Automated Abstract Certification of Non-interference with object aliasing in Rewriting Logic

Mauricio Alba-Castro

Abstract*—*Non–interference **is a semantic program property that assigns confidentiality levels to data objects and prevents illicit information flows from high to low security levels. In this paper, we extend a certification technique for confidentiality of Java classes regarding non–interference, in order to consider objects and object aliasing. The technique is based on rewriting logic, which is efficiently implemented in the high-level programming language Maude. Starting from a previous Java abstract semantics specification written in Maude, we develop an information flow sensitive Java semantics that allows us to observe global non-interference properties, with object aliasing. In order to achieve a finite state transition system, we develop an abstract Java semantics that we use for secure and effective confidentiality analysis. We have implemented our methodology and developed some experiments that demonstrate the feasibility of our approach.**

Keywords*—*Computer Security, Non–interference, Declarative Programming, Software engineering.

I. Introduction

Confidentiality is a property by which information that is related to an entity or party is not made available or disclosed to unauthorized entities. A user might establish a confidentiality policy by stipulating that no data that is visible to other users be affected by confidential data. Such a policy 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. A security policy of this sort is called a *non-interference policy* because confidential data should not interfere with publicly observable data. Program non–interference is a high–level security property that guarantees that there is no illegal information flow through program execution and that the *confidentiality* of secret data is kept [19]. To ensure that a program adheres to a non-interference policy means to analyze how information flows within the program.

The mechanism for transfering information through a computing system is called a *channel*. Variable updating, parameter passing, dynamic object creation, 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* 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* because it depends on the value of the condition that guards the control structure. We are interested in both explicit and implicit flows for non-interference with object aliasing analysis. 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 channels.

Non–interference policies label data objects with their confidentiality levels (usually two levels, `High` and `Low`) and allow *only* information flows from `Low` data objects to `High` data objects. In order to express confidentiality policies, we use the syntax of JML, *Java modeling language* [16], which is a property specification language for Java modules. The JML annotations may include the modifier `model` within fields, methods and classes declarations. The `model` declarations are used for specification purposes only and cannot appear in executable Java code. The initial confidentiality level of a variable in a Java program is written with the word `setLabel` as a `model` JML annotation (e.g. `setLabel(var,High)`). The confidentiality label of program variables is `Low` if nothing is specified (i.e., program variables are public by default). These JML-like annotations, together with the default assumption, define the labeling function of the non–interference policy.

Example 1: Consider the following Java program borrowed from [12] that models a bank account:

```
public class Account { int balance;
  //@ setLabel(balance, High);
  public boolean extraService;
  public Account(int initialamount) {
  //@ setLabel(initialamount, High);
    balance = initialamount; extraService = false; }
  public void writeBalance(int amount) {
  //@ setLabel(amount, High);
    if (amount>=10000) extraService=true;
    else extraService=false; balance = amount; }
  private int readBalance() {return balance;}
  public boolean readExtra() {return extraService;}
}
```

This non-interference policy specifies that the object field `balance` holds secret data and that the formal parameters `initialamount` and `amount` of the object methods also hold secret data. The method `writeBalance` has implicit and

Mauricio Alba-Castro, Computer Science Department, Universidad Autónoma de Manizales UAM, Manizales, Colombia, (email: malba@autonoma.edu.co).

978-1-4799-6717-9/14/$ 31.00 ©2014 IEEE

explicit flows [1]. 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` (namely whether `amount` >= 10000).

A pointer is an object reference. A pointer alias means that there are more than one reference to a given object. Most non–interference works assume that the object references -i.e. the pointers to objects- are opaque [6], [7]. This means that we don't know their values. The attacker can only see the object to which it points, and at most she can test pointer equality [5], [14]. Pointer aliasing happens in many Java programs, and may leak confidential information, as shown by the next example.

Example 2: Consider a simple Java version of an example borrowed from [4]. In this example we have two objects of class `Z` whose references are stored in the variables `p` and `q`, which are assumed `Low`–labeled by default. The objects have a `Low`–labeled field (`info`). In line `A`, the program creates an object alias in the variable `z`. The annotation `setLabel` sets the confidentiality label of the variables `z` and `h` to `High`.

```
class Safe1NIAlias03  {
  public static void main(String[] args) {
  Z  z;       //@ setLabel(z, High);
  int h = 8;  //@ setLabel(h, High);
  Z p,q; boolean l;         // Low
  p = new Z();  // object 1
  q = new Z();  // object 2
  if (h > 7)             // LINE A
     z = p;     // z aliases p
  else z = q;   // z aliases q
  z.info = 42;           // LINE B
  if (p.info == 42)      // LINE C
     l = true;    // h > 7
  else l = false; // h <= 7
  System.out.println(l);  }  }
class Z {    int  info; // Low  }
```

The program in line `A`, creates and stores an object alias of an object reference, either `p` or `q`, in a `High`–labeled variable `z`, depending on the value of the `High`–labeled variable `h`. The implicit flow in line `A` is licit because `z` and `h` are both `High`–labeled variables. In line `B`, the `Low`–labeled field `info` is licitly updated with a constant (all program constants are `Low`–labeled) using the `High`–labeled reference `z`. In line `C`, the `Low`–labeled variable `l` is updated by a constant, either `true` or `false`, depending on the value of the `Low`–labeled field `info` of the object referenced by `p`. In spite of the fact that implicit flow of line `C` is licit because of the `Low`–labeled expression guard, after line `C` execution, we have in `l` information of the secret variable `h`.

In [2], we proposed an abstract methodology for certifying Non-interference of Java source code, as a safety property. It is based on *Rewriting logic* (RWL) and is implemented in Maude [10], which is a high-performance language that implements RWL.

The methodology of [2] is the following. 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. The Java semantics is a specification in Maude. Given a safety property the unreachability of the system states that denote 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, we achieve a finite search space by using abstraction [11]. 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 full-fledged formulation of the abstract global non–interference certification methodology of [2] to consider objects with confidentiality level labels and object aliasing, which allow us to analyze confidentiality of more realistic programs.

We provide a clear–cut information flow sensitive semantics of Java programs that deal with non–interference, and object aliasing. This semantics is formulated as an extension of the operational Java semantics of [13] written in Maude. Such a new information flow semantics provides a *unifying* model for dealing with non–interference of programs with objects and object aliasing as safety policies. We provide an abstract, finite-state version of the information flow operational semantics which supports finite program verification. Thanks to the different handling of rules and equations in Maude, our methodology does not suffer from the state–space explosion of more traditional approaches. Our Java certification methodology can be applied to existing Java programs simply by inserting the confidentiality annotations of the desired policy. As a by–product of the verification, a certificate is delivered which consists of a set of (abstract) rewriting proofs that can be easily checked by the code consumer using a standard rewriting logic engine. For the best of our knowledge, this is the first adaptation of the PCC principle to static analysis of non–interference and object aliasing based on term rewriting . We have implemented this framework, as an extension of the Non–interference with and without erasure framework of [3].

The paper is organized as follows. Sections II and III introduce, respectively, the non-interference policy, and the rewriting logic semantic specification of Java we have used. Section IV summarizes the proposed extended rewriting logic data flow-sensitive semantic specification of Java that include objects with confidentiality levels together with fields with confidentiality levels, dynamic object creation, method invocations, and object aliasing. Section V describes the abstract semantics that corresponds to the concrete extended version introduced in Section IV. Section VI discuss related work and Section VII draw some conclusion and future work.

[1]A *explicit flow* is associated to an assignment or memory write, e.g. the assignment "`balance` = `amount`". An *implicit flow* is associated to a control flow guard expression using variables with a high confidentiality level, e.g. "`if` (`amount` >= `10000`) `extraService` = `true`; `else` `extraService` = `false`;".

II. Non–interference Policies

A non-interference policy establishes a confidentiality level for each source program variable of primitive datatypes. It guarantees that actual values of variables with a higher confidentiality level do not influence the output of a variable with a lower confidentiality level during program execution [19]. It is implicitly assumed that constants that appear in a program always have the lowest confidentiality level (i.e., the considered program is authorized to access secret data, but it does not contain secret data in its code).

A non-interference policy can be represented [2] by a *partially ordered set* $\langle Labels, \leq \rangle$ and a labeling function $Labeling : Var \rightarrow Labels$, 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 [15]. There are usually two confidentiality levels: $Labels = \{\texttt{Low}, \texttt{High}\}$. These represent public non-secret data (low confidentiality) and secret data (high confidentiality), respectively. $\langle Labels, \leq \rangle$ forms a lattice where `Low` is the greatest lower bound or *bottom* element ($\bot$), `High` is the least upper bound or *top* element ($\top$), $\texttt{Low} \leq \texttt{High}$, and $\texttt{High} \nleq \texttt{Low}$. The *join* operator ($\sqcup$) is defined as $\texttt{Low} \sqcup \texttt{Low} = \texttt{Low}$; otherwise, $X \sqcup Y = \texttt{High}$. Enforcing non-interference means that the values of `Low`-labeled source variables can flow to `High`-labeled source variables. However, it also means that the values of `High`-labeled source variables cannot flow to `Low`-labeled source variables. The attacker model for global non–interference that we formalize below 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 (i.e., temporal security breaches are only relevant if they influence a variable that can be observed at the final state).

We assume a fixed Java program P_{Java}. $Vars(P_{\text{Java}})$ denotes the set of variables of P_{Java}. We denote the set of *Low* program variables as $Low(P_{\text{Java}}) = \{var \in Vars(P_{\text{Java}}) \mid Labeling(var) = \texttt{Low}\}$. A program state St is a set of value assignments to program variables. Given $var \in Vars(P_{\text{Java}})$ and a state St, $St[var]$ denotes the value of variable var in St. If the variable var does not exist in the state St, then $St[var]$ is not defined. 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 do not consider Java threads, therefore only deterministic Java programs are analysed. $\mapsto_{\text{Java}}$ is the transition relation that describes any possible one-step transition between any two Java program states. An *execution* (or trace) of P_{Java} is a sequence $\langle P_{\text{Java}}, St_0\rangle \mapsto_{\text{Java}} \cdots \langle P_i, St_i\rangle \mapsto_{\text{Java}} \cdots \mapsto_{\text{Java}} \langle \checkmark, St_n\rangle$, which is simply denoted by $\langle P_{\text{Java}}, St_0\rangle \mapsto^*_{\text{Java}} \langle \checkmark, S_n\rangle$ if the intermediate states are irrelevant. We can also abbreviate $\langle \checkmark, S_n\rangle$ by $\langle S_n\rangle$.

We define in [2] program non–interference by using an equivalence $=_{Low}$ 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 $St_1 =_{\texttt{Low}} St_2$, if for all $var \in Low(P_{\text{Java}}), St_1[var] = St_2[var]$.

What the attacker can see from a final state is determined by a relation $\approx_{\texttt{Low}}$. Two executions of a program P_{Java} are related by $\approx_{\texttt{Low}}$ if they are indistinguishable to the attacker [19]. The notion of non–interference is therefore parametric on $\approx_{\texttt{Low}}$, which defines the attacker capabilities. A program is non–interferent if, whenever different initial program states are indistinguishable at level `Low`, this implies that the corresponding final states are also indistinguishable at level `Low`.

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 program states St'_1, St'_2 such that $\langle P_{\text{Java}}, St_1\rangle \mapsto^*_{\text{Java}} \langle St'_1\rangle$, $\langle P_{\text{Java}}, St_2\rangle \mapsto^*_{\text{Java}} \langle St'_2\rangle$, we have that $St_1 =_{Low} St_2 \Rightarrow St'_1 \approx_{Low} St'_2$.

We follow in [2] the standard approach in the literature that considers $St \approx_{Low} St'$ iff $St =_{Low} St'$. Then, the non–interference condition of Definition 2 is understood as the lack of any *strong dependence* of `Low`-labeled variables on any of the `High`-labeled variables [19].

Non–interference was characterized as an hyperproperty [9]. This means that it cannot be analysed by checking sets of program traces. Instead, it have to be analysed by checking sets of sets of program traces.

For instance, the verification process for non-interference should check the (possibly infinite) set of (possibly infinite) sets of final states issued from the (possibly infinite) sets of indistinguishable initial configurations. In contrast, the verification process for a safety property should simply check the traces issuing from the (possibly infinite) set of initial configurations, which is much simpler.

III. The Rewriting Logic Semantics of Java

In the following, we briefly recall the rewriting logic semantics of Java that was originally given in [13] .

In [13], 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 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 $u \rightarrow^r_{\text{Java}} v$ the fact that the concrete

```
---Obtain variable location and evaluate expression
eq k(Var = E -> K) env([Var, Loc] Env) =
   k(E -> =(Loc) -> K) env([Var, Loc] Env) .
---Assign value to location
eq k(Val -> =(Loc) -> K) =
   k([Val -> Loc] -> (Val -> K)) .
---General procedure to update the memory
eq k([Val -> Loc] -> K) store([Loc,Val'] ST) =
   k(K) store([Loc,Val] ST) .
```

Fig. 1. Continuation-based equations for the Java assignment operator

terms u, v (which denote Java program states) are rewritten (at the top position) 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 $u_1, \ldots, u_k$ such that $u \rightarrow_{\text{Java}} u_1 \rightarrow_{\text{Java}} u_2 \cdots u_k \rightarrow_{\text{Java}} v$).

The rewrite theory $\mathcal{R}_{\text{Java}}$ is defined on terms of a concrete sort State, with the main state attributes (represented by means of constructor symbols of the algebraic type State) such as in, out, fstack for handling function calls, lstack for handling loops, env for assignments of variables to memory locations, and store for assignments of memory locations to their actual values. They define an algebraic structure that is parametric w.r.t. a generic sort Value that defines all the possible values returned by Java functions or stored in the memory. For instance, the int and bool constructor symbols describe Java integer and boolean values and are defined in Maude as "op int : Int → Value ." and "op bool : Bool → Value .", where Int and Bool are the internal built–in Maude sorts that define integer and boolean data types. 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).

The semantics of Java is defined in a *continuation-based style* 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 -> k) is evaluated, its result will be passed on to the remaining continuation k. The semantics of the assignment operator for the Java variables is specified in Figure 1.Due to space limitations we do not discuss here object creation neither heap manipulation.

IV. Analysing Non–interference with object aliasing by using an Extended Instrumented Semantics

In previous paper [2], we prove 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. We consider simple valued variables, i.e. primitive type variables, integer, character, and so on. Intuitively, the semantic instrumentation is defined as follows: i) attach a confidentiality label to each memory location; this allows us to observe their confidentiality level at the final execution state; ii) attach a confidentiality label to the evaluation of program expressions; this allows us to know whether the evaluation of an expression involves high confidentiality data. iii) associate a confidentiality label to the evaluation of program statements, particularly those involving conditional expressions or guards; this allows us to determine whether the control flow at a given execution point depends on the actual value of high confidential variables; However, this label is not attached to each program statement; rather it is kept as an extra attribute of states in the extended Java semantics; this corresponds to the notion of a *context label* being updated after each evaluation step in [15].

This extended semantic keep track of the initial and final confidentiality level label of memory locations, and it allows us to check whether a secret value from some High–labeled variable is stored in the memory location of a public Low–labeled variable when program ends execution.

One solution to the problem of object aliasing depicted by Example 2 is to forbid updating of Low–labeled fields of High–labeled objects in every case, as proposed in [5]. This solution is sound but imprecise. There are non–interferent programs that do not have object aliasing and update Low–labeled fields of High–labeled objects. A precise solution should do aliasing analysis in order to forbid updating of Low–labeled fields in High–labeled objects, only if the updated object has an alias with different confidentiality label.

In order to consider objects and pointer alias, we extend this framework as follows: i) attach confidentiality labels to the memory locations that store objects; ii) attach a confidentiality label to the evaluation of class field access expressions; the label of expression z.info in Example 2 is *Labeling*(z) ⊓ *Labeling*(info), where *Labeling*(z) is the confidentiality label of the reference z and *Labeling*(info) is the confidentiality label of the field info that corresponds to the object referenced by z as in [5]; iii) associate a confidentiality label to the evaluation of method invocations; given that acc is a reference to an Account object of Example 1, the command acc.writeBalance(100); temporarily stores and sets up the context level to the confidentiality label of the reference acc; when method ends execution the context label is restored; iv) analyse pointer aliasing when variable updating; in case of updating of a Low–labeled field in High–labeled objects, we check if the object has a reference alias with different confidentiality label; in this case, we set up the confidentiality label of the updated field to the joining of confidentiality label of the object and the confidentiality label of the expression, as in [5]; after executing the assignment z.info = 12 in line B of Example 2, the new confidentiality label of field info is *Labeling*(z) ⊓ *Labeling*(12), i.e. High; then, the guard expression p.info in line C has the label Low ≫ High, which means that there is an implicit illicit flow, and the program is interferent.

The extended semantic specification for the memory updating of Java variables with object aliasing is shown in Figure 2. Figure 3 shows the final state of the extended computation of Example 2. The store keep the location, the value and the id of the owner thread (a triplet). The extended value is a triplet that consist of the concrete value of the variable, together with it's confidentiality level label and the erasure policy if any. The equation is used when executing the assignment z.info = 12 in line B of Example 2. L is the location (l(7)) of the object

```
---Memory updating with aliasing
ceq t(k( < V, SLab2  ,  Pol2 > -> (obLoc ( L' ) ->
     (=(L) ->   K))) TC)
  store([L', < o(OA), Lab', Pol'  >, I' ]
        [L'', < o(OA), Lab'' , Pol''  >, I'' ] ST)
 =  t(k([< V, SLab2 join Lab'  ,  Pol2 join Pol' >
      -> L] -> (< V, SLab2  ,  Pol2 > -> K))  TC)
  store([L', < o(OA), Lab', Pol'  > , I'  ]
        [L'', < o(OA), Lab'' , Pol''  > , I'' ] ST)
 if L' =/= L'' /\ Lab' =/= Lab''
```

Fig. 2. Continuation-based equation for the extended Java assignment operator

```
out(pl(< bool(true),Low >> High,nopol >))
store(
[l(2),< o(f([t(t('Z)),f(['info,l(7)])]) curr(t('Z))
        orig(t('Z))),High,nopol >,0]
[l(4),< o(f([t(t('Z)),f(['info,l(7)])]) curr(t('Z))
        orig(t('Z))),Low,nopol >,0]
[l(6),< bool(true),Low >> High,nopol >,0]
[l(7),< int(42),Low >> High, nopol >,0])
```

Fig. 3. Partial final state of example 2

field `info`, $\mathtt{L}'$ is the location (`l(2)`) of variable `z` which is a pointer to the object, and $\mathtt{L}''$ is the location (`l(4)`) of variable `p` another object pointer (that in this case also references the same object, i.e. the object alias), and `OA` are the object attributes. The constructor `obLoc` is used to keep the location of the object whose field is being updated.

Note in Figure 3 that the output value has the label $\mathtt{Low} \gg \mathtt{High}$ which means that a `High` labeled value was stored in the `Low` labeled variable `l` in location `l(6)`. Note also that variable `z` (with location `l(2)`) and variable `p` (`l(4)`) both references the same object whose field `info` is stored in location (`l(7)`). The integer value of the field `info` (i.e. `int(42)`) are also labeled $\mathtt{Low} \gg \mathtt{High}$, which means precisely that another illicit information flow happens in this example.

Here we recall our novel characterization of non-interference as a safety property based on the sensitive flow extended semantic of [2]:

Definition 3 (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 $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E\rangle$, we have that for all $var \in Low(P_{\text{Java}})$, $St_2^E[var] = \langle Val, \mathtt{Low}\rangle$ for a value Val.

The soundness proof of our extended semantic with pointer aliasing is similar [2] to the soundness proof of the extended semantics of [2] whose details can be founded in [1] then omitted here.

V. Approximating Non–interference with pointer aliasing by using an Abstract Semantics

In the following, we develop an abstract, rewriting logic Java semantics that allows us to statically analyse global non–interference with objects and pointer aliasing. Similar to [2], the purpose of the abstract semantics is to correctly approximate the extended computations in a finite way. Given the extended Java semantics, where there are concrete labeled values, we simply get rid of the values in the abstract semantics, and use their confidentiality labels as the abstract values instead. There is one important exception to this: to abstract a dynamic object creation, we do keep the object but remove the data from its fields. However, we still maintain the confidentiality labels of each field.

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 $\mathcal{R}_{\text{Java}^{\text{E}}}$ (taking advantage of its modularity) by abstracting the domain to $(Labels \cup LabelChange)$ and introducing approximate versions of the Java constructions and operators tailored to this domain.

In this section, our abstraction function $\alpha : \wp(\mathsf{State}^E) \rightarrow \wp(\mathsf{State}^E)$ is a simple homeomorphic extension to sets of states of the function $2nd$: $\mathsf{Value}\times$ $(Labels \cup LabelChange) \rightarrow (Labels \cup LabelChange)$, meaning that we disregard the actual values of data.

A program is non–interferent for a given labeling function if the abstract values (the confidentiality labels) of the *Low* variables in the final state of an abstract program execution do not have the label $\mathtt{Low} \gg \mathtt{High}$ [2]:

Theorem 1 (Abstract Non-Interference Soundness): Given a Java program P_{Java}, P_{Java} is non–interferent (Definition 2) if for all $SSt_1 \in \wp(\mathsf{State}^E)$ s.t. $\langle P_{\text{Java}}, SSt_1\rangle \mapsto^*_{\text{Java}^\#} \langle SSt_2\rangle$, for all $St \in SSt_2$, and for all variables $var \in Low(P_{\text{Java}})$, either $St[var] = \mathtt{Low}$, or $St[var] = \langle Object, \mathtt{Low}\rangle$ for an object $Object$.

The soundness of our abstract interpretation is similar to [2]. First, we have to consider the soundness of the abstraction function α. This is done by proving that this abstraction function and a corresponding concretization function, both constitute a Galois insertion[11]. Then we have to proof the soundness of the abstract computation, i.e. that all extended program traces and states have corresponding abstract program traces and states such that no extended program trace (state) is disregarded. The proof is by induction on the length n of the extended program trace or rewriting sequence denoted by $\langle P_{\text{Java}}, St_1^E\rangle \mapsto^*_{\text{Java}^{\text{E}}} \langle St_2^E\rangle$ (n is also the length of the corresponding abstract program trace $\langle P_{\text{Java}}, SSt_1\rangle \mapsto^*_{\text{Java}^\#} \langle SSt_2\rangle$). The details of the two proofs are similar to Chapter 6 in [1] hence omitted here.

The certification methodology presented here has been implemented in Maude. The prototype system has been tested by using examples (and variations on them) with pointer aliasing borrowed from [4], [5], and it is able to check confidentiality global program properties related to non–interference with pointer aliasing analysis. Here http://wp.me/p3E3QQ-1E, we have the extended and abstract Java semantic and several examples, together with their corresponding traces and certificates.

VI. Related Work

Most works about non-interference and object aliasing propose sensitive-flow labeled type systems unlike the work [4] which uses a Hoare-like logic based information flow analysis,

[2]When object variable updating, we just consider the additional case where the object to be updated has an alias.

and ours, based on rewriting logic. There are many works that combine static and dynamic analysis [6], [17] while some others use only static analysis [5], [14], [7] as our approach. The Jif proposal [17], a Java extension for information flow, approach to non–interference , together with our's are implemented, unlike the above cited. However, Jif doesn't consider object aliasing, and it haven't a soundness proof because of its dynamic analysis. Unlike the approaches in [4], [17] and our's, most works don't allow temporary breaches: it doesn't accept secure programs that have interferent subprograms. We consider that different objects may have different security levels even objects of the same class, unlike the proposals [5], [6]. In order to control field updating regarding object aliasing, most type-system based proposals uses a local method context security level as a lower bound of the object fields that can be updated, called the heap effect [5], [6], [7]. Unlike [14], we and most works assume opaque pointers. The works [7], [17] considers exception analysis while most, including ours, it doesn't. The considered realistic languages are: Jif, a Java extension, a subset of the JVM [7], [8] and Java (our proposal). However, the work [8] do not include method calls, exceptions and threads. The approach of [4] abstracts sets of concrete memory and heap locations into abstract locations that have abstract addresses. This abstraction allows them to analyse programs with an unbounded number of object instances.

Our proposal considers that objects and object fields have security levels, objects of a given class can have different security levels, we can deal with object creation, method calls, allows temporary breaches, and unlike all works above, it can generate a formal proof, or a counterexample, regarding program security. However we cannot handle programs with threads, exceptions and unbounded number of object instances, neither to analyse non opaque pointers.

VII. Conclusion

In this paper, we extend a framework for automatically certifying global non–interference of Java sequential programs with objects and regarding object pointer aliasing, but assuming opaque pointers and a fixed size memory. The proposed framework fully accounts for explicit as well as implicit flows, and allows the inference of rewriting logic safety proofs, thus providing support for the code producer in 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.

Since our approach is based on a rewriting logic semantics specification of the full Java 1.4 language the methodology developed in this work can be easily extended to cope with exceptions, heaps, and multithreading since they are considered in the Java rewriting logic semantics.

Future work includes relaxing opaque pointers assumption, abstraction of memory locations, programmed exceptions, threads and considering declassification policies (controlled ways of downgrading secret information)[20].

References

[1] M. Alba-castro. *Abstract Certification of Java Programs in Rewriting Logic*. PhD thesis, Technical University of Valencia UPV, DSIC, 2011. Available at: http://dialnet.unirioja.es/servlet/tesis?codigo=24228.

[2] M. Alba-Castro, M. Alpuente, and S. Escobar. Abstract certification of global non-interference in rewriting logic. In *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, Berlin Heidelberg, Germany, 2010.

[3] M. Alba-Castro, M. Alpuente, and S. Escobar. Approximating non-interference and erasure in rewriting logic. In *12th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC 2010) Sept. 23-26, Timisoara, Romania*, pages 124–132, Los Alamitos, CA USA, 2010. IEEE Computer Society.

[4] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 91–102, New York, NY, USA, 2006. ACM.

[5] A. Banerjee and D. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 239–253, Los Alamitos, CA USA, 2002. IEEE Computer Society.

[6] A. Banerjee and D. Naumann. Stack-based access control and secure information flow. *Functional Programming*, 15(2):131–177, 2005.

[7] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *Proc. 16th European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, Berlin Heidelberg, Germany, 2007.

[8] F. Bavera and E. Bonelli. Type-based information flow analysis for bytecode languages with variable object field policies. In *Proceedings of the 2008 ACM symposium on Applied computing (SAC '08)*, pages 347–351, New York, NY, USA, 2008. ACM.

[9] M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Proc. IEEE 21st Computer Security Foundations Symp. (CSF'08)*, pages 51 – 65, Los Alamitos, CA USA, 2008. IEEE Computer Society.

[10] 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, Berlin Heidelberg, Germany, 2007.

[11] 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, New York, NY, USA, 1979. ACM.

[12] A. Darvas, R. Hahnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In *second international conference on Security in Pervasive Computing(SPC2005)*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer-Verlag, Berlin / Heidelberg, Germany, 2005.

[13] 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.

[14] D. Hedin and D. Sands. Noninterference in the presence of non-opaque pointers. In *Proc. 19th IEEE Computer Security Foundations Workshop (CSFW'06)*, pages 217–229, 2006.

[15] 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, New York, NY USA, 2006. ACM.

[16] G. Leavens, A. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31:1–38, May 2006.

[17] G. Malecha and S. Chong. A more precise security type system for dynamic security tests. In *Proceedings of the 5th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security PLAS 10 June 10, 2010 Toronto, Canada*. ACM New York.

[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, New York, NY, USA, 1997. ACM.

[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.