx^\pi_{\mathcal{A}} SF_2|_i$. Note that this requires the two call stacks to be of equal length. Now the equivalence of two semantic configurations, modulo $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$, can be defined:

Definition 9 ($\mathcal{A}$-equivalence). *Let $C_i = \langle H_i, SF_i\rangle$ for $i = 1, 2$ be semantic configurations and let $\mathcal{A} = (\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$ then C_1 and C_2 are* $\mathcal{A}$-equivalent*, written $C_1 \approx_{\mathcal{A}} C_2$, if and only if there exists an bijective partial map, $\pi : \mathsf{Loc} \to \mathsf{Loc}$, such that $SF_1 \approx^\pi_{\mathcal{A}} SF_2$ and for all $(\mathsf{Ref}\ \sigma) \in \mathrm{dom}(\hat{H})$ and $f \in \sigma.fields$ the following holds:*

$$
\begin{array}{l}
\hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) \sqsubseteq \mathsf{L} \Rightarrow \\
\quad \forall loc_1\!: H_1(loc_1).class = H_2(\pi(loc_1)).class \\
\qquad\qquad H_1(loc_1).f \equiv_\pi H_2(\pi(loc_1)).f \\
\quad \forall loc_2\!: H_2(loc_2).class = H_1(\pi^{-1}(loc_2)).class \\
\qquad\qquad H_2(loc_2).f \equiv_\pi H_1(\pi^{-1}(loc_1)).f
\end{array}
$$

Note that this definition is very similar to that for heap equivalence, cf. Definition 4, with added requirements on the local heap and the operand stack.

We now state the main technical lemma needed to prove the main theorem. The lemma (called a "hexagon lemma" in [5]) shows that $\mathcal{A}$-equivalence on configurations is preserved by reduction or, more precisely, that $\mathcal{A}$-equivalence is preserved by sufficiently long reduction sequences. Figure 7 summarises the lemma and illustrates the source of its name. Note that this proof works on the assumption that the stack height is fixed for each instruction which is one of the properties guaranteed by the bytecode verifier.

Lemma 2 (Diamond property). *Let $P \in \mathsf{Program}$, $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$, $C_1, C_2 \in \mathsf{Conf}$ such that $\mathcal{A} \models_{\mathit{IFA}} P$ and $P \vdash C_1 \Longrightarrow C_1'' \Longrightarrow^* \langle H_1''', \langle \mathsf{Ret}\ v_1'''\rangle\rangle$, $P \vdash C_2 \Longrightarrow C_2'' \Longrightarrow^* \langle H_2''', \langle \mathsf{Ret}\ v_2'''\rangle\rangle$ with $C_1 \approx_{\mathcal{A}} C_2$ then $\exists C_1', C_2'$ such that $P \vdash C_1'' \Longrightarrow^* C_1'$, $P \vdash C_2'' \Longrightarrow^* C_2'$, and $C_1' \approx_{\mathcal{A}} C_2'$.*

$$\begin{tikzcd}
 & P \vdash C_1 \arrow[dl, Rightarrow] & \approx_{\mathcal{A}} & P \vdash C_2 \arrow[dr, Rightarrow] & \\
C_1'' \arrow[dr, Rightarrow, "*"] & & & & C_2'' \arrow[dl, Rightarrow, "*"'] \\
 & C_1' & \approx_{\mathcal{A}} & C_2' &
\end{tikzcd}$$

Fig. 7. Diamond property

Proof. By case analysis.

Having established the diamond property for $\mathcal{A}$-equivalence all that remains is to relate the security policy for a program to the security levels found by the information flow analysis:

Definition 10 (Security compatible). *For a program,* $P \in \mathsf{Program}$*, such that* $(\hat{K}, \hat{H}, \hat{L}, \hat{S}) \models_{CFA} P$ *the analysis,* $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM)$*, is said to be* security compatible *with* P *if* $\forall \sigma \in P.classes$: $\forall f \in \sigma.fields$: $f.level = \mathsf{L} \Rightarrow \hat{H}_{\downarrow 2}(\mathsf{Ref}\ \sigma)(f) = \mathsf{L}$

Finally, the main non-interference result can be stated and proved:

Theorem 1 (Non-Interference). *Let* $P \in \mathsf{Program}$ *and* $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$ *such that* $\mathcal{A} \models_{CFA} P$ *and* $\mathcal{A}$ *is security compatible with* P*. If* C_0 *and* C_0' *are initial configurations for* P *such that* $C_0 \approx_{\mathcal{A}} C_0'$ *and* $P \vdash C_0 \Longrightarrow^* \langle H, \langle \mathsf{Ret}\ v \rangle\rangle$ *and* $P \vdash C_0' \Longrightarrow^* \langle H', \langle \mathsf{Ret}\ v' \rangle\rangle$ *then* P *is non-interfering, i.e.,* $H \approx_{\mathsf{L}} H'$*.*

Proof. Follows directly by application of Lemma 2 and Definition 9.

The theorem shows that if the security level found by the information flow analysis agrees with those of the given security policy for a given program, then the program is non-interfering and thus no secret information can be leaked.

Returning to the example program of Figure 4, the information flow analysis shows that information from the `Customer.account_no` field may be leaked to the `Merchant.stolen_acct` and thus that the program may be insecure.

5 Simple Erasure Policies

While the notion of non-interference defined in the preceding sections is quite flexible and well-suited to the Carmel execution model, it is sometimes necessary to allow specific confidential information to be made temporarily available to untrusted entities for specific purposes. Examples of this include: account numbers in e-commerce, votes in e-voting systems. In such applications non-interference is too strict. Recent work, cf. [3], suggests *erasure policies* as a possible solution.

Briefly, the idea underlying erasure policies, as defined in [3], is that information that has been labelled with an erasure policy, e.g., $\mathsf{L} \overset{c}{\nearrow} \mathsf{H}$, is available at level L until the condition c holds, after which the information should only be available at level H. This kind of policy is useful for applications where sensitive information is needed only temporarily, e.g., for voting or e-commerce. In the remainder of this section we describe a simplified notion of erasure policy in the context of Carmel programs. This is work in progress and we therefore do not present any formal proofs.

For Carmel programs such erasure policies can be interpreted as follows: once a program run has ended in a terminal configuration, e.g., $\langle H_1', \langle \mathsf{Ret}\ v_1 \rangle\rangle$, any further program runs using H_1' as the initial heap should not be able to extract any information from H_1' about any field that contained information that was to be erased. This kind of policy can be seen as a Chong/Myers erasure policy of the form $\mathsf{L} \overset{c}{\nearrow} \mathsf{H}$ where c is then fixed to mean "when the current program run ends". We shall call such policies "simple erasure policies", written $\mathsf{L} \overset{\mathsf{end}}{\nearrow} \mathsf{H}$, and formally define them as follows: First extend the Level security lattice with an additional element, ${}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}$, such that the three elements of the lattice are ordered in the following way: $\mathsf{L} \sqsubseteq {}^{\mathsf{L}}\!\nearrow_{\mathsf{H}} \sqsubseteq \mathsf{H}$. The element ${}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}$ is then assigned to fields that contain data that should be erased. Given that domain we can now define simple erasure policies formally:

Definition 11 (Simple Erasure). *Let* $P \in \mathsf{Program}$, $H_1, H_2 \in \mathsf{Heap}$ *and let* $\langle H_i, \langle m_i, 0, L_i, \epsilon\rangle\rangle$ *for* $i = 1, 2$ *be initial configurations for* P *such that* $P \vdash \langle H_i, \langle m_i, 0, L_i, \epsilon\rangle\rangle \Longrightarrow^* \langle H_i', \langle \mathsf{Ret}\ v_i\rangle\rangle$ *then* P *is said to comply with the erasure policy* $\mathsf{L} \nearrow \mathsf{H}$ *if and only if* P *is non-interfering and* $H_1 \approx_{\mathsf{L}} H_2 \;\Rightarrow\; H_1' \approx_{{}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}} H_2'$

Intuitively the above definition states that no matter what information is initially stored in a field with the label ${}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}$ such information is erased when the program ends, since the requirement $H_1' \approx_{{}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}} H_2'$ implies that such a field must have the same final value for every program run. This ensures that the next program run, starting from H_1' (or H_2' or any other final heap), will not be able to extract any information about the previous values of such fields. In order for a program to fulfil such a simplified erasure policy it must ensure that all fields classified as ${}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}$ must be explicitly erased or overwritten before program termination. We conjecture that it is relatively straightforward to augment the information flow analysis to also statically verify simple erasure policies by requiring that every field of level ${}^{\mathsf{L}}\!\nearrow_{\mathsf{H}}$ is explicitly erased before the end of a program or before a method returns (possibly by using the already computed post-dominators). This extension of the analysis is left for future work.

We conclude this section with an example illustrating some of the issues discussed above. The example, shown in Figure 8, is a modified version of the program shown in Figure 4. In the modified example a customer first has to create a new "shopping basket" in order to buy goods from the merchant. This procedure requires the customer to send an account number to the merchant. Next the customer adds the items of interest to the basket; the purchase is finalised by cehcking out the items in the basket. This is also where the merchant uses the account number to verify that the customer has sufficient funds to pay

```
public class Customer
{
  int account_no;
  Merchant merchant;
  /* ... */
  public void buysomething(void)
  {
     0: load 0
     1: getfield merchant
     2: load 0
     3: getfield account_no
     4: invokevirtual Merchant.new_basket(int)
     5: load 0
     6: getfield merchant
     7: push 42
     8: invokevirtual Merchant.add_item(int)
     9: load 0
    10: getfield merchant
    11: invokevirtual Merchant.checkout()
    12: return
  }
}

public class Merchant
{
  Bank bank;
  int  acct_cache;
  int  amount;

  /* ... */
```

```
  public void new_basket(int)
  {
    0: load 0
    1: load 2
    2: putfield acct_cache
    /* ... */
  }

  public void add_item(int)
  {
    /* ... */
  }

  public void checkout(void)
  {
    /* validate the purchase */
     0: load 0
     1: getfield bank
     2: load 0
     3: getfield acct_cache
     4: load 0
     5: gefield amount
     6: invokevirtual Bank.validate(int,int)
     7: if eq 0 goto 11
     8: load 0
     9: getfield bank
    10: invokevirtual Bank.execute(void)
    11: return
  }
}
```

Fig. 8. Modified example program

for the items in the basket. Since the merchant retains the account number in an instance field until the purchase is completed the example program cannot be validated as secure using non-interference alone since the account number is indeed "leaked". This is exactly the kind of problem erasure policies were developed to solve: we wich to specify a policy that allows the customer's account number to be stored *temporarily* by the merchant but *only* in the merchant's account cache (from where it can only be sent to the bank for validation). Such a policy can easily be achieved by classifying the account number as high (H) and the account cache as "low until end of purchase" ($^{\mathsf{L}}\nearrow_{\mathsf{H}}$) and finally all other fields of the merchant should be classified as low (L). Using the definition of simple erasure policies on the example program in Figure 8 it can be established that the account number may actually be leaked because the merchant does not take any steps to wipe the account cache once the purchase is completed.

6 Conclusion

We have presented a flexible and strong notion of non-interference for a low-level bytecode language with dynamic memory allocation and argued that is fits the Java Card bytecode execution. Based on this non-interference we have furthermore defined a simplified notion of *erasure policy* useful for programs where sensitive information must be made temporarily available to untrusted parties. To the best of our knowledge this is the first application of erasure policies for a concrete language. A more thorough investigation of suggested simple erasure policies is left as future work.

References

1. Johan Agat. Transforming out Timing Leaks. In *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, POPL'00*, pages 40–53, Boston, Massachusetts, January 2000. ACM Press.
2. Marco Avvenuti, Cinzia Bernardeschi, and Nicoletta De Francesco. Java bytecode verification for secure information flow. *SIGPLAN Notices*, 38(12):20–27, December 2003.
3. Stephen Chong and Andrew C. Myers. Language-Based Information Erasure. In *Proc. of the 18th IEEE Computer Security Foundations Workshop*, Aix-en-Provence, France, June 2005. IEEE Computer Society.
4. Common Criteria Project Sponsoring Organisations. *Common Criteria for Information Technology Security Evaluation*, August 1999. Version 2.1. Also appears as International Standard ISO/IEC 15408:1999.
5. Karl Crary, Aleksey Kliger, and Frank Pfenning. A Monadic Analysis of Information Flow Security with Mutable State. Technical Report CMU-CS-03-164, School of Computer Science, Carnegie Mellon University, July 2003.
6. Elena Ferrari, Pierangela Samarati, Elisa Bertino, and Sushil Jajodia. Providing Flexibility in Information Flow Control for Object-Oriented Systems. In *Proc. of the IEEE Symposium on Security and Privacy1997*, pages 130–140, Oakland, CA, USA, May 1997. IEEE Computer Society.
7. René Rydhof Hansen. *Flow Logic for Language-Based Safety and Security.* PhD thesis, Technical University of Denmark, 2005.
8. Naoki Kobayashi and Keita Shirane. Type-Based Information Analysis for Low-Level Languages. In *Proc. of Asian Symposium on Programming Languages and Systems, APLAS'02*, pages 302–316, Shanghai, China, November/December 2002.
9. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Verlag, 1999.
10. Pierangela Samarati, Elisa Bertino, Alessandro Ciampichetti, and Sushil Jajodia. Information Flow Control in Object-Oriented Systems. *IEEE Transactions on Knowledge and Data Engineering*, 9(4):524–538, July/August 1997.
11. Igor Siveroni. Operational Semantics of the Java Card Virtual Machine. *Journal of Logic and Algebraic Programming*, 58(1–2):3–25, January–March 2004. Special issue on Formal Methods for Smart Cards.