Robust Declassification for Bytecode

Francisco Bavera[1] and Eduardo Bonelli[2]*

[1] UNRC - CONICET, Ruta 36 km 601, Río Cuarto, Argentina (`fbavera@gmail.com`)
[2] UNQ - CONICET, Quilmes, Argentina (`ebonelli@unq.edu.ar`)

Abstract. The noninterference property states that a system is void of insecure information-flows from private to public data. This constitutes too severe a requirement for many systems which allow some form of controlled information release. Therefore, an important effort is being invested in studying forms of intended information declassification. Robust declassification is a property of systems with declassification primitives that ensures that a third party cannot affect the behavior of a system in such a way as to declassify information other than was intended. This paper studies robust declassification for a core fragment of bytecode. We observe that, contrary to what happens in the setting of high-level languages, variable reuse, jumps and operand stack manipulation can be exploited by an attacker to declassify information that was not intended to be. A type system for ensuring robust declassification is introduced and proved to be sound.

Key words: Confidentiality, Information-Flow, Type System, Declassification, Security

1 Introduction

This work is concerned with practical, language-based information flow [SM03] policies for mobile code. Information flow policies in terms of noninterference ensure the absence of information channels from private to public data. Although an intuitively appealing information security policy, it has been recognized to be too strict for practical applications since many applications do allow some form of information declassification or downgrading. Examples are: releasing the average salary from a secret database of salaries for statistical purposes, a password checking program, releasing an encrypted secret, etc. This calls for relaxed notions of noninterference where primitives for information declassification are adopted. At the same time we are also interested in mobile code. Currently a large amount of code is distributed over the Internet largely in the form of bytecode [LY99] or similar low-level languages. Direct verification of security properties of bytecode allows the code consumer to be assured that its security policy is upheld. It should also be stressed that most efforts on information flow target high-level languages yet compilation can produce a program which does not necessarily satisfy the same security properties. Analysis at the bytecode

* Work partially supported by LIFIA and Instituto Tecnológico de Buenos Aires

level has the benefit of independence of the mechanism (compiled, handwritten, code generated) that generated it.

An important issue that arises in mobile code (or any code) that provides a declassification mechanism is whether it can be exploited affecting the data that is declassified or whether declassification happens at all. If this does not arise we declare our declassification mechanism *robust*. We briefly illustrate that with an example from [MSZ04] translated to bytecode.

1. `load` $x_{\mathtt{L}}$
2. `if` 6
3. `load` $z_{\mathtt{H}}$
4. `declassify L`
5. `store` $y_{\mathtt{L}}$
6. $\cdots$

This program releases private information held in $z_{\mathtt{H}}$ to a publicly readable variable $y_{\mathtt{L}}$ if the variable $x_{\mathtt{L}}$ is true, and otherwise has no effect. Here, `declassify` explicitly indicates a release from secret to public of the level of the topmost element on the stack. As a consequence, this program does not enjoy noninterference. However, we can observe something more on the reliability of the declassification mechanism. Indeed, an attacker can affect whether information is declassified at all (assuming she can change the value of $x_{\mathtt{L}}$) or can affect the information that is declassified (assuming she can change the value of $z_{\mathtt{H}}$). This program is therefore declared not to be robust under any of these two assumptions.

The starting point of this paper is a core fragment of bytecode that includes a declassification instruction `declassify` κ, where κ is a security level (see Sec. 3). The effect of this instruction is to declassify the top element of the operand stack of the JVM run-time system. Our language allows flow sensitivity [HS06] w.r.t. the security level of the variables (i.e. local variables may be reused with different security levels, as in standard bytecode). Under an attacker that observes and manipulates data, we address the requirements that programs must meet in order for robustness to be enjoyed.

Contributions. We study the notion of robust declassification of a low-level language such as bytecode and reveal subtle ways in which, in low-level code, the standard notion of robust declassification fails. Particularly, this fails in the presence of flow-sensitive types and low-level stack manipulation primitives. We propose a type system for enforcing robust declassification in bytecode and prove that this system is sound.

Structure of the paper. Sec. 2 introduces motivating examples. It is followed by the attacker model and the bytecode language. The type system is presented in Sec. 4. Sec. 5 addresses noninterference for confidentiality and robust declassification. Finally, we conclude and suggest further avenues for research. For further details and full proofs please consult [BB09].

2 Motivating examples

In general, an active attacker may change system behavior by injecting new code in the program. Such attacker-controlled low-integrity computation may be interspersed with trusted high-integrity code. To distinguish the two, the high-integrity code is represented as a program in which some statements are missing, replaced by holes ($\bullet$). The idea is that the holes are places where the attacker can insert arbitrary low-integrity code. There may be multiple holes (possibly none) in the high-integrity code, represented by the notation $\overrightarrow{\bullet}$. Such expressions are called *high-integrity contexts*, written $B[\overrightarrow{\bullet}]$, and formally defined as programs in which the set of bytecode instructions is extended with a distinguished symbol "$\bullet$". These holes can be replaced by a vector of attacker fragments $\overrightarrow{a}$ to obtain a complete method $B[\overrightarrow{a}]$. An attacker is thus a vector of such code fragments. A passive attacker is an attack vector that fills all holes with the low-integrity code from the original program. An active attacker is any attack vector that fills some hole in a way that changes the original program behavior.

One way an attacker can cause information leak is to directly violate confidentiality and integrity (eg. using declassify in its own code). However, one is interested in how an attacker can exploit insecure information flow in the trusted code. Therefore, we restrict our attention to attacks that are *active* in this sense (cf. Def. 2). We discuss some examples[3].

```
1. load x
2. load y
3. [•]
4. declassify L
5. ···
```
(a)

```
1. [•]
2. load x
3. declassify L
4. ···
```
(b)

```
1. [•]
2. load x
3. declassify L
4. return
```
(c)

```
1. [•]
2. load   x
3. if   5
4. declassify L
5. ···
```
(d)

In (a), assuming x and y are high-integrity at line 4, the value of y is declassified. If the attacker consists of a `pop` instruction, then declassification of the value of x is forced. Likewise the attacker could push an element on the stack rather than remove it. In this case it is the newly pushed element that is declassified. Let us consider another example. In (b) the security level of x at 2 is assumed to be confidential. If the attacker inserts `load` y; `goto` 3 where y holds some high confidentiality data, then by affecting the control-flow y is now declassified. Finally, if the code is that of (c), then by inserting the code `load` y; `return` declassification of x is avoided.

One final consideration is that we must also restrict variable reuse in low-integrity contexts. Consider the code excerpt of (d) where x is assumed high-integrity at 2 and y low-integrity at 2. Line 4 attempts to declassify depending on the value of x (a high-integrity value). Suppose the attacker reuses x with a low-integrity value, such as in `load` y; `store` x. Then in line 4 a declassification is attempted depending on the value of the variable y (a low-integrity

[3] In these examples, for expository purposes, the argument of `declassify` is `L`. However these labels are actually pairs of levels (cf. Sec. 3).

value). Therefore the attacker can manipulate the condition in line 3 which decides whether declassification is performed or not. This situation is avoided by restricting attackers from manipulating high-integrity variables such as x above.

Our definition of *active attack* thus requires attacks to not include `return` nor `declassify` instructions. Likewise, we require all jumps to be local to the attackers code. The size of the stack before the attacker code is executed should be the same after it is executed. Finally, high-integrity variables and stack elements should not be modifiable. These observations are made precise in Def. 2 and the typing schemes for bytecode (Sec. 4).

3 Attacker Model and the Bytecode Language

Since robustness of a system depends on the power an attacker has to affect the execution of the system we follow [MSZ04] and introduce *integrity* labels in our model. High-integrity information and code are trusted and assumed not to be affected by the attacker while low-integrity information and code are untrusted and assumed to be under the control of the attacker. Both confidentiality and integrity labels are merged into a single lattice of *security levels*.

We assume given a lattice of confidentiality levels $\mathcal{L}_C$ with ordering $\preceq_C$ and a lattice of integrity levels $\mathcal{L}_I$ with ordering $\preceq_I$. We use $\bot_C$ and $\top_C$ for the bottom and top elements of $\mathcal{L}_C$ and similarly for those of $\mathcal{L}_I$. We write l_C for elements of the former and l_I for elements of the latter and $\sqcup_C$ and $\sqcup_I$, resp., for the induced join. A *security level* is a pair (l_C, l_I). These pairs are elements of a lattice $\mathcal{L}$ and are typically denoted using letters $\kappa, \lambda, \kappa_i, \lambda_i, \ldots$. The partial order $\preceq$ on $\mathcal{L}$ is defined as: $(l_C, l_I) \preceq (l'_C, l'_I)$ iff $l_C \preceq_C l'_C$ and $l_I \preceq_I l'_I$. The join on security levels is: $(l_C, l_I) \sqcup (l'_C, l'_I) = (l_C \sqcup_C l'_C, l_I \sqcup_I l'_I)$. The projections of the first and second component of a security level are $\mathsf{C}((l_C, l_I)) = l_C$ and $\mathsf{I}((l_C, l_I)) = l_I$. Also, we occasionally write C_κ for $\mathsf{C}(\kappa)$ and I_κ for $\mathsf{I}(\kappa)$.

The power of an attacker is described by a security level $\mathtt{A}$, where $\mathsf{C}(\mathtt{A})$ is the confidentiality level of data the attacker is expected to be able to read, and $\mathsf{I}(\mathtt{A})$ is the integrity level of data or code that the attacker is expected to be able to affect. Thus, the robustness of a system is w.r.t. the attacker security level $\mathtt{A}$.

The attacker security level $\mathtt{A}$ determines both high and low-confidentiality ($\mathtt{H}_C = \{\kappa \mid \mathsf{C}(\kappa) \not\preceq \mathsf{C}(\mathtt{A})\}$ and $\mathtt{L}_C = \{\kappa \mid \mathsf{C}(\kappa) \preceq \mathsf{C}(\mathtt{A})\}$) areas and high and low-integrity areas ($\mathtt{H}_I = \{\kappa \mid \mathsf{I}(\mathtt{A}) \not\preceq \mathsf{I}(\kappa)\}$ and $\mathtt{L}_I = \{\kappa \mid \mathsf{I}(\mathtt{A}) \preceq \mathsf{I}(\kappa)\}$) in the security lattice $\mathcal{L}$. We can identify four key areas of the lattice: (1) $\mathtt{LH} \simeq \mathtt{L}_C \cap \mathtt{H}_I$; (2) $\mathtt{HH} \simeq \mathtt{H}_C \cap \mathtt{H}_I$; (3) $\mathtt{LL} \simeq \mathtt{L}_C \cap \mathtt{L}_I$; and (4) $\mathtt{HL} \simeq \mathtt{H}_C \cap \mathtt{L}_I$. The $\mathtt{LL}$ region is under complete control of the attacker: she can read and modify data in this region. Data in the $\mathtt{LH}$ region may be read but not modified by the attacker. The attacker may not read nor modify data in the region $\mathtt{HH}$. Finally, data in the $\mathtt{HL}$ region may not be inspected but can be modified by the attacker.

The Bytecode Language Let $\mathbb{N}$ stand for the natural numbers and $\mathbb{X}$ for the set of local variables whose elements we refer to by x, y. A *program* B is a sequence of the following bytecode instructions where op is $+$ or $\times$, $x \in \mathbb{X}$, $j \in \mathbb{N}$:

`push` j	push j on stack	`pop`	pop from stack
`load` x	load value of x on stack	`store` x	store top of stack in x
`prim` op	primitive operation	`ifeq` j	conditional jump
`goto` j	unconditional jump	`return`	return
`declassify` κ	declassifies top of stack		

These instructions are standard except for `declassify` κ which *declassifies* the security level of value on the top of the operand stack. It does not have any effect on the state of the program, as described below. We write $\mathsf{Dom}(B)$ for the set of *program points* of B and $B(i)$, with $i \in 1..n$ and n the length of B, for the i^{th} instruction of B.

We write $\mathbb{V}$ for the set of *values* whose elements are denoted by v. Values are taken to be the set of integers. *Machine states* are tuples $\langle i, \alpha, \sigma \rangle$, where $i \in \mathbb{N}$ is the program counter that points to the next instruction to be executed; α, (local variable array) is a mapping from local variables to values; σ (stack) is an operand stack. A machine state *for* B is one in which the program counter points to an element in $\mathsf{Dom}(B)$. If $\langle i, \alpha, \sigma \rangle$, then we define $\mathtt{pc}(s) = i$.

The small-step operational semantics is standard (cf. Appendix). We write $s_1 \longrightarrow_B s_2$ if both s_1, s_2 are states for B and s_2 results by one step of reduction from s_1. In the sequel we work with states for some program B which shall be clear from the context and thus simply speak of states without explicit reference to B. Expressions of the form $\langle v \rangle$ are referred to as *final states.* A non-final state s may either reduce to another non-final state or to a final state. Initial states take the form $\langle 1, \alpha, \epsilon \rangle$, where ϵ denotes the empty stack. We write $\twoheadrightarrow_B$ for the reflexive-transitive closure of $\longrightarrow_B$. The trace $\mathsf{Tr}_B(\langle i, \alpha, \sigma \rangle)$ of the execution from state $\langle i, \alpha, \sigma \rangle$ ($i \in \mathsf{Dom}(B)$) is the sequence $\langle i, \alpha, \sigma \rangle \longrightarrow_B \langle i_1, \alpha_1, \sigma_1 \rangle \longrightarrow_B \langle i_2, \alpha_2, \sigma_2 \rangle \longrightarrow_B \langle i_3, \alpha_3, \sigma_3 \rangle \longrightarrow_B \cdots$.

4 Type System

A *method type* is an expression of the form $(x_1 : \kappa_1, \ldots, x_n : \kappa_n, \kappa_r)$, abbreviated $(\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r)$, where $x_1, \ldots, x_n$ are the formal parameters together with their security levels and κ_r is the return value security level. A *method* $M[\overrightarrow{\bullet}]$ is a pair $((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])$, where $B[\overrightarrow{\bullet}]$ is a program referred to as the method's body. Methods $M[\overrightarrow{\bullet}]$ are typed by means of a *typing judgment*: $\mathcal{V}, \mathcal{S}, \mathcal{A} \rhd M[\overrightarrow{\bullet}]$. Each of $\mathcal{V}, \mathcal{S}$ and $\mathcal{A}$ are called *typing contexts.* Typing contexts supply information needed to type each instruction of $M[\overrightarrow{\bullet}]$. As such they constitute families of either functions ($\mathcal{V}, \mathcal{S}$) or security levels ($\mathcal{A}$) indexed by a finite set of instruction addresses. We write $\mathcal{V}_i$ to refer to the i^{th} element of the family and similarly with the others. The first typing context assigns a *variable array type* V to each instruction (of $M[\overrightarrow{\bullet}]$). A variable array type is a function assigning security labels to each local variable. Thus $\mathcal{V}_i$ indicates the types of the local variables at instruction i. The $\mathcal{S}$ typing context assigns a *stack type* S to each instruction.

A stack type is a function assigning security labels to numbers from 1 to n, where n is assumed to be the length of the stack. $\mathcal{A}$ indicates the level of each instruction. If $\mathcal{A}(i)$ is high at some program point because $\mathsf{C}(\mathcal{A}(i)) \in \mathtt{H}_C$, then an attacker might learn secrets from the fact that execution reached that point. If $\mathcal{A}(i)$ is high because $\mathsf{I}(\mathcal{A}(i)) \in \mathtt{L}_I$, the attacker might be able to affect whether control reaches that point.

Two further ingredients are required before formulating the type system. The first is a notion of subtyping for variable array and stack types.

$$\frac{\forall x \in \mathbb{X}. V'(x) \preceq V''(x)}{V' \leq V''} \qquad \frac{\begin{array}{c}|S'| = |S''| = n \\ \forall j \in 1..n. \begin{cases} S'(j) \preceq S''(j) \text{ if } \kappa \preceq S'(j), \\ S'(j) = S''(j) \text{ otherwise} \end{cases}\end{array}}{S' \leq_\kappa S''}$$

Frame types are compared pointwise using the ordering on security labels. We write $|S|$ for the length of S. $S' \leq_\kappa S''$ asserts that stack type S' is a subtype of S'' at level κ. Stack types may only be compared if they are of the same size. Furthermore, we allow depth subtyping at position j provided that $S'(j)$ is at least κ. The other ingredient is the availability of *control dependence regions* [BR05,BPR07].

Control Dependence Region High-level languages have control-flow constructs that explicitly state dependency. For example, from `while x begin c1;c2 end` one easily deduces that both `c1` and `c2` depend on `x`. Given that such constructs are absent from bytecode, as in most low-level languages, and that they may be the source of unwanted information flows, our type system requires this information to be supplied. Let $\mathsf{Dom}(B)^\sharp$ denote the set of program points of $B[\overrightarrow{\bullet}]$ where branching instructions occur (i.e. $\mathsf{Dom}(B)^\sharp = \{k \in \mathsf{Dom}(B) \mid B[\overrightarrow{\bullet}](k) = \mathtt{if}\ i\}$). We assume given two functions ($\wp$ below denotes the powerset operator): (1) region: $\mathsf{Dom}(B)^\sharp \to \wp(\mathsf{Dom}(B))$ and (2) jun: $\mathsf{Dom}(B)^\sharp \to \mathsf{Dom}(B)$.

The first computes the *control dependence region*, an over-approximation of the range of branching instructions and the second the unique junction point of a branching instruction at a given program point. We only require some abstract properties of these functions to hold [BPR07]. We need the notion of successor relation to formulate these properties.

Definition 1. *The successor relation* $\mapsto \subseteq \mathsf{Dom}(B) \times \mathsf{Dom}(B)$ *of a method* $B[\overrightarrow{\bullet}]$ *is defined by the clauses: (1) If* $B[\overrightarrow{\bullet}](i) = \mathtt{goto}\ k$*, then* $i \mapsto k$*; (2) If* $B[\overrightarrow{\bullet}](i) = \mathtt{if}\ k$*, then* $i \mapsto i+1$ *and* $i \mapsto k$ *; (3) If* $B[\overrightarrow{\bullet}](i) = \mathtt{return}$*, then* $i \not\mapsto$ *(i has no successors); and (4) Otherwise,* $i \mapsto i+1$*.*

A small set of requirements, the *safe over approximation property* or *SOAP*, on region and jun suffice for the proof of noninterference. They state some basic properties on how the successor relation, region and jun are related.

Property 1. Let $i \in \mathsf{Dom}(B)^\sharp$. (1) If $i \mapsto i'$, then $i' \in \mathsf{region}(i)$. (2) If $i' \mapsto i''$ and $i' \in \mathsf{region}(i)$, then $i'' \in \mathsf{region}(i)$ or $i'' = \mathsf{jun}(i)$. (3) If $i \mapsto^* i'$ and $i' \in$

$\mathsf{Dom}(B)^\sharp \cap \mathsf{region}(i)$, then $\mathsf{region}(i') \subseteq \mathsf{region}(i)$. (4) If $\mathsf{jun}(i)$ is defined, then $\mathsf{jun}(i) \notin \mathsf{region}(i)$ and $\forall i'$ such that $i \mapsto^* i'$, either $i' \mapsto^* \mathsf{jun}(i)$ or $\mathsf{jun}(i) \mapsto^* i'$.

Intuitively, successors i' of a branching instruction i should be in the same region. As for successors i'' of an instruction i' with $i' \in \mathsf{region}(i)$, either i'' should be in the region associated to i or be a junction point for this region. Regarding the third item we write $\mapsto^*$ for the reflexive, transitive closure of $\mapsto$ and say that i' is an *indirect successor* of i when $i \mapsto^* i'$. This item reads as follows: if i' is the program point of a branching instruction that is an indirect successor of a branching instruction i that has not left the region of i, then the region of i' is wholly inside that of i. The final item states that the junction point of a region are not part of this region and, moreover, all indirect successors of the program point i of a branching instruction either come before or after (in terms of the indirect successor relation) the junction point of the region of i.

In what follows we shall adopt two assumptions that, without sacrificing expressiveness, simplify our analysis. First we assume a method body $B[\overrightarrow{\bullet}]$ has a unique `return` instruction. Second, that predecessors of a junction point are always `goto` instructions (i.e. $i \in \mathsf{region}(k)$ and $i \mapsto i'$ and $i' = \mathsf{jun}(k)$ implies $B(i) = \texttt{goto}\ i'$). Note that this last assumption also helps us guarantee Property 1(1).

4.1 Typing Schemes

We assume throughout this work that we are dealing with valid JVM bytecode in the sense that it passes the bytecode verifier. For example, in an instruction such as `prim` $+$ we do not check whether the operands are indeed numbers. A *program* $B[\overrightarrow{\bullet}]$ is *well-typed* under method type $(\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r)$ if there exist typing contexts $\mathcal{V}$, $\mathcal{S}$ and $\mathcal{A}$ such that the type judgment $\mathcal{V}, \mathcal{S}, \mathcal{A} \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])$, defined below, holds.

$$\frac{\forall x_j \in \boldsymbol{x}.\mathcal{V}_1(x_j) = \kappa_j \quad \forall i \in \mathsf{Dom}(B).\mathcal{V}, \mathcal{S}, \mathcal{A}, i \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])}{\mathcal{V}, \mathcal{S}, \mathcal{A} \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])}$$

The first requirement is that the variable array type $\mathcal{V}_1$ provide security levels for all parameters and must agree with the ones assigned to each of them by the declaration $\boldsymbol{x} : \boldsymbol{\kappa}$. Also, all program points in $B[\overrightarrow{\bullet}]$ must type as the second requirement indicates. A sample of the typing schemes defining this second requirement are given in Fig. 1 (cf. appendix for the full set) and are described below. Note that there may be more than one set of typing contexts $\mathcal{V}$, $\mathcal{S}$ and $\mathcal{A}$ such that a program is well-typed. For reasons of technical convenience we assume, without loss of generality, that the chosen $\mathcal{A}$ is minimal such that $\forall i \in \mathsf{Dom}(B).\mathcal{A}(1) \preceq \mathcal{A}(i)$. A *method* $((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])$ is *well-typed* if $B[\overrightarrow{\bullet}]$ is a well-typed program under $(\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r)$ with the additional assumption that the stack type is empty, $\mathcal{S}_1 = \epsilon$.

We now briefly describe the typing schemes. They introduce a set of constraints between the frames type and stacks type at different instructions in the program. We write $\mathcal{S}_{i+1}(0)$ for the topmost element of the stack type $\mathcal{S}_{i+1}$ and

$$\text{(T-If)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \texttt{if}\ i' \\ \mathcal{V}_i \leq \mathcal{V}_{i+1}, \mathcal{V}_{i'} \\ \mathcal{S}_i \leq_{\mathcal{A}(i)} \kappa \cdot \mathcal{S}_{i+1} = \kappa \cdot \mathcal{S}_{i'} \\ \mathcal{S}_i(0) \in \mathtt{L}_I,\ \textit{if}\ \mathcal{A}(i) \in \mathtt{L}_I \\ \forall k \in \texttt{region}(i).\kappa \preceq \mathcal{A}(k) \\ i+1, i' \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, i \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])}$$

$$\text{(T-Str)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \texttt{store}\ x \\ \mathcal{S}_i \leq_{\mathcal{A}(i)} \mathcal{V}_{i+1}(x) \cdot \mathcal{S}_{i+1} \\ \mathcal{V}_i \backslash x \leq \mathcal{V}_{i+1} \backslash x \\ \mathcal{A}(i) \preceq \mathcal{V}_{i+1}(x) \\ \mathcal{S}_i(0), \mathcal{V}_i(x) \in \mathtt{L}_I,\ \textit{if}\ \mathcal{A}(i) \in \mathtt{L}_I \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, i \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])}$$

$$\text{(T-Declassify)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = \texttt{declassify}\ \kappa \\ (\mathcal{S}_i \backslash 0) \leq_{\mathcal{A}(i)} (\mathcal{S}_{i+1} \backslash 0) \\ \mathcal{A}(i) \sqcup \kappa \preceq \mathcal{S}_{i+1}(0) \\ \mathsf{I}(\mathcal{S}_i(0)) = \mathsf{I}(\kappa) \\ \mathsf{I}(\mathcal{A}(i)), \mathsf{I}(\mathcal{S}_i(0)) \in \mathtt{H}_I \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, i \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])}$$

$$\text{(T-Hole)}\quad \frac{\begin{array}{l} B[\overrightarrow{\bullet}](i) = [\bullet] \\ \mathcal{A}(i) \in \mathtt{L}_C \\ \mathcal{S}_i \leq_{(\bot_C, \mathsf{I}(\mathtt{A}))} \mathcal{S}_{i+1} \\ \mathcal{V}_i \leq \mathcal{V}_{i+1} \\ i+1 \in \mathsf{Dom}(B) \end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A}, i \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])}$$

Fig. 1. Typing Schemes

$\mathcal{S}_{i+1} \backslash 0$ for $\mathcal{S}_{i+1}$ without the topmost element. Subtyping (rather than equality) for variable array and stack types are required given that instruction $i+1$ may have other predecessors apart from i. T-Store, since instruction `store` x does change local variables, $\mathcal{V}_i \backslash x$ must be a subtype of $\mathcal{V}_{i+1} \backslash x$ as stated by the third line. Furthermore, the value of local variable x is modified by stack top, then $\mathcal{S}_i \leq \mathcal{V}_{i+1} \cdot \mathcal{S}_{i+1}$ must hold and . The four line states that the security level of the x value in follow instruction must be greater than or equal to the security level of current address. Regarding T-If, we assume that `if` i' instructions have always two branches (in this case i' and $i+1$). T-If requires $\mathcal{V}_{i'} = \mathcal{V}_{i+1}$ must hold. Since control jumps to i' or $i+1$, $\mathcal{V}_i$ must be a subtype of $\mathcal{V}_{i'}$ and $\mathcal{V}_{i+1}$ as stated by the second line. $\mathcal{S}_i$ must be a subtype of $\kappa \cdot \mathcal{S}_{i'}$ and $\kappa \cdot \mathcal{S}_{i+1}$. Moreover, the security level of $\mathsf{region}(i)$ must be greater than or equal to the security level κ of the value inspected at i. T-Declassify states that only high-integrity data is allowed to be declassified and that declassification might only occur at a high-integrity security environment. Because the possible flow origins is within the high-integrity area of the lattice, the attacker (who can only affect the low-integrity area of the lattice) cannot influence uses of the declassification mechanism and therefore cannot exploit it. Regarding T-Hole, the second condition deserves a comment. For robustness, it is important that holes not be placed at program points where the type environment is high-confidentiality, because then the attacker would be able to directly observe implicit flows to the hole and could therefore cause information to leak even without a `declassify`.

As a final comment, note that these schemes disable high integrity variable reuse and popping of high integrity values off the stack in low integrity contexts. These constraints are necessary for proving that the type system guarantee robustness.

Now that the type system is in place and our motivating examples have been introduced we can formalize the notion of an active attack.

Definition 2 (Active attack). *A program a is an active attack under method type $(\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r)$ if a does not contain* `declassify` *or* `return` *instructions and there exist typing contexts $\mathcal{V}$, $\mathcal{S}$ and $\mathcal{A}$ such that the following judgment holds:*

$$\frac{\begin{array}{c}\forall x_j \in \boldsymbol{x}.\mathcal{V}_1(x_j) = \kappa_j \qquad \forall i \in \mathsf{Dom}(a).\mathcal{V}, \mathcal{S}, \mathcal{A}, i \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), a) \\ \mathcal{A}(1) = (\bot_C, \mathsf{I}_{\mathtt{A}}) \quad |\mathcal{S}_1| = |\mathcal{S}_{exit}|\end{array}}{\mathcal{V}, \mathcal{S}, \mathcal{A} \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), a)}$$

The index *exit* in $\mathcal{S}_{exit}$ refers to the program point $n + 1$ where n is the size of a. We assume that execution always ends at this program point. Also, as in the definition of well-typed programs, we assume the chosen $\mathcal{A}$ verifies $\forall i \in \mathsf{Dom}(a).\mathcal{A}(1) \preceq \mathcal{A}(i)$.

5 Soundness

The type system enforces two interesting properties: noninterference and robust declassification. Noninterference states that any two (terminating) runs of a well-typed method, that does not use `declassify`, starting from initial states that coincide on public input data produce final states in which public output data are identical. This relation on states, called *indistinguishability*, is formalized in Sec. 5.1 and we then give the structure of the proof of noninterference in Sec. 5.2. Robust declassification states that an attacker may not manipulate the declassification mechanism to leak more information than intended. This property applies to programs that may contain `declassify` instructions. Sec. 5.3 introduces the precise definition and proves the main result of this paper.

5.1 Indistinguishability

Indistinguishability for states at some level λ is formulated by considering each of its components at a time (values, local variable assignments and stacks). States are declared indistinguishable depending on what may be observed and this, in turn, depends on the level κ of the observer. For further discussion on these notions please consult [BB08,BB09].

Definition 3 (Value indistinguishability). *Given values v_1, v_2 and security levels κ, λ we define $v_1 \sim^{\lambda}_{\kappa} v_2$ ("values v_1 and v_2 are indistinguishable at level λ w.r.t. observer level κ") to hold if either (1) $\kappa \preceq \lambda$ and $v_1 = v_2$; or (2) $\kappa \not\preceq \lambda$.*

Definition 4 (Local variable assignment indistinguishability). *Let α_1, α_2 be local variable assignments, V_1, V_2 frame types and κ, λ security levels. We write $\alpha_1 \sim^{\lambda}_{(V_1,V_2),\kappa} \alpha_2$ (or $\alpha_1 \sim^{\lambda}_{V_1,\kappa} \alpha_2$ when $V_1 = V_2$ and $\kappa \preceq \lambda$).*

- *Low-indist. assignments ($\kappa \preceq \lambda$ and $V_1 = V_2$). α_1 and α_2 are considered low-indist. at level λ if for all $x \in \mathbb{X}$, $\alpha_1(x) \sim^{\lambda}_{V_1(x)} \alpha_2(x)$.*
- *High-indist. assignments ($\kappa \not\preceq \lambda$). α_1 and α_2 are considered high-indist. at level λ if for all $x \in \mathbb{X}$, $\alpha_1(x) \sim^{\lambda}_{V_1(x) \sqcup V_2(x)} \alpha_2(x)$.*

Definition 5 (Stack indistinguishability). *Let σ, σ' be stacks, S, S' stack types and λ, κ security levels. We write $\sigma \sim^{\lambda}_{(S,S'),\kappa} \sigma'$ (or $\sigma \sim^{\lambda}_{S,\kappa} \sigma'$, when $S = S'$ and $\kappa \preceq \lambda$).*

– *Low-indist. stacks ($\kappa \preceq \lambda$).*

$$\frac{\kappa \preceq \lambda}{\epsilon \sim^{\lambda}_{\epsilon,\kappa} \epsilon} \qquad \frac{\sigma \sim^{\lambda}_{S,\kappa} \sigma' \quad v \sim^{\lambda}_{\kappa'} v' \quad \kappa \preceq \lambda}{v \cdot \sigma \sim^{\lambda}_{\kappa' \cdot S,\kappa} v' \cdot \sigma'}$$

– *High-indist. stacks ($\kappa \not\preceq \lambda$).*

$$\frac{\sigma \sim^{\lambda}_{S,\kappa'} \sigma' \quad \kappa' \preceq \lambda \quad \kappa \not\preceq \lambda}{\sigma \sim^{\lambda}_{(S,S),\kappa} \sigma'} \quad \frac{\sigma \sim^{\lambda}_{(S,S'),\kappa} \sigma' \quad \kappa', \kappa \not\preceq \lambda}{v \cdot \sigma \sim^{\lambda}_{(\kappa' \cdot S,S'),\kappa} \sigma'} \quad \frac{\sigma \sim^{\lambda}_{(S,S'),\kappa} \sigma' \quad \kappa', \kappa \not\preceq \lambda}{\sigma \sim^{\lambda}_{(S,\kappa' \cdot S'),\kappa} v \cdot \sigma'}$$

Two states are indistinguishable if they are either both executing in a high region (in which case we say they are *high-indistinguishable*) or they are both executing the same instruction in a low region (*low-indistinguishable*). Furthermore, in any of these cases the frame types and operand stacks should also be indistinguishable.

Definition 6 (Machine state indistinguishability). *Given security level λ, $\langle i, \alpha, \sigma \rangle \sim^{\lambda} \langle i', \alpha', \sigma' \rangle$ holds iff (1) $\mathcal{A}(i), \mathcal{A}(i') \not\preceq \lambda$ or ($\mathcal{A}(i) = \mathcal{A}(i') \preceq \lambda$ and $i = i'$); (2) $\alpha \sim^{\lambda}_{(\mathcal{V}_i,\mathcal{V}_{i'}),\mathcal{A}(i)} \alpha'$; and (3) $\sigma \sim^{\lambda}_{(\mathcal{S}_i,\mathcal{S}_{i'}),\mathcal{A}(i)} \sigma'$.*

5.2 Noninterference

Henceforth we assume $M[\overrightarrow{a}]$ to be well-typed: $\mathcal{V}, \mathcal{S}, \mathcal{A} \rhd ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{a}])$. We let $s_1 = \langle i, \alpha_1, \sigma_1 \rangle$ and $s_2 = \langle i, \alpha_2, \sigma_2 \rangle$ be states for $M[\overrightarrow{a}]$. Furthermore, let λ be a security level.

Definition 7. *$M[\overrightarrow{a}]$ satisfies noninterference, written $\mathsf{NI}(M[\overrightarrow{a}])$, if for every $\kappa \preceq \mathtt{A}$, α_1, α_2 variable arrays and v_1, v_2 values: (1) $\langle 1, \alpha_1, \epsilon \rangle \longrightarrow^{*} \langle v_1 \rangle$, (2) $\langle 1, \alpha_2, \epsilon \rangle \longrightarrow^{*} \langle v_2 \rangle$ and (3) $\alpha_1 \sim^{\mathtt{A}}_{\mathcal{V}_1,\kappa} \alpha_2$ imply $v_1 \sim^{\mathtt{A}}_{\kappa_r} v_2$.*

Our notion of noninterference assumes the attacker has the ability to see the initial low variables value, and its final result (variant of termination-insensitive information flow). We do not address other covert channels (e.g. termination, timing, or abstraction-violation attacks) in this type system.

Proposition 1. *All well-typed methods $M[\overrightarrow{a}]$, without `declassify` instructions, satisfy $\mathsf{NI}(M[\overrightarrow{a}])$.*

Prop. 1 is a consequence of a stronger property (Prop. 2) that we state below. This latter property is used in our result on robustness (Prop. 3). First we introduce some definitions. The λ-projection of a trace t, written $t|_{\lambda}$, where $t = \langle i_0, \alpha_{i_0}, \sigma_{i_0} \rangle \longrightarrow_{B[\overrightarrow{a}]} \langle i_1, \alpha_1, \sigma_1 \rangle \longrightarrow_{B[\overrightarrow{a}]} \cdots$, is the sequence of states $\langle i_0, \alpha_{j_0}, \sigma_{j_0} \rangle \langle j_1, \alpha_{j_1}, \sigma_{j_1} \rangle \langle j_2, \alpha_{j_2}, \sigma_{j_2} \rangle \langle j_3, \alpha_{j_3}, \sigma_{j_3} \rangle \cdots$ s.t. $0 < j_1 < j_2 < \cdots$ and for all i_k, if $\mathcal{A}(i_k) \preceq \lambda$ then $i_k = j_n$, for some n. We define two finite traces t_1

and t_2 of the same length n to be indistinguishable, written $t_1 \sim^\lambda t_2$, iff their λ-projections are indistinguishable $t_1|_\lambda \sim^\lambda t_2|_\lambda$. This latter property holds iff for all $k \in 0..n$, $\langle i_k, \alpha_{i_k}, \sigma_{i_k}\rangle \sim^\lambda \langle j_k, \alpha_{j_k}, \sigma_{j_k}\rangle$ where $t_1|_\lambda = \langle i_0, \alpha_{i_0}, \sigma_{i_0}\rangle \cdots \langle i_n, \alpha_{i_n}, \sigma_{i_n}\rangle$ and $t_2|_\lambda = \langle j_0, \alpha_{j_0}, \sigma_{j_0}\rangle \cdots \langle j_n, \alpha_{j_n}, \sigma_{j_n}\rangle$.

Proposition 2. *Suppose $M[\overrightarrow{a}]$ has no `declassify` instructions and s_1 and s_2 are initial states such that $s_1 \sim^\lambda s_2$. Then $\mathsf{Tr}_{B[\overrightarrow{a}]}(s_1) \sim^\lambda \mathsf{Tr}_{B[\overrightarrow{a}]}(s_2)$.*

The proof relies on three *unwinding lemmas.*

Lemma 1. *Suppose (1) $\mathcal{A}(i) \preceq \lambda$; (2) $s_1 \longrightarrow_{B[\overrightarrow{a}]} s_1'$; (3) $s_2 \longrightarrow_{B[\overrightarrow{a}]} s_2'$; and (4) $s_1 \sim^\lambda s_2$. Moreover, assume $B[\overrightarrow{a}]$ does not contain `declassify`. Then (1) either $s_1' = \langle i_1', \alpha_1', \sigma_1'\rangle$, $s_2' = \langle i_2', \alpha_2', \sigma_2'\rangle$ and $s_1' \sim^\lambda s_2'$; or (2) $s_1' = \langle v_1'\rangle$, $s_2' = \langle v_2'\rangle$ and $v_1' \sim^\lambda_{\kappa_r} v_2'$.*

Lemma 2. *Let $i_1, i_2 \in \mathsf{region}(k)$ for some k such that $\forall k' \in \mathsf{region}(k) : \mathcal{A}(k') \not\preceq \lambda$. Furthermore, suppose: (1) $s_1 \sim^\lambda s_1'$; (2) $s_1 \longrightarrow_{B[\overrightarrow{a}]} s_2$; (3) $s_2 = \langle i_1', \alpha_1', \sigma_1'\rangle$; (4) and $i_1' \in \mathsf{region}(k)$. Then $s_2 \sim^\lambda s_1'$.*

Lemma 3. *Let $i_1, i_2 \in \mathsf{region}(k)$ for some k such that $\forall k' \in \mathsf{region}(k) : \mathcal{A}(k') \not\preceq \lambda$. Suppose: (1) $s_1 \longrightarrow_{B[\overrightarrow{a}]} s_1'$ and $s_2 \longrightarrow_{B[\overrightarrow{a}]} s_2'$; (2) $s_1 \sim^\lambda s_2$, f; and (3) $s_1' = \langle i_1', \alpha_1', \sigma_1'\rangle$ and $s_2' = \langle i_2', \alpha_2', \sigma_2'\rangle$; (4) and $\mathcal{A}(i_1') = \mathcal{A}(i_2')$ and $\mathcal{A}(i_1') \preceq \lambda$. Then $i_1' = \mathsf{jun}(k) = i_2'$ and $s_1' \sim^\lambda s_2'$.*

Remark 1. Noninterference may be proved for any program which is well-typed rather than just well-typed methods: the requirement that execution begin at a state where the stack is empty is not necessary for the result to hold.

5.3 Robustness

Definition 8 (Robustness). *Let $M[\overrightarrow{\bullet}] = ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])$ be a well-typed method. We say that it satisfies robustness with respect to active attacks at level $\mathtt{A}$ if for every s_1, s_1' initial states and $\overrightarrow{a_1}$, $\overrightarrow{a_2}$ active attacks:*

$$\mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1) \sim^{(\mathtt{C}(\mathtt{A}), \top_I)} \mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1') \textit{ implies } \mathsf{Tr}_{B[\overrightarrow{a_2}]}(s_1) \sim^{(\mathtt{C}(\mathtt{A}), \top_I)} \mathsf{Tr}_{B[\overrightarrow{a_2}]}(s_1')$$

Robust declassification holds if the attacker's observations from the execution of $M[\overrightarrow{a_2}]$ may not reveal any secrets apart from what the attacker already knows from observations about the execution of $M[\overrightarrow{a_1}]$. The main result of this work is therefore:

Proposition 3. *All well-typed methods satisfy robustness.*

In order to address the proof two auxiliary results are required. The first (Lem. 4) states that an active attack cannot consult or modify high-integrity data on the stack and the second (Lem. 8) extends this observation to the states of a trace.

Lemma 4. *An active attack: (1) cannot consult or modify high-integrity data on the stack; and (2) satisfies noninterference.*

The proof of the second item follows immediately from Prop. 2, the first one is by case analysis on the instructions of the attack. Regarding the second auxiliary result, it's proof proceeds on the basis of three unwinding lemmas. First a definition.

Definition 9. *Let $s_1 = \langle i_1, \alpha_1, \sigma_1 \rangle$ and $s_2 = \langle i_2, \alpha_2, \sigma_2 \rangle$. We define:*

1. $\mathtt{H_I}(\alpha_1) = \mathtt{H_I}(\alpha_2)$ *iff* $\forall x \in \mathbb{X}.\mathcal{V}_{i_1}(x), \mathcal{V}_{i_2}(x) \in \mathtt{H}_I \Rightarrow \alpha_1(x) = \alpha_2(x)$*;*
2. $\mathtt{H_I}(\sigma_1) = \mathtt{H_I}(\sigma_2)$ *iff* $\forall k \in 0..min(|\sigma_1|, |\sigma_2|) - 1.\mathcal{S}_{i_1}(k), \mathcal{S}_{i_2}(k) \in \mathtt{H}_I \Rightarrow \sigma_1(k) = \sigma_2(k)$*;*
3. $\mathtt{H_I}(s_1) = \mathtt{H_I}(s_2)$ *iff* $\mathtt{H_I}(\alpha_1) = \mathtt{H_I}(\alpha_2)$, $\mathtt{H_I}(\sigma_1) = \mathtt{H_I}(\sigma_2)$ *and* $(\mathcal{A}(i_1), \mathcal{A}(i_2) \in \mathtt{H}_I \Rightarrow \mathcal{A}(i_1) = \mathcal{A}(i_2))$*.*

The three unwinding lemmas follow together with the statement of Lem. 8.

Lemma 5. *Suppose (1)* $\mathcal{A}(s_1) \in \mathtt{H}_I$*; (2)* $s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s_1'$*; (3)* $s_2 \longrightarrow_{B[\overrightarrow{a_2}]} s_2'$*; and (4)* $\mathtt{H_I}(s_1) = \mathtt{H_I}(s_2)$*. Then* $\mathtt{H_I}(s_1') = \mathtt{H_I}(s_2')$*.*

Lemma 6. *Let* $\mathsf{pc}(s_1), \mathsf{pc}(s_1'), \mathsf{pc}(s_2) \in \mathtt{L}_I$*. Furthermore, suppose: (1)* $\mathtt{H_I}(s_1) = \mathtt{H_I}(s_1')$*; (2)* $s_1 \longrightarrow_{B[\overrightarrow{a}]} s_2$*; (3) and* $\mathsf{pc}(s_2) \in \mathsf{region}(k)$*. Then* $\mathtt{H_I}(s_2) = \mathtt{H_I}(s_1')$*.*

Lemma 7. *Let* $\mathsf{pc}(s_1), \mathsf{pc}(s_2) \in \mathtt{L}_I$ *and* $\mathsf{pc}(s_1'), \mathsf{pc}(s_2') \in \mathtt{H}_I$*. Furthermore, suppose: (1)* $s_1 \longrightarrow_{B[\overrightarrow{a_1}]} s_1'$ *and* $s_2 \longrightarrow_{B[\overrightarrow{a_2}]} s_2'$*; (2)* $\mathtt{H_I}(s_1) = \mathtt{H_I}(s_2)$*; and (3) and* $\mathsf{pc}(s_1') = \mathsf{pc}(s_2')$*. Then* $\mathtt{H_I}(s_1') = \mathtt{H_I}(s_2')$*.*

Lemma 8 (Preservation of equality of high integrity data). *Let* $M[\overrightarrow{\bullet}] = ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{\bullet}])$ *be a well-typed method and* $\overrightarrow{a_1}$*,* $\overrightarrow{a_2}$ *active attacks. Let* κ *s.t. for all* $\kappa' \in \mathtt{H}_I : \kappa' \preceq \kappa$*,* s_i *in* $\mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1)|_{(\top_C, \kappa)}$*,* s_i' *in* $\mathsf{Tr}_{B[\overrightarrow{a_2}]}(s_1')|_{(\top_C, \kappa)}$*, and* $\mathsf{pc}(s_i), \mathsf{pc}(s_i') \in \mathtt{H}_I$*. If* $\mathtt{H_I}(s_1) = \mathtt{H_I}(s_1')$*, then* $\mathtt{H_I}(s_i) = \mathtt{H_I}(s_i')$*.*

We now deliver the promised result, namely the proof of Prop. 3. Consider any two execution paths of $M[\overrightarrow{a_2}] = ((\boldsymbol{x} : \boldsymbol{\kappa}, \kappa_r), B[\overrightarrow{a_2}])$:

$$\begin{array}{c} s_1 \longrightarrow_{B[\overrightarrow{a_2}]} s_2 \longrightarrow_{B[\overrightarrow{a_2}]} \cdots \longrightarrow_{B[\overrightarrow{a_2}]} s_n \\ s_1' \longrightarrow_{B[\overrightarrow{a_2}]} s_2' \longrightarrow_{B[\overrightarrow{a_2}]} \cdots \longrightarrow_{B[\overrightarrow{a_2}]} s_m' \end{array}$$

where (1) $s_1 = \langle 1, \alpha_1, \sigma_1 \rangle$ and $s_1' = \langle 1, \alpha_1', \sigma_1' \rangle$ with $\sigma_1 = \sigma_1' = \epsilon$, (2) $s_n = \langle v \rangle$ and $s_m' = \langle v' \rangle$, and (3) $s_1 \sim^{(\mathtt{C(A)}, \top_I)} s_1'$ and suppose $\mathcal{A}(1) \in \mathtt{L}_C$. If $\mathtt{H}_I = \emptyset$ then declassification is disallowed by the typing rules and the result follows from Prop. 2. Otherwise we proceed as follows. Starting from s_1 and s_1' repeatedly apply Lem. 1 until it is no longer possible. Let $s_{j'}$ and $s_{j'}'$ be the states that are reached. Note $s_{j'} \sim^{(\mathtt{C(A)}, \top_I)} s_{j'}'$. If $j = n$, then the result holds immediately since the full trace has been attained. Otherwise, $j < n$ and we have two cases to treat (we write $\mathcal{A}(s)$ for $\mathcal{A}(i)$, where $s = \langle i, \alpha, \sigma \rangle$):

1. Case $\mathcal{A}(s_j), \mathcal{A}(s'_j) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$, $s_j = \langle i, \alpha, v \cdot \sigma\rangle$ and $s'_j = \langle i', \alpha', v' \cdot \sigma'\rangle$ and $B[\overrightarrow{a_2}](i) = \mathtt{declassify}\ \kappa$. From $s_j \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} s'_j$ and $\mathcal{A}(s_j), \mathcal{A}(s'_j) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$ we deduce $i = i'$. Also, from the semantics $s_{j+1} = \langle i+1, \alpha, v \cdot \sigma\rangle$ and $s'_{j+1} = \langle i+1, \alpha', v' \cdot \sigma'\rangle$. Finally, by typability of $B[\overrightarrow{\bullet}]$ all hypothesis of T-Declassify are assumed.
   We wish to prove $s_{j+1} \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} s'_{j+1}$. The only non-trivial case is $v \cdot \sigma \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_{i+1},\mathcal{A}(i+1)} v' \cdot \sigma'$. For this we first check that $\sigma \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_{i+1}\backslash 0,\mathcal{A}(i+1)} \sigma'$. This fact is obtained by reasoning as follows:
   - If $\mathcal{A}(i+1) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$, then the result follows from $\sigma \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_i\backslash 0,\mathcal{A}(i)} \sigma'$, $(\mathcal{S}_i\backslash 0) \leq_{\mathcal{A}(i)} (\mathcal{S}_{i+1}\backslash 0)$ the fact that stack indist. is preserved by subtyping.
   - If $\mathcal{A}(i+1) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$, the result follows from the previous item and H-Low.

   Second we must check that $v \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_{i+1}(0)} v'$. By $s_j \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} s'_j$, we know that $v \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_i(0)} v'$. We consider two cases depending on whether $\mathcal{S}_i(0) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$ or $\mathcal{S}_i(0) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$. Suppose $\mathcal{S}_i(0) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$.
   - if $\mathcal{S}_{i+1}(0) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$, $v \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_{i+1}(0)} v'$ follows from $s_j \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} s'_j$ and value indist.;
   - otherwise, we conclude directly by value indist.

   Suppose now $\mathcal{S}_i(0) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$. If $v = v'$ then $v \sim^{(\mathsf{C}(\mathtt{A}),\top_I)}_{\mathcal{S}_{i+1}(0)} v'$ is immediate. But, if $v \neq v'$ and $\mathcal{S}_{i+1}(0) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$, since declassification has occurred, indistinguishability is not a priori guaranteed. Since v, v' are high-integrity data and active attackers cannot modify them (Prop. 4(i)), these same values are at the same program point in the execution of $B[\overrightarrow{a_1}]$ (Lem. 8). Then these same data, v, v', are also declassified by $B[\overrightarrow{a_1}]$ at this program point (since `declassify` instructions do not belong to $B[\overrightarrow{\bullet}]$). By Lem. 8, if high integrity values not are equal at the same program point, then there exist high integrity data on initial states s.t. they are not equal either. If $v \neq v'$, then the given assumption $\mathsf{Tr}_{B[\overrightarrow{a_1}]}(s_1) \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} \mathsf{Tr}_{B[\overrightarrow{a_1}]}(s'_1)$ would fail. Hence it must be the case that $v = v'$.
2. Case $\mathcal{A}(s_j), \mathcal{A}(s'_j) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$. As a consequence, there exists a program point k in $\mathsf{Dom}(B)^\sharp$ such that k was the value of the program counter for s_{j-1} and s'_{j-1}. Furthermore, let $\mathtt{pc}(s_j) = g$ and $\mathtt{pc}(s'_j) = g'$ then we have $k \mapsto g$ and $k \mapsto g'$ by the definition of the Successor Relation. By the SOAP properties (1(i)), both $g, g' \in \mathsf{region}(k)$. Furthermore, by T-IF, $\forall d \in \mathsf{region}(k).\mathcal{S}_k(0) \preceq \mathcal{A}(d)$. In particular, $\mathcal{S}_k(0) \preceq \mathcal{A}(s_j), \mathcal{A}(s'_j)$. Therefore, by the minimality assumption in the definition of well-typed programs, we may assume $\mathcal{S}_k(0) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$. Hence, $\forall d \in \mathsf{region}(k).\mathcal{A}(d) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$. Therefore the hypothesis of Lem. 2 is satisfied.
   Now, we repeatedly apply Lem. 2 in both branches of executions until it is no longer possible. Let us call s_{h_1} and s'_{h_2} the reached states. By transitivity and symmetry of indisinguishability of states, $s_{h_1} \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} s'_{h_2}$. By Lem. 2, $\mathcal{A}(s_{h_1}), \mathcal{A}(s'_{h_2}) \not\preceq (\mathsf{C}(\mathtt{A}), \top_I)$. Also, if $h_1 \mapsto h'_1$ and $h_2 \mapsto h'_2$,

then $\mathcal{A}(h'_1), \mathcal{A}(h'_2) \preceq (\mathsf{C}(\mathtt{A}), \top_I)$. We are therefore at a junction point, $h'_1 = \mathsf{jun}(k) = h'_2$. Finally, by Lem. 3, $s_{h'_1} \sim^{(\mathsf{C}(\mathtt{A}),\top_I)} s'_{h'_2}$.

We repeat the argument, namely applying Lem. 1 until no longer possible and then analyzing cases 1 and case 2. Given that the derivations are finite, eventually case 1 is reached.

6 Related work

A brief description of related literature on type based information flow analysis (IFA) for declassification and for low-level languages follows. Zdancewic and Myers [ZM01] introduce robust declassification (RD) in an attempt to capture the desirable restrictions on declassification (as discussed in this paper), but did not propose a method for determining when a program satisfies RD. Zdancewic [Zda03] shows that a simple change to a security type system can enforce it: extend the lattice of security labels to include integrity constraints as well as confidentiality constraints and then require that the decision to perform declassification have high integrity. Myers et al [MSZ04] introduce a type system for enforcing RD and also, they show how to support upgrading (endorsing) data integrity. Barthe et al [BR05] define a simple JVM bytecode type system for IFA. They prove preservation of noninterference for a compilation function from a core imperative language to bytecode. [BPR07] extends the type system to handle objects and method calls. And present a type-preserving compilation for a Java-like language with objects and exceptions. All these works assume fields have a fixed security level. Less importantly, they do not allow variable reuse and have a less precise management of information flow of the operand stack. [BB08] proposes a core bytecode language together with a flow-sensitive type system that performs IFA in a setting where the security level assigned to fields may vary at different points of the program where their host object is created. Rezk et al [BCR08] provide a modular method for achieving sound type systems for declassification from sound type systems for noninterference, and instantiate this method to a sequential fragment of the JVM. They consider a declassification policy called delimited non-disclosure that combines the *what* and *where* dimensions of declassification.

7 Conclusion

We present a type system for ensuring secure information flow in a core JVM-like language that includes a mechanism for performing downgrading of confidential information. It is proved that the type system enforces robustness of the declassification mechanism in the sense of [ZM01]: attackers may not affect what information is released or whether information is released at all. The restriction of the attackers ability to modify data is handled by enriching the confidentiality lattice with integrity levels.

In presence of flow-sensitive type systems we have showed that variable reuse endows the attacker with further means to affect declassification than those available in high-level languages in which variables are assigned a fixed type during their entire lifetime. Additional means of enhancing the ability of an attacker to declassify information in stack-based languages (e.g. bytecode) is by manipulating the stack, as exemplified in Sec. 2. Also, in unstructured low-level languages, we are required to restrict the use of jumps.

Our current efforts are geared toward enriching the core bytecode language while maintaining our results. Also, the inclusion of a primitive for *endorsement* (for upgrading the integrity of data) as studied elsewhere [MSZ04] in the setting of high-level languages would be interesting. Note that although confidentiality and integrity are dual properties, the treatment of `declassify` and `endorse` should not be symmetric for otherwise the attacker would be given total control over the data.

Acknowledgements. To the referees for helping improve this work considerably.

References

[BB08] Francisco Bavera and Eduardo Bonelli. Type-based information flow analysis for bytecode languages with variable object field policies. In *SAC'08, Proceedings of the 23rd Annual ACM Symposium on Applied Computing. Software Verification Track*, 2008.

[BB09] Francisco Bavera and Eduardo Bonelli. `www.lifia.info.unlp.edu.ar/~eduardo/publications/robustLong.pdf`, 2009.

[BCR08] G. Barthe, S. Cavadini, and T. Rezk. Tractable enforcement of declassification policies. *Computer Security Foundations Symposium, IEEE*, 0:83–97, 2008.

[BPR07] Gilles Barthe, David Pichardie, and Tamara Rezk. A Certified Lightweight Non-Interference Java Bytecode Verifier. In *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science. Springer-Verlag, 2007.

[BR05] Gilles Barthe and Tamara Rezk. Non-interference for a JVM-like language. In *TLDI '05: Proceedings of the 2005 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 103–112, New York, NY, USA, 2005. ACM Press.

[HS06] S. Hunt and D. Sands. On flow-sensitive security types. In *POPL'06, Proceedings of the 33rd Annual. ACM SIGPLAN - SIGACT. Symposium. on Principles of Programming Languages*, January 2006.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification.* Addison Wesley, 1999.

[MSZ04] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *CSFW*, pages 172–186. IEEE Computer Society, 2004.

[SM03] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003.

[Zda03] S. Zdancewic. A type system for robust declassification, 2003.

[ZM01] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *CSFW-14 2001*, pages 5–. IEEE Computer Society, 2001.