Non-Interference and Simple Erasure Policies for Java Card Bytecode
(Extended Abstract)

René Rydhof Hansen

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

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. It is therefore very 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 are often required to be formally certified in accordance with one or more rigorous security standards, e.g., the Common Criteria [4], that 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 very hard and doing it manually for even a modestly sized application is infeasible.

In this paper we propose a notion of *non-interference* for an abstract version of the Java Card bytecode language. Programs that have the non-interference property are guaranteed to not leak any secret information. 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 be implemented and used to automatically verify the absence of leaks in a program. Furthermore we argue that our notion of non-interference can be used to also verify certain simple *erasure policies*.

2 The Carmel Core Language

Carmel Core 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 [11, 7] for a specification and discussion of the full Carmel language. We shall use both “Carmel” and “Carmel Core” to denote Carmel Core.

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

Fig. 1. Carmel Core instruction set

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

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

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

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

Fig. 2. Small step semantics for Carmel Core (excerpt)

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 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 a number of instance fields, $\sigma.fields \subseteq \mathsf{Field}$. Each method comprises an instruction for each program counter, $pc \in \mathbb{N}_0$ in the method, $m.instructionAt(pc) \in \mathsf{Instr}$.

The semantics for Carmel Core is defined as a straightforward small step semantics. In Figure 2 an excerpt of the semantics is shown.

3 Non-Interference for Carmel

The notion of secure information flow defined here 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 flexbile enough that it allows 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]. 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 1 (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 in this paper, we consider a simple security lattice with only two levels: low (L) and high (H) with the obvious ordering: $\mathsf{L} \sqsubseteq \mathsf{H}$.

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 3) the map must be an isomorphism on the locations that point to objects containing fields with a low-security classification:

Definition 2 (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 if 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 3 (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*

$$\begin{aligned} &\forall loc_1 \in \mathrm{dom}(H_1) : \forall f \in H_1(loc_1).\mathit{fields} : f.\mathit{level} \sqsubseteq \mathsf{L} \Rightarrow \\ &\quad H_1(loc_1).\mathit{class} = H_2(\pi(loc_1)).\mathit{class} \wedge H_1(loc_1).f \equiv_\pi H_2(\pi(loc_1)).f \end{aligned}$$

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 4 (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 high-security field could have influenced any low-security field. Note that only the heap is needed to achieve the (informal) notion of security discussed previously in this section.

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. In [1] a program transformation that can eliminate such timing leaks is discussed. The second thing to note is the definition of non-interference, and thus security, could be extended to also cover return values and thus implement an aspect of input/output non-interference as well as heap-equivalence.

4 Control and Information Flow Analysis

One of the main problems for an information flow analysis to overcome is to take implicit flows into account and ensure that they are handled correctly. The information flow analysis described below 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 basically 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 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 $\langle H, \langle m, pc, L, S\rangle :: SF\rangle$ let $C.address = (m, pc)$. The post-dominator can then be formally defined in the current setting as follows:

Definition 5 (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* $C_1.address = (m_1, pc_1)$ *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* $C_i.address = (m_1, pc_1')$.

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

Fig. 3. Abstract Domains for the Information Flow Analysis

For lack of space we shall not go into further details here but merely refer to [7] for a formal proof that the above indeed captures the intended intuition and an algorithm for computing the post-dominators in a given Carmel program.

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 3.

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: The least upper bound of the security levels of the possible implicit flows at an address, i.e., $\sqcup\{\ell' \mid (\ell', (m', pc')) \in \hat{C}(m, pc)\}$, is called the *security context* of that address and is written $\sqcup\hat{C}(m, pc)$. 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 $DOM \in \mathsf{Dominators}$:

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

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: $(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{instr}$. Figure 4 shows

an excerpt containing the most interesting Flow Logic judgements. Below the judgements for conditionals and method invocations are discussed in more detail. For the full specification and 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.

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: ${A_1}^{\ell_1} :: {A_2}^{\ell_2} :: X \triangleleft \hat{S}(m, pc)$. Based on the security levels of the stack values and the implicit flows the security level for the current instruction is calculated: $\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \ell_1 \sqcup \ell_2$. 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: ${A_1}^{\ell_1} :: \cdots :: {A_{|m_0|}}^{\ell_{|m_0|}} :: B^{\ell_0} :: X \triangleleft \hat{S}(m, pc) :$. The security levels are then used to calculate the current security context: $\ell \triangleleft \sqcup \hat{C}(m, pc) :$. 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:

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

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

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

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{load}\ t\ x$
iff $\ell \triangleleft \sqcup \hat{C}(m, pc) \sqcup \hat{L}_{\downarrow 2}(m, pc)(x) :$
$\hat{L}_{\downarrow 1}(m, pc)(x)^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$

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

$(\hat{H}, \hat{L}, \hat{S}, \hat{C}; DOM) \models_{\text{IFA}} (m, pc) : \texttt{new}\ \sigma$
iff $\ell \triangleleft \sqcup \hat{C}(m, pc) :$
$\{(\mathsf{Ref}\ \sigma)\}^{\ell} :: \hat{S}(m, pc) \sqsubseteq \hat{S}(m, pc + 1)$
$default_{\ell}(\sigma) \sqsubseteq \hat{H}(\mathsf{Ref}\ \sigma)$
$\hat{L}(m, pc) \sqsubseteq \hat{L}(m, pc + 1)$
$\hat{C}(m, pc) \sqsubseteq_{DOM} \hat{C}(m, pc + 1)$

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

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

Fig. 4. Information Flow Analysis (1)

4.3 Soundness and Non-Interference

Proving that the information flow analysis is semantically sound amounts to showing that it can be used to show that a program is non-interfering in the sense of Definition 4. We prove this by showing 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. For lack of space only the most important definitions and lemmas are stated; 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 6. *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 7 ($\mathcal{A}$-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:*

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

Note that this definition is very similar to that for heap equivalence, cf. Definition 3, 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 5 summarises the lemma and illustrates the source of its name. Note that this proof works on the

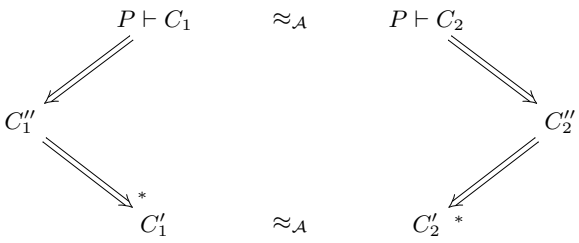

Fig. 5. Diamond property

assumption that programs are bytecode verified and thus that the stack height is fixed for each instruction.

Lemma 1 (Diamond property). *Let* $P \in \mathsf{Program}$, $\mathcal{A} \in \widehat{\mathsf{Analysis}}_{\mathsf{IFA}}$, $C_1, C_2 \in \mathsf{Conf}$ *such that* $\mathcal{A} \models_{IFA} P$ *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* $C_1' \approx_{\mathcal{A}} C_2'$.

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 8 (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 1 and Definition 7.

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.

5 Simple Erasure Policies (work in progress)

Here we present a way to model simple erasure policies using the notion of non-interference defined previously. The material in this section is work-in-progress

and therefore we state no theorems or make any formal proofs. This is left for future work.

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. For Carmel programs such 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 program 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 9 (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* $H_1 \approx_{\mathsf{L}} H_2 \ \Rightarrow\ H_1' \approx_{\mathsf{H}} H_2'$

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 $H_1' \approx_{\mathsf{H}} H_2'$ 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.

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, e.g., through a special `erase`-instruction, before the end of a program or before a method returns. This extension of the analysis is left for future work.

6 Conclusion

We have presented a notion of non-interference for a low-level bytecode language with dynamic memory allocation and argued how this notion can be used to verify certain simple erasure policies.

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. Available for download at `http://www.commoncriteria.org`.
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 and Chris Hankin. A Proposal for the JCVMLe Operational Semantics. SECSAFE-ICSTM-001-2.2, October 2001.