

Approximating Non-interference and Erasure in Rewriting Logic

Mauricio Alba-Castro

Universidad Autónoma de Manizales, Manizales, Colombia
ELP-DSIC, Universidad Politécnica de Valencia, Valencia, Spain
malba@autonoma.edu.co

María Alpuente and Santiago Escobar

ELP-DSIC, Universidad Politécnica de Valencia
Valencia, Spain
{alpuente,sescobar}@dsic.upv.es

Abstract—*Non-interference* is a semantic program property that assigns confidentiality levels to data objects and prevents illicit information flows to occur from high to low security levels. *Erasure* is a way of strengthening confidentiality by upgrading data confidentiality levels, up to the extreme of demanding the removal of secret data from the system. In this paper, we propose a certification technique for confidentiality of complete Java classes that includes non-interference and erasure policies. This technique is based on rewriting logic, which is a very general *logical* and *semantic framework* that is efficiently implemented in the high-level programming language Maude. In order to achieve a finite state transition system, we develop an abstract Java semantics which correctly approximates non-interference and erasure. The analysis produces certificates that are independently checkable, and are small enough to be used in practice. We have implemented our methodology and developed some experiments that demonstrate the feasibility of our approach.

I. INTRODUCTION

Confidentiality is a property by which information that is related to an entity is not made disclosed to unauthorized individuals, or processes. A *non-interference policy* [10] is a confidentiality policy that allows programs to manipulate and modify confidential data as long as the observable data generated by those programs do not improperly reveal information about the confidential data, i.e., confidential data does not interfere with publicly observable data. Thus, ensuring that a program adheres to a non-interference policy means analyzing how information flows within the program. The mechanism for transferring information through a computing system is called a *channel*. Variable updating, parameter passing, value return, file reading and writing, and network communication are channels. Channels that use a mechanism that is not designed for information communication are called *covert channels* [19]. There are covert channels such as the control structure of a program, termination, timing, exceptions, and resource exhaustion channels. The information flow that occurs through channels is called *explicit flow* [10] because it does not depend on the specific information that flows. The information flow that occurs through the control structure of a program (conditionals, loops, breaks, and exceptions) is called an *implicit flow* [10] because it depends on the condition that guards the control structure. We are interested in both explicit and

implicit flows for non-interference and erasure analysis of deterministic Java programs. However, in this paper we do not consider channels such as file reading and writing, and network communication, neither covert channels such as exceptions, termination, timing, and resource exhaustion.

Non-interference policies are not restrictive enough for computer systems that are required to remove (or *erase*) the secret data after its intended use [4], [5]. The following example adapted from [5] illustrates erasure policies.

Example 1. Consider an on-line diagnosis web system implemented in Java which, once the patient has entered information about her symptoms, returns information about possible corresponding diseases back to the user. The website policy states that the client symptoms and diagnostics are private, and that no record of them will be stored after the user has finished the application session.

```
class MedicalDiagnosis {
    boolean malaise, fever, influenza, userReqExit;
    public void getSymptoms(){/* */} public void getUserReq(){/* */}
    public void exit(){/* */} public void diagnosis(){/* */}
    public void appEnd(){
        malaise=false; fever=false; influenza=false;}
    public void medicalDiagnosis(){ while (!userReqExit){
        getSymptoms();diagnosis();getUserReq();;appEnd();exit();} }
```

When the user requests to exit the application, the condition `userReqExit` becomes true, the method `appEnd` is executed, and the three sensitive variables `malaise`, `fever`, and `influenza` are erased. That is, the variables `malaise`, `fever`, and `influenza` have a high confidentiality level but no record of their values is kept after execution.

In [2], [1], we proposed an abstract methodology for certifying safety properties of Java source code. The methodology is based on *Rewriting logic* (RWL) and is implemented in Maude [7], which is a high-performance language that implements RWL. Non-interference is usually defined as a hyperproperty [6], i.e., a property defined on a set of sets of traces, and cannot be established by simply checking a (safety) property on a set of runs. However, we are able to analyze non-interference by observing a stronger property which can be checked as a safety¹ property by using an instrumented flow-sensitive semantics.

¹There are other approaches for proving non-interference as a safety property, which use self-composition [9], [3] or flow-sensitive types [13].

The methodology of [2], [1] is as follows. Consider a Java program together with a specification of the Java semantics. The Java program is a concrete expression (i.e., term) that represents the initial state of the Java interpreter running the considered Java program. Given a safety property (i.e., a system property that is defined in terms of certain events that do not happen), the unreachability of the system states denoting the events that should never occur allows us to infer the desired safety property. Unreachability analysis is performed using the standard Maude (breadth-first) search command, which explores the entire state space of the program from an initial system state. In the case where the unreachability test succeeds, the corresponding rewriting proofs that demonstrate that those states cannot be reached are delivered as the expected outcome certificate. Very often, the unreachability test does not succeed because there is an infinite search space; thus, in order to achieve a finite search space we use abstraction [8]. In our methodology, certificates are encoded as (abstract) rewriting sequences that (together with an encoding of the abstraction in Maude) can be checked by standard reduction. Our methodology is an instance of Proof-carrying code (PCC), a mechanism originated by Necula [18] for ensuring the secure behavior of programs.

This article extends the formulation of the abstract global non-interference certification methodology of [2] in order to consider strengthened confidentiality requirements that include erasure policies. The contributions of the paper are as follows:

- We provide an information-flow sensitive semantics of Java programs that deals with erasure. This semantics is formulated as a further extension of the instrumented operational Java semantics of [2].
- We provide an abstract, finite-state version of the information flow operational semantics that supports sound finite program verification. As a by-product of the verification, a certificate is delivered which consists of a set of (abstract) rewriting proofs.
- We have implemented this framework in Maude and made it publicly available on-line².

This paper is organized as follows. In Section II, we recall the notions of erasure with and without non-interference. Then, we describe a mechanism that allows us to specify non-interference and erasure policies in JML. In Section III, we recall the specification of the Java semantics in rewriting logic. In Section IV, we extend this semantics to handle confidential policies that include non-interference and erasure. We formulate two certification methodologies that observe erasure with and without non-interference, both as safety properties. In Section V, we develop an approximation of the extended Java semantics that produces a finite search space for any input Java program. By using this abstract

semantics (which we implement as a source-to-source transformation of the extended semantics in Maude) we formulate our analyses of erasure without and with non-interference, and prove their soundness. We include some experiments in Section VI. A thorough discussion of related work is contained in Section VII. Section VIII concludes.

II. NON-INTERFERENCE AND ERASURE POLICIES

In this section, we first recall the notion of non-interference and introduce both erasure with and without non-interference policies. For simpler policies that just observe non-interference alone, we refer to [2].

A. Non-interference

We assume a fixed Java program P_{Java} . $\text{Vars}(P_{\text{Java}})$ denotes the set of *static* source variables that may be initialized by the *main* function call. We denote the set of public non secret (low-confidentiality) program variables as $\text{Low}(P_{\text{Java}})$. A program state St is a set of value assignments to program variables. Given $var \in \text{Vars}(P_{\text{Java}})$ and a state St , $St[var]$ denotes the value of variable var in St . We model a *Java program* P_{Java} as a state transition system between pairs $\langle P, St \rangle$, where P is the current, still-to-be-executed part of the Java program P_{Java} and St represents the current program state. $\langle P_{\text{Java}}, St_0 \rangle$ denotes the initial *configuration* of standard program execution and $\langle \checkmark, St \rangle$ denotes a final *configuration*, where \checkmark stands for the empty program. Note that we assume that every Java program properly terminates for each set of input data (i.e., we do not consider non-terminating programs, deadlocks, or runtime errors). We also assume deterministic Java programs, without threads or exceptions. \mapsto_{Java} is the transition relation that describes any possible one-step transition between two Java program states. An *execution* (or trace) of P_{Java} is a sequence $\langle P_{\text{Java}}, St_0 \rangle \mapsto_{\text{Java}} \dots \mapsto_{\text{Java}} \langle P_i, St_i \rangle \mapsto_{\text{Java}} \dots \mapsto_{\text{Java}} \langle \checkmark, St_n \rangle$, which is simply denoted by $\langle P_{\text{Java}}, St_0 \rangle \mapsto_{\text{Java}}^* \langle \checkmark, St_n \rangle$ if the intermediate states are irrelevant. We can also abbreviate $\langle \checkmark, St_n \rangle$ by $\langle S_n \rangle$.

Non-interference policies label data objects with their confidentiality levels (let us start with two levels, *High* and *Low*) and allow *only* information flows from *Low* to *High* data objects. We define program non-interference by using an equivalence $=_{\text{Low}}$ relationship between states [19]. Roughly speaking, non-interference establishes that any two terminating runs of a program that start from indistinguishable initial states produce indistinguishable final states.

Definition 1 (State equality [19]). Given a Java program P_{Java} , two states St_1 and St_2 for P_{Java} are *indistinguishable* at the confidentiality level *Low*, written $St_1 =_{\text{Low}} St_2$, if for all $var \in \text{Low}(P_{\text{Java}})$, $St_1[var] = St_2[var]$.

Definition 2 (Non-interference [19]). A Java program P_{Java} is *non-interferent* iff for every pair of different program initial states St_1 and St_2 , and for their corresponding final

²At <http://www.dsic.upv.es/users/elp/toolsMaude/JavaPCC.html>.

program states S'_{t_1}, S'_{t_2} such that $\langle P_{\text{Java}}, S_{t_1} \rangle \mapsto_{\text{Java}}^* \langle S'_{t_1} \rangle$ and $\langle P_{\text{Java}}, S_{t_2} \rangle \mapsto_{\text{Java}}^* \langle S'_{t_2} \rangle$, we have that $S_{t_1} =_{\text{Low}} S_{t_2}$ implies $S'_{t_1} =_{\text{Low}} S'_{t_2}$.

The non-interference condition of Definition 2 is understood as the lack of any *strong dependence* of Low-confidentiality variables on any of the High-confidentiality variables [19]. The attacker model for non-interference assumes that the attacker is passive and can only see the Low-labeled source variables of the Java program at the initial and final states and not at the intermediate states.

A non-interference policy can be represented by a *partially ordered set* $\langle \text{Labels}, \leq \rangle$ and a labeling function $\text{Labeling} : \text{Var} \rightarrow \text{Labels}$, where Labels is the finite set of confidentiality levels, \leq is a partial order between confidentiality levels, and Var is the set of source program variables [10], [13]. For the case of two confidentiality levels, we let $\text{Labels} = \{\text{Low}, \text{High}\}$. These represent public non-secret data (low confidentiality) and secret data (high confidentiality), respectively.

In order to express confidentiality policies, we use the *Java modeling language* JML [16], which is a property specification language for Java modules. The text of a JML annotation can either be in one line after the `//@` marker, or in many lines enclosed between the `/*@` marker and the `@*/` marker. They are ignored by traditional compilers. The initial confidentiality level of a variable in a Java program is written with the word `setLabel` as a JML annotation (e.g. `setLabel(var, High)`). The confidentiality label of program variables is Low if nothing is specified. We do not need to specify the label of the formal parameters, nor of the local variables because they can be inferred from the confidentiality labels of other program variables if they are properly initialized. These JML annotations, together with the default assumption, define the labeling function of the non-interference policy.

Example 2. Consider the following Java program borrowed from [9] that models a bank account and the initial state given by the execution of the function `main`:

```
public class Account { public boolean extraService;
  int balance; //@ setLabel(balance, High);
  public Account() { balance = 0; extraService = false; }
  public void writeBalance(int amount) { balance = amount;
    if (balance >= 100000) extraService = true;
    else extraService = false; }
  private int readBalance() { return balance; }
  public boolean readExtra() { return extraService; } }
class System { static Account a = new Account();
  static int initbalance; //@ setLabel(initbalance, High);
  public static void main(String[] args) {
    initbalance = Integer.parseInt(args[0]);
    a.writeBalance(initbalance); System.out.println(readExtra()); }
```

This non-interference policy specifies that the object field `balance` of the global object `a` and the initialization parameter `initbalance` (i.e. `args[0]`) hold secret data. This program is insecure w.r.t. this policy since an observer with low access rights can obtain partial information about

the variable `balance` via an observation of the non-secret variable `extraService`.

B. Erasure

An erasure policy is a confidentiality policy that specifies that the confidentiality level of a given variable is upgraded to the extent that the system should not keep its value [4], [5], [14]. Erasure does not imply non-interference (a program that satisfies erasure may not satisfy non-interference), nor vice versa. The erasure of a variable var is expressed as $var : L_1 \nearrow L_2$ where $L_1 \in \{\text{Low}, \text{High}\}$, $L_2 \in \{\text{High}, \top\}$, \top is the top element, and $L_1 < L_2$. This means that the (value of the) variable var should be explicitly erased within the program code in every execution, as well as any other variable that *depends on* var . The partial order \leq is extended so that $\text{Low} < \text{High} < \top$. The commutative \sqcup operator is also extended such that $\text{Low} \sqcup \text{Low} = \text{Low}$, $\text{Low} \sqcup \text{High} = \text{High}$, and $X \sqcup \top = \top$.

If $var : L_1 \nearrow \top$, the erasure policy means complete erasure [14], i.e. initial states that differ only in the value of variable var produce the same final state. This means that even if an observer can see High-labeled variables it should not distinguish the erased values. Erasure policies with complete erasure mean that attackers can see Low-labeled variables (as the Low clearance of the non-interference attacker) and also High-labeled variables. If $var : L_1 \nearrow \text{High}$, this means partial erasure, i.e. initial states that differ only in the value of variable var produce the same final state, except for High variables. In summary, given an erasure policy $var : L_1 \nearrow L_2$, the erasure attacker has a clearance level as high as the highest level L such that $L < L_2$.

The erasure policy will be enforced from the program point where the JML annotation is located until the end of the entire program. This means that we should place the erase annotation at the program point just *after* the variable is updated with the confidential data that must be erased. This can be seen as an instrumentation in our setting of the (conditioned) erasure of [4], [5]. In our JML-like notation for erasure policy specification, an erasure policy $var : L_1 \nearrow L_2$ is written as a JML annotation with the word `erase`, e.g. `erase(influenza, High, Top)` represents $\text{influenza} : \text{High} \nearrow \top$.

Example 3. Consider the following example adapted from [14] and the initial state given by the execution of the function `main`:

```
class Testclass { int xh, yh, zl;
  //@ setLabel(xh, High); setLabel(yh, High);
  public void Testclass() { }
  public void setxh(int xp) { /*@ setLabel(xp, High); @*/ xh = xp; }
  public void setyh(int yp) { /*@ setLabel(yp, High); @*/ yh = yp; }
  public void setzl(int zp) { zl = zp; /*@ erase(zl, Low, High); @*/ }
  public void mE3() { xh = xh + yh + zl; yh = yh + 2; zl = 0; } }
class Erasure3 { static int initzl;
  static Testclass t = new Testclass();
  public static void main(String[] args) {
    initzl = Integer.parseInt(args[0]); t.setzl(initzl); t.mE3(); }
```

In the `TestClass`, the fields `xh` and `yh` are labeled `High`, and the field `z1` is labeled `Low` by default. The program `Erase3` obeys the erasure policy $z1 : \text{Low} \nearrow \text{High}$ because it erases the value of the `z1` variable that is set by the method `setz1` before program ends execution. In fact, it erases this value in the last assignment statement of the method `mE3`. The other variable `yh` that is labeled `High`, does not depend on `z1` whereas the variable `xh` depends on `z1` but it remains labeled as `High`. It must be noted that if we modify the erasure policy of `z1` from $z1 : \text{Low} \nearrow \text{High}$ to $z1 : \text{Low} \nearrow \top$, the program is no longer secure, since variable `xh` should also be erased. Finally, if we replace the statement “`z1 = 0;`” by “`z1 = yh;`”, the program satisfies the erasure policy but not the non-interference policy.

We define erasure of variables following the end-to-end erasure of non-interactive programs [14], but regarding the erasure of n variables. A variable var_i is erased to some confidentiality level L_i if varying the initial value of var_i does not change the final state to all observers except those at level L_i or above. Given a subset of program variables $V \subseteq \text{Vars}(P_{\text{Java}})$, we define an equivalence relationship $=_{\bar{V}}$ between states as $S_{t_1} =_{\bar{V}} S_{t_2}$ if for all $var \in \text{Vars}(P_{\text{Java}}) - V$, it holds that $S_{t_1}[var] = S_{t_2}[var]$. We extend the notation $S_{t_1} =_{\text{Low}} S_{t_2}$ of Definition 1 as follows: $S_{t_1} =_L S_{t_2}$ with $L \in \text{Labels} = \{\text{Low}, \text{High}, \top\}$, if for all $var \in \text{Vars}(P_{\text{Java}})$ such that $\text{Labeling}(var) \leq L$ it holds that $S_{t_1}[var] = S_{t_2}[var]$. Note that $S_{t_1} =_{\top} S_{t_2}$ implies $S_{t_1} =_{\text{High}} S_{t_2}$, which implies $S_{t_1} =_{\text{Low}} S_{t_2}$.

Definition 3 (Erasure [14]). A Java program P_{Java} complies with the erasure policy $var_i : L_i \nearrow L'_i$, with $i = 1 \dots n$, iff for every pair of different program initial states S_{t_1} and S_{t_2} , and for their corresponding final program states³ S'_{t_1} , S'_{t_2} , we have that $S_{t_1} =_{\bar{V}} S_{t_2}$ implies $S'_{t_1} =_L S'_{t_2}$ for all $L < [L]$, where $V = \{var_1, \dots, var_n\}$ and $[L] = L'_1 \sqcup \dots \sqcup L'_n$.

This means that if each variable var_i is erased to confidentiality level L'_i , then varying the values of var_i does not change the final state to all observers except those at level $[L]$ or above, where $[L]$ is the lowest upper bound (*join*) of the L'_i confidentiality levels. Note that the attacker clearance depends on that *lub* level $[L]$. The following example considers the erasure of two variables to different confidentiality levels.

Example 4. Consider the following example with two `High`-labeled variables `xh` and `yh`, and two `Low`-labeled variables `u1` and `z1`, together with a `main` function and a constructor method similar to Example 3, but with a different erasure policy.

```
class Testclass { int xh; int yh; int z1; int u1;
  // setLabel(xh, High); setLabel(yh, High);
  public void setxh(int xp){/*@ setLabel(xp, High); */ @*/ xh = xp; }
  public void setyh(int yp){/*@ setLabel(yp, High); */ @*/ yh = yp; }
```

³i.e. such that $\langle P_{\text{Java}}, S_{t_1} \rangle \mapsto_{\text{Java}}^* \langle S'_{t_1} \rangle$ and $\langle P_{\text{Java}}, S_{t_2} \rangle \mapsto_{\text{Java}}^* \langle S'_{t_2} \rangle$

```
public void setu1(int up){ u1 = up; /*@ erase(u1, Low, Top); */ }
public void setz1(int zp){ z1 = zp; /*@ erase(z1, Low, High); */ }
public void mE4(){ xh = xh + z1; yh = yh + u1; z1 = 0; u1 = 0; }
class Erasure4 { static Testclass t = new Testclass();
  static int initu1, initz1;
  public static void main(String[] args) {
    initu1 = Integer.parseInt(args[0]); t.setu1(initu1);
    initz1 = Integer.parseInt(args[1]); t.setz1(initz1); t.mE4(); }
```

This program does comply with the single erasure policy $z1 : \text{Low} \nearrow \text{High}$ because it satisfies the condition $S'_{t_1} =_{\text{Low}} S'_{t_2}$. However, it does not comply with the single erasure policy $u1 : \text{Low} \nearrow \top$ even if it satisfies the condition $S'_{t_1} =_{\text{Low}} S'_{t_2}$, because it does not satisfy the condition $S'_{t_1} =_{\text{High}} S'_{t_2}$ (i.e. the variable `u1` is erased, but the variable `yh` is not). When we consider the erasure of two variables $z1 : \text{Low} \nearrow \text{High}$ and $u1 : \text{Low} \nearrow \top$ altogether, it is enough to require the conditions corresponding to the later erasure, e.g. $S'_{t_1} =_{\text{High}} S'_{t_2}$ and $S'_{t_1} =_{\text{Low}} S'_{t_2}$, because this implies the condition required by the former erasure (i.e. $S'_{t_1} =_{\text{Low}} S'_{t_2}$). Thus, the erasure policy of the two variables is fulfilled, whenever $S'_{t_1} =_L S'_{t_2}$, for all $L < \text{High} \sqcup \top$.

C. Erasure and Non-interference

In practice, it is useful to enforce erasure together with non-interference [14], [5]. We state that a program P_{Java} complies with a combined erasure with non-interference policy if it simply satisfies the conditions of both, Definition 2 and Definition 3, independently. If we wish to enforce an erasure policy together with a non-interference policy, regarding both the variables directly affected by the erasure policy and the variables that depend on variables directly affected by an erasure policy, the erasure of a given variable var should be done by using an expression whose value has a confidentiality label as high as $\text{Labeling}(var)$.

III. THE REWRITING LOGIC SEMANTICS OF JAVA

In the following, we briefly recall the rewriting logic semantics of Java that was originally given in [11]. We refer the reader to [17] for further technical details on rewriting logic semantics.

In [11], a sufficiently large subset of full Java 1.4 language is specified in Maude. However, Java native methods and many of the available Java built-in libraries are not supported. The specification of the Java operational semantics is a rewrite theory: a triple $\mathcal{R}_{\text{Java}} = (\Sigma_{\text{Java}}, E_{\text{Java}}, R_{\text{Java}})$ where Σ_{Java} is an order-sorted *signature*; $E_{\text{Java}} = \Delta_{\text{Java}} \uplus B_{\text{Java}}$ is a set of Σ_{Java} -equational *axioms* where Δ_{Java} is a set of terminating and confluent (modulo B_{Java}) equations and B_{Java} are algebraic axioms such as associativity, commutativity and unity. Finally, R_{Java} is a set of Σ_{Java} -rewrite rules that are not required to be confluent nor terminating. Intuitively, the sorts and function symbols in Σ_{Java} describe the static structure of the Java program state space as an algebraic data type; the equations in Δ_{Java} describe the operational semantics of its deterministic features; and the rules in R_{Java} describe its concurrent features. Following the rewriting logic framework

we denote by $u \rightarrow_{\text{Java}}^r v$ the fact that the concrete terms u, v (which denote Java program states) are rewritten (at the top position, see [11]) by using r , which is either a rule in R_{Java} or an equation in Δ_{Java} (both of which are applied modulo B_{Java}). We simply write $u \rightarrow_{\text{Java}} v$ when the applied rule or equation is irrelevant. We denote by $\rightarrow_{\text{Java}}^*$ the extension of $\rightarrow_{\text{Java}}$ to multiple rewrite steps (i.e., $u \rightarrow_{\text{Java}}^* v$ if there exist u_1, \dots, u_k such that $u \rightarrow_{\text{Java}} u_1 \rightarrow_{\text{Java}} u_2 \cdots u_k \rightarrow_{\text{Java}} v$).

Intuitively, equations in Δ_{Java} and rules in R_{Java} are used to specify the changes to the program state (i.e., the changes to the memory, input/output, etc). Since we consider only deterministic Java programs, our specification of the Java semantics in rewriting logic contains only equations and no rules. For a RWL specification of the semantics of a programming language with threads we refer to [17], [1].

The semantics of Java is defined in a *continuation-based style* [17] and specified in Maude itself. Continuations maintain the control context, which explicitly specifies the next steps to be performed. The sequence of actions that still need to be executed are stacked. We use letters K, K' to denote continuation variables, letters E, E' to denote expressions to be evaluated, and Val, Val' to denote values (i.e., the result of evaluating an expression). Once the expression e on the top of a continuation ($e \rightarrow k$) is evaluated, its result will be passed on to the remaining continuation k . The if-then-else

$$\begin{aligned} \text{eq } k((\text{if } E \text{ S else } S') \rightarrow K) &= k(E \rightarrow (\text{if}(S, S') \rightarrow K)) . \\ \text{eq } k(\text{bool}(\text{true}) \rightarrow (\text{if}(S, S') \rightarrow K)) &= k(S \rightarrow K) . \\ \text{eq } k(\text{bool}(\text{false}) \rightarrow (\text{if}(S, S') \rightarrow K)) &= k(S' \rightarrow K) . \end{aligned}$$

Figure 1. Continuation-based equations for if-then-else statement

statement is shown in Figure 1. While statements (loops), break statements, method calls and heap manipulation are not shown in this paper due to space limitations.

IV. PROVING ERASURE AND NON-INTERFERENCE WITH ERASURE BY USING AN EXTENDED INSTRUMENTED SEMANTICS

In previous work [2], we proved (a strong notion of) non-interference as a safety property by instrumenting the Java semantics in order to dynamically keep track of the change of the confidentiality labels of program variables. The semantic instrumentation for non-interference is defined in [1], [2] by attaching confidentiality level labels to memory locations, programs expressions and program statements.

In order to consider both erasure and non-interference, the semantic instrumentation is extended altogether in this paper as follows:

- 1) Attach two extra labels to each memory location: a confidentiality label from the set $\{\text{Low}, \text{High}, \text{Low} \gg \text{High}, \text{Low} \gg \top, \text{High} \gg \text{Low}, \text{High} \gg \top\}$ and an erasure label from the set $\{\text{Low} \nearrow \text{High}, \text{Low} \nearrow \top, \text{High} \nearrow \top\}$. When program ends execution, a label $L \gg L'$ with $L \neq L'$ indicates a change of the confidentiality level label of the associated memory location, from the initial level L to the final level L' . This way we can detect hazardous confidentiality level changes of these

locations, e.g. from level *Low* to level *High*. In other words, we can not only observe the confidentiality levels of the memory locations at the final execution state but also detect whether a location is directly affected by an erasure policy (or if it depends on variables that must obey an erasure policy).

- 2) Attach a confidentiality label and an erasure label to the evaluation of program expressions; this allows us to track whether the evaluation of the expression involves high confidentiality data and data that are affected by an erasure policy.
- 3) Associate a confidentiality label and an erasure label to the evaluation of program statements, especially conditional statements, to control implicit flows from high confidential variables and from variables that should be erased. However, these confidentiality and erasure labels are not attached to each program statement. Rather they are kept as extra attributes of states in the extended Java semantics.

A. Extended Semantics for Non-interference and Erasure

Here we extend further, in a modular way, the extended Java semantics for non-interference of [2] to deal with the JML annotations of the form $\text{erase}(\text{Var}, L_1, L_2)$, which correspond to the erasure policy. We describe the information-flow extended version of the rewriting logic semantics of Java by the rewrite theory $\mathcal{R}_{\text{Java}^E} = (\Sigma_{\text{Java}^E}, E_{\text{Java}^E}, R_{\text{Java}^E})$, $E_{\text{Java}^E} = \Delta_{\text{Java}^E} \uplus B_{\text{Java}^E}$ and its corresponding $\rightarrow_{\text{Java}^E}$ rewriting relation. In the new semantics, program data do not only consist of standard concrete values but each value is decorated with its corresponding confidentiality and erasure labels. We introduce the new confidentiality label \top . Thus, we now have $\text{Labels} = \{\text{Low}, \text{High}, \top\}$ and $\text{ConfLabChange} = \{\text{Low} \gg \text{High}, \text{Low} \gg \top, \text{High} \gg \text{Low}, \text{High} \gg \top\}$. We extend the join operator consistently in order to consider the new \top label, as well as the new composed labels $\{\text{Low} \gg \top, \text{High} \gg \top\}$. Also, $L \text{ join } \top = \top$ for any $L \in \text{Labels}$. Regarding variable updating, we also extend the \gg operator (that computes the new confidentiality label in terms of the previous label at the memory location) as shown in Figure 2.

Previously Stored Label	\gg	New Label	=	New Stored Label
L_1	\gg	L_1	=	L_1
L_1	\gg	L_2	=	$L_1 \gg L_2$
$L_1 \gg L_2$	\gg	L_1	=	L_1
$L_1 \gg L_2$	\gg	L_3	=	$L_1 \gg L_3$
$L_1 \in \{\text{Low}, \text{High}\}, L_2, L_3 \in \{\text{Low}, \text{High}, \top\}$				

Figure 2. Updating memory locations for Erasure

In order to record erasure, we introduce the sort $\text{EraLabels} = \{\emptyset, \text{Low} \nearrow \text{High}, \text{Low} \nearrow \top, \text{High} \nearrow \top\}$. The domain of program variables in the extended semantics for erasure and non-interference is now $\text{Value} \times (\text{Labels} \cup \text{ConfLabChange}) \times \text{EraLabels}$. The empty label (\emptyset) of domain EraLabels means no erasure policy. If a variable var has the labeled value $\langle \text{val}, L, L' \nearrow L'' \rangle$, L is the confidentiality level of the variable, with $L \in \text{Labels} \cup \text{ConfLabChange}$, and

$L' \nearrow L''$ is the erasure label of the variable. It holds that either $L = L''$ or $L = L''' \gg L''$. The erasure label $L' \nearrow L''$ means that the variable var has to be erased, either because this erasure label corresponds to an erasure policy for var (i.e. $var : L' \nearrow L''$) or because, the variable var is (recursively) affected by an erasure policy for other variables on which var depends on. With these labels, we can check whether the variables are involved in erasure or non-interference policies. If they are involved in an erasure policy, we also can check if they must be erased or not when program ends execution, using their confidentiality labels.

We also extend the labels of expressions and statements in order to propagate the erasure label of a variable to the variables that depend on it, explicitly or implicitly. The context label has now two labels, the confidentiality label and the erasure label. We introduce a new commutative \boxplus (join) operator for erasure labels in order to propagate the strongest policies on erased values and its dependences, as shown in Figure 3 for an op binary operator. The \boxplus operator is used during constant and variable evaluation to join the erasure label of the constant (or variable) with the erasure label of the program context. The \boxplus operator is also used during the evaluation of expressions with operators to join the erasure labels of the operands. When a boolean expression guards a conditional statement, the \boxplus operator is also used to update the erasure label of the program context in order to consider implicit flows, as shown in Figure 5.

$$\frac{\text{Labeled Value } op \text{ Labeled Value} = \text{Resulting Labeled Value}}{\langle v_1, l_1, p_1 \rangle \quad op \quad \langle v_2, l_2, p_2 \rangle = \langle v_1 \quad op \quad v_2, l_1 \sqcup l_2, p_1 \boxplus p_2 \rangle}$$

$v_1, v_2 \in \text{Value}, l_1, l_2 \in \text{Labels} \cup \text{ConfLabChange}$

Figure 3. Expression evaluation

$$\frac{\text{Erasure Label } \boxplus \text{ Erasure Label} = \text{Resulting Erasure Label}}{l_1 \nearrow l_2 \quad \boxplus \quad \emptyset = l_1 \nearrow l_2}$$

$$l_1 \nearrow l_2 \quad \boxplus \quad l_3 \nearrow l_4 = (l_1 \sqcup l_3) \nearrow (l_2 \sqcup l_4)$$

Figure 4. Joining over erasure labels

```

--- Evaluates expression keeping the then and else stmts
ceq k((if E S else S') -> K) lenv(CL, EL) = k(E ->
  (if(S, S') -> restoreLEnv(CL, EL) -> K)) lenv(CL, EL)
if not break-or-continue(S) and not break-or-continue(S') .
ceq k((if E S else S') -> K) = k(E -> (if(S, S') -> K))
  if break-or-continue(S) or break-or-continue(S') .
eq k(<bool(true), LVal, EVal> -> (if(S, S') -> K)) lenv(CL, EL)
  = k(S -> K) lenv(CL join LVal, EL join EVal) .
eq k(<bool(false), LVal, EVal> -> (if(S, S') -> K)) lenv(CL, EL)
  = k(S' -> K) lenv(CL join LVal, EL join EVal) .
--- New equation to restore previous context labels
eq k(restoreLEnv(CL, EL) -> K) lenv(CL', EL') = k(K) lenv(CL, EL) .

```

Figure 5. Extended equations for the if-then-else with erasure labels

Finally, we automatically transform every JML annotation of the form $\text{erase}(\text{Var}, L_1, L_2)$ into calls to the special Java operators eraseT and eraseH as shown below, where $X \in \{\text{Low}, \text{High}\}$.

JML erasure annotations:	Java generated code:
// @ erase(Var, X, Top);	eraseT(Var);
// @ erase(Var, Low, High);	eraseH(Var);

These eraseT and eraseH Java operators have no semantics in Java (i.e., they behave as the identity function), but

our technique interprets them in the proper way as shown in Figure 6. The Java eraseH operator upgrades the Low

```

--- get and keep the variable location and obtain its value
eq k((eraseT < Var'> -> K) obj(o(OA)) env([Var', L'] E'))
  = k(#(L') ->(eraseT(Var', L') ->K)) obj(o(OA)) env([Var', L'] E') .
--- Then upgrade label and store at var location
eq k(Val->(eraseT(Var, Loc)-> K)) = k([eraseT(Val)-> Loc] -> K) .
op eraseT : Value -> Value .
eq eraseT(< Val, L1, EL >) = < Val , Top, L1 -> Top > .
eq eraseT(< Val, L1 >> L2, EL>) = < Val , Top, L1 -> Top > .

```

Figure 6. Java and Maude eraseT operator equations

confidentiality label of the variable var up to High. The definitions and equations of the Java and Maude eraseH operator are similar to the eraseT case described above and are thus omitted.

B. Proving Erasure and Non-interference with Erasure as a Safety property

Let us introduce a notion of erasure that is stated as a safety property.

Definition 4 (Strong Erasure). For a given labeling function, a Java program P_{Java} strongly complies with the erasure policy $var_i: L_i \nearrow L'_i$, for $i = 1 \dots n$, if for every extended initial state St_1^E and for its corresponding final program state St_2^E given by $\langle P_{\text{Java}}, St_1^E \rangle \mapsto_{\text{JavaE}}^* \langle St_2^E \rangle$, we have that for all $var \in \text{Vars}(P_{\text{Java}})$, either $St_2^E[var] = \langle \text{Val}, \text{Lab}, \emptyset \rangle$ or $St_2^E[var] = \langle \text{Val}, \text{High}, \text{Low} \nearrow \text{High} \rangle$, for a value Val and a label Lab .

If a public variable has a non-empty erasure label, this means that the variable must be erased. However, if a secret variable has a non-empty erasure label, this does not necessarily mean that the variable must be erased. A secret variable with the erasure label “Low \nearrow High” must not be erased, because it has the confidentiality label “High” that corresponds to its High-confidentiality. For a program that does not comply with a strong erasure policy, this means that in the final state of, at least, one extended execution there is one variable that, either, (i) it is a public variable that has a non-empty erasure label, or (ii) it is a secret variable that has a non-empty erasure label that indicates complete erasure (i.e. $L \gg \top$ for $L \in \{\text{Low}, \text{High}\}$).

We conclude that strong erasure implies erasure as given by the following result.

Theorem 1 (Strong Erasure Soundness). Given a Java program P_{Java} with the erasure policy $var_i: L_i \nearrow L'_i$, if P_{Java} strongly complies with this erasure policy (Definition 4), then P_{Java} complies with this erasure policy (Definition 3).

In other words, we transform erasure into a stronger, safety property in the extended semantics. Obviously, we are not able to certify the security of all of the programs that are secure with erasure. Note that this is not a limitation of our method but a simple consequence of the undecidability of erasure, and affects any erasure analysis technique including those based on type inference [14].

If we want to analyse non–interference with erasure we have to use our stronger notion of non–interference as a safety property [2], that we recall as follows.

Definition 5 (Strong Non-Interference). A Java program P_{Java} is *strongly non–interferent* for a given labeling function if for every extended initial state St_1^E and for its corresponding final program state St_2^E given by $\langle P_{\text{Java}}, St_1^E \rangle \mapsto_{\text{Java}^E}^* \langle St_2^E \rangle$, we have that for all $var \in \text{Low}(P_{\text{Java}})$, $St_2^E[var] = \langle \text{Val}, \text{Low} \rangle$ for a value Val .

In [2] we prove the soundness of our approach to non–interference stated in the theorem that follows:

Theorem 2 (Strong Non-Interference Soundness). Given a Java program P_{Java} , if P_{Java} is strongly non–interferent (Definition 5), then P_{Java} is non–interferent (Definition 2).

In our methodology, a program P_{Java} complies with both, non–interference and erasure policies if it complies with the two policies separately. The following result is immediate.

Corollary 1 (Strong Erasure with Non–interference Soundness). A program P_{Java} is non–interferent (Definition 2) and secure with end-to-end erasure (Definition 3), if it satisfies conditions of Definitions 4 and 5.

V. APPROXIMATING ERASURE AND ERASURE WITH NON–INTERFERENCE BY USING AN ABSTRACT SEMANTICS

We develop an abstract version of the extended rewriting logic semantics of Java developed in Section IV, which we describe by the rewrite theory $\mathcal{R}_{\text{Java}^\#} = (\Sigma_{\text{Java}^\#}, E_{\text{Java}^\#}, R_{\text{Java}^\#})$, $E_{\text{Java}^\#} = \Delta_{\text{Java}^\#} \uplus B_{\text{Java}^\#}$ and its corresponding $\rightarrow_{\text{Java}^\#}$ rewriting relation. As in Section IV, our approach for the abstract Java semantics consists of modifying the original theory $\mathcal{R}_{\text{Java}^E}$ (taking advantage of its modularity) by abstracting the domain to $(\text{Labels} \cup \text{ConfLabChange}) \times \text{EraLabels}$, and introducing approximate versions of the Java constructions and operators tailored to this domain. Intuitively, this means that we simply get rid of the values in the abstract semantics, and use only their confidentiality and erasure labels as the abstract values instead.

```
r1 k( (LVal, EraVal) -> (if(S,S') -> K) ) lenv(CL,CEraL)
=> k(S -> K) lenv(CL join LVal, CEraL join EraVal) .
r1 k( (LVal, EraVal) -> (if(S,S') -> K) ) lenv(CL,CEraL)
=> k(S' -> K) lenv(CL join LVal, CEraL join EraVal) .
```

Figure 7. Abstract rules for the if-then-else

The abstract semantics is mainly a straightforward extension of the extended semantics. The only difference is that any set of equations that become non confluent (because they have the same left hand side in the abstract semantics) is transformed into rules. As a representative example, the abstract rules associated to two of the equations of the extended semantics of the if-then-else statement are shown in Figure 7. The formalization and soundness of our abstract interpretation is similar to [2], and hence omitted.

Example 5. Consider Example 3 together with the constructor method:

```
public Testclass(int xp,int yp,int zp) { @@ setLabel(xp,High);
/*@ setLabel(xp,High); @*/ xh =xp; yh =yp; zl =zp; }
and the main class:
class Erasure3 { static Testclass t = new Testclass(3, -3, 0);
public static void main(String[] args){ t.setzl( 6); t.mE3();}}
```

In the search command below, we ask for all possible program abstract final states.

```
search in PGM-SEMANTICS-ABSTR :
java((preprocess(EX5-MAUDE) noType . 'main
<new string [i(0)]> noVal)) =>! JS:JavaState .
Solution 1 (state 0)
store(...[1(2),o(f([t(t('Testclass)),f(['xh,l(6)] ['yh,l(7)]
['zl,l(8)]))])..),0)]..[1(6),High,0] [1(7),High,0] [1(8),Low,0])..)
No more solutions.
```

After the execution of the program “Erasure3.java”, the search command returns that only one extended final state of the Java program is possible, which has the values High, High, and Low, in the memory locations 1(6), 1(7), and 1(8) which correspond to the variables xh, yh, and zl, respectively. These final variable values show that the erasure policy is fulfilled.

VI. EXPERIMENTS

The certification methodology presented here has been implemented in Maude. The prototype system offers a rewriting-based program certification service, which is able to analyze confidentiality global program properties related to non–interference and erasure.

The Java operational semantics in rewriting logic that we have used is modular and has 2635 lines of code in 4 files [11]. We have modified less than 25 of the 1527 lines of code in the main file of the original Java semantics. The abstract operational Java semantics was developed as a source-to-source transformation in rewriting logic and consists of 419 lines of extra code with 123 equations and 8 rules. This is equivalent to saying that, in our current system, the *trusted computing base* (TCB)⁴ is less than a sixth of the size of the original Java semantics (at least one order of magnitude smaller than the standard rewriting infrastructure, and even much smaller than other PCC systems).

Programs	Source Size LOC	Source Cicl. Comp.	Full Cert. Size (Kb)	Red. Cert. Size (Kb)	Size Rel. (Red. / Source)	Full Cert. Gen. Time (ms)	Red. Cert. Gen. Time (ms)
1	90	3	2991.911	14.371	0.006	1765	88
2	34	2	1377.158	6.253	0.04	1156	313
3	51	1	1017.592	6,253	0.004	407	22
4	46	2	1127.606	10.373	0.007	581	53
5	88	3	2820.719	18.86	0.008	1680	97
6	47	1	1020.961	6,253	0.004	437	35
7	117	192	20009.28	389.62	0.111	30409	1778

Table I
CERTIFICATE SIZES AND CERTIFICATION TIMES

We have benchmarked our implementation including the examples used in the paper. In Table I, we study three key

⁴The TCB is the part of the code that is used to check if other code can be safely run, and that have to be assumed that it is trusted.

points for the practicality of our approach: the size of the reduced certificate versus the Java source code, the size of the reduced certificate versus the size of the full certificate and the relative efficiency of producing certificates. The experiments have been performed on a MacBook with 2 Gb RAM.

The columns “Source Size LOC”, and “Source Cycl. Comp.”, show two source metrics, namely the lines of source code and the cyclomatic complexity (number of paths), respectively. The two columns for “Full Cert.” show the size in Kbytes (similarly for the two columns of “Red. Cert.”) and the generation time, respectively, for the full certificates. Running times are given in milliseconds and were averaged over a sufficient number of iterations.

Programs 1, 2 and 3 are respectively the ones of Examples 1, 2 and 3 previously introduced in the paper. Example 4 is adapted from [14] and does not comply with the required erasure policy. Example 5 is a version of Example 1 that does not erase the required variables either. Example 6 is adapted from [14] and it does comply with its erasure policy. Example 7, borrowed from [2], includes three methods that invoke nine simple example methods (seven interferent methods and two non-interferent ones) taken from [20], [21]. Since Examples 4 and 5 are counterexamples, the figures correspond to the generation time and size of the abstract traces. The certificates of Examples 1 and 6 correspond to the full annotated versions.

Note the correlation between both the source size and the cyclomatic complexity, with the size and generation time of the certificates. The experiments are very encouraging since the reduction in size of the certificate is very significant (at least two orders of magnitude in all cases), the quotient “Red. Cert. Size/Full Cert. Size” ranging from 1.94% in Example 7 to 0.45% for Example 2. When the time employed to generate the full and reduced certificates is compared, the reduced certificate generation time takes only 5.8% of the full certificate generation time for the biggest source size of the full annotated version of Example 7.

VII. RELATED WORK

In the following, we discuss the existing literature regarding erasure, with or without non-interference. For a discussion of the much wider literature related to non-interference (without erasure), we refer to [2], [1] and references therein. The first work that addressed information erasure from an information flow perspective was [4], where erasure is combined with downgrading (declassification) policies. Erasure policies are defined in [4], [5] with conditions (i.e. $var: L_1 \nearrow L_2$), and are enforced only when condition c is satisfied during program execution. In [5], the Jif programming language (a source Java extension) is extended to handle erasure and declassification. In this paper, we provide a less precise but simpler and convenient alternative to explicit

conditions. Both papers [4], [5] analyse erasure with non-interference whereas we can analyse erasure with or without non-interference. Another advantage of our proposal is that it can be applied to real Java programs by just inserting the code annotations that express the required policies. Moreover, we do not need runtime policy enforcement since our hybrid policy verification methodology is based on static as well as dynamic mechanisms. Finally, we provide an effective implementation that supports program certification.

A static analysis approach to enforce erasure was proposed in [14], which extends the flow sensitive type system for non-interference of [13] to enforce erasure with non-interference. It considers erasure with and without non-interference for terminating deterministic programs with input and output channels. The work introduces a block structured input command “input $x : a \nearrow b$ in C ”, where x is a variable, a is an input channel with confidentiality level a , b is a higher confidentiality level, and C is a command. This local end-to-end erasure condition means that command C should erase the input x obtained from channel a to the level b . This proposal does not enforce erasure without non-interference, nor includes procedure neither function invocations. Also, it does not consider high guarded loops, and for the best of our knowledge, it has not been implemented yet. In [12], the notion of non-interference was extended to consider a simple erasure policy to be applied to the Carmel Core language, an abstract version of the Java Card bytecode (a subset of the Java bytecode). The simple erasure policy $Low \nearrow High$ indicates that the erasure of a whole level, instead of some program variables, should be done before the program ends its execution. The paper analyzes erasure as an extension of non-interference but the erasure of $High$ -labeled variables is not considered. Moreover, verification or enforcement of erasure policies is not considered either. The work [15] studies erasure in multi-threaded programs written in the prototypical MWL language. The considered erasure policies state the erasure of program variables as an upgrading of low-labeled variables into high-labeled ones. Erasure is observed as an extension of non-interference, namely as a reclassification of some low-labeled variables. However, the erasure of $High$ -labeled variables is not considered, and as far as we know, this proposal is not yet implemented.

VIII. CONCLUSION

In this paper, we formalize a framework for automatically certifying erasure with and without non-interference of Java programs. Our methodology relies on an (abstract) extended semantics for Java written in rewriting logic that can be model-checked by using Maude’s breadth-first search space exploration. In the extended semantics, erasure with and without non-interference are observed as safety properties, and we formally demonstrate the correctness of the analysis. The proposed framework fully accounts for explicit as well

as implicit flows, and allows not only the inference of rewriting logic safety proofs but also the checking of existing ones, thus providing support for proof-carrying code. Actually, the steps that the abstract semantics takes are recorded in order to construct a certificate ensuring that the program satisfies the desired property. By turning a potentially infinite labelled state space of a Java program into a finite abstract space, the abstract semantics not only makes the approach feasible, but also greatly reduces the size of the certificates that must be checked on the consumer's end. Since our approach is based on a rewriting logic semantics specification of the full Java 1.4 language [17], the methodology developed in this work can be easily extended to cope with exceptions, and multithreading since they are considered in the Java rewriting logic semantics.

REFERENCES

- [1] M. Alba-Castro, M. Alpuente, and S. Escobar. Automated certification of non-interference in rewriting logic. In D. D. Cofer and A. Fantechi, editors, *Formal Methods for Industrial Critical Systems, 13th International Workshop (FMICS 2008), Revised Selected Papers*, volume 5596 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2009.
- [2] M. Alba-Castro, M. Alpuente, and S. Escobar. Abstract certification of global non-interference in rewriting logic. In F. S. de Boer, M. M. Bonsangue, S. Hallerstede, and M. Leuschel, editors, *Proc. 8th Int. Symp. Formal Methods for Components and Objects (FMCO 2009), Revised Lectures*, volume 6286 of *Lecture Notes in Computer Science*, pages 105–124. Springer Verlag, 2010.
- [3] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *17th IEEE Computer Security Foundations Workshop (CSFW-17)*, pages 100–114. IEEE Computer Society, 2004.
- [4] S. Chong and A. C. Myers. Language-based information erasure. In *18th IEEE Computer Security Foundations Workshop (CSFW-18)*, pages 241–254. IEEE Computer Society, 2005.
- [5] S. Chong and A. C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 98–111. IEEE Computer Society, 2008.
- [6] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 51–65. IEEE Computer Society, 2008.
- [7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
- [8] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. of Sixth ACM Symp. on Principles of Programming Languages*, pages 269–282, 1979.
- [9] A. Darvas, R. Hahnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *2nd International Conference on Security in Pervasive Computing (SPC 2005)*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer Verlag, 2005.
- [10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977.
- [11] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. JavaRL: The rewriting logic semantics of Java. Available at http://fsl.cs.uiuc.edu/index.php/Rewriting_Logic_Semantics_of_Java, 2007.
- [12] R. Hansen and C. Probst. Non-interference and erasure policies for java card bytecode. In *Proc. 6th International Workshop on Issues in the Theory of Security, WITS'06*, 2006.
- [13] S. Hunt and D. Sands. On flow-sensitive security types. In *Conf. record of the 33rd symposium on Principles of programming languages (POPL'06)*, pages 79–90, 2006.
- [14] S. Hunt and D. Sands. Just forget it, the semantics and enforcement of information erasure. In *Proc. 17th European Symposium on Programming, ESOP'08*, volume 4960 of *Lecture Notes in Computer Science*, pages 239–253. Springer Verlag, 2008.
- [15] L. P. L. Jiang and X. Pan. Handling information release and erasure in multi-threaded programs. In *Proc. IEEE Int. Conf. on Computational Intelligence and Security*, pages 824–828, 2007.
- [16] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
- [17] J. Meseguer and G. Rosu. The rewriting logic semantics project. *Theoretical Computer Science*, 373(3):213–237, 2007.
- [18] G. C. Necula. Proof carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Annual Symposium on Principles of Programming Languages POPL 1997, Paris, France*, pages 106–119, 1997.
- [19] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.
- [20] A. Sabelfeld and D. Sands. Declassification: dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
- [21] M. Warnier. *Language Based Security for Java and JML*. PhD thesis, Radboud University Nijmegen, 2005.