Class-level Non-Interference

Damiano ZANARDINI

CLIP, Universidad Politécnica de Madrid
E-28660 Boadilla del Monte, Madrid, SPAIN

damiano@clip.dia.fi.upm.es

Received June 2nd, 2011

Abstract The Information-Flow property of *Non-Interference* was recently relaxed into *Abstract Non-Interference* (ANI), a weakened version where attackers can only observe *properties* of data, rather than their exact value. ANI was originally defined on integers, where a property models the set of numbers satisfying it. The present work proposes an *Object-Oriented*, *Java*-based formulation of an instance of ANI where data take the form of objects, and the observed property comes to be their type. The execution of a program is taken to be the invocation of some method by an external user: a class is *secure* if, for all its (non-private) methods, the type of their *low-security* data after the execution does not depend on the initial type of its *high-security* data (i.e., there are no *illicit flows*). The relation to ANI theory (in its *abstract* version) can be seen in the representation of abstract domains in terms of class hierarchies: an *upper closure operator* map an object into the *smallest* class it is an instance of. An analyzer for a non-trivial subset of Java is illustrated. A sketch of a *soundness* proof is provided: a program is never misclassified as secure, i.e., it is rejected whenever the absence of illicit flows cannot be guaranteed.

§1 Introduction

Abstract Non-Interference (ANI) [10] provides a well-founded and parametric framework where the standard information-flow notion of *Non-Interference* (NI) [13, 20] can be relaxed. Such a weakening can be useful because many programs which should be regarded as reasonable do not satisfy NI, as the latter requires the separation between

low-security — i.e., information every user can observe — and *high-security*[*1] — i.e., data which have to be protected from unauthorized users — data to be complete. This means that Non-Interference requests that the low-security part of the output must not depend in any way on the high-security part of the input (in other words, there are no *illicit flows* from *high* data to *low* data). In practice, it is often the case that some flows should be allowed, as long as (i) attackers cannot detect them; or (ii) there is no need to protect some *aspects* of the information which is leaked.

Several techniques have been proposed in order to weaken NI. Existing approaches either limit the *observational power* of attackers, or *declassify* [27, 6] the released information. Abstract Non-Interference belongs to the first family: it deals with attackers which can only observe *properties* of data — instead of exact, *concrete* data. Therefore, an illicit flow may be invisible to attackers since the property they can observe does not change: such a flow should be considered as harmless, and the program should be safely accepted as *secure*. ANI describes properties as abstract domains [8], i.e., mappings which only distinguish between values if they differ with respect to the property itself.

1.1 Main contribution

Abstract Non-Interference initially referred to a simple imperative language IMP with global, static variables, and integers as the only data type. Data properties are *sets of values*: e.g., "to be even" is the set of even numbers, so that the corresponding abstract domain of parity maps a number to "even" or "odd" depending on its parity. The present work defines information flow in an Object-Oriented (OO) framework, modeled on a Java-like language, Java$_{IF}$ (Section 2.1), where values take the form of objects [*2]. The main idea is to see a class, which represents a collection of objects with the same internal structure, as an *abstract property* [7]: checking if an object satisfies a property amounts to observing its type — or, equivalently, the class it is an instance of. Accordingly, *class hierarchies* describe *abstract domains*, the subclass relation being the partial ordering on abstract values (i.e., abstract properties).

It must be pointed out that type/class information has always been very useful in program analysis [17, 22]. Moreover, types are a powerful tool in program design, where the developer often uses types as way to statically guarantee, to some extent, the correct behavior of his or her code. E.g., in the Haskell functional language, *type synonyms*

[*1] Most works on Information-Flow analysis use *private* and *public* to denote the two levels of security. However, in the present article, the terms *high-security*, or *high*, and *low-security*, or *low*, are used instead, in order to avoid confusion with Java modifiers (see Section 2.5).

[*2] A discussion on *primitive types* can be found in Section 3.2.

and related techniques allow to build a very articulated type system which makes it possible to check statically the absence of a number of runtime errors. Similarly, in an Object-Oriented framework, classes can be used to represent the fact that a piece of data satisfies a given property. To this end, it could be reasonable to declare two classes C_1 and C_2 with the same internal structure, but such that an object is an instance of C_1 (resp., C_2) if and only if it satisfies the property P_1 (resp., P_2). Reasoning about classes in the context of Information-Flow analysis goes precisely in this direction: to enforce the promising approach of Abstract Non-Interference by considering types as properties of data. The presented framework is a special case of the parametric model of Abstract Non-Interference, where the observational power of attackers is instantiated to be the type of objects.

A second contribution is a model for Information Flow in Object-Oriented languages (Section 2.5). The paper applies it to Abstract Non-Interference, but the model can be, in principle, considered when dealing with standard Information Flow. The model specifies what is a program, what is attacker, and how they are supposed to interact with each other. Clearly, it is not the only model which can be conceived for the study of this problem; yet, we believe it is a reasonable proposal.

A third contribution is the definition of an *algorithmic approach* to this specific instance of Abstract Non-Interference. An analyzer is shown, which checks ANI relying on *class-based dependencies* in Java$_{IF}$. Its purpose is to check whether the output type of low-security fields depends on the initial type of high-security fields. If this happens, then an illicit *information flow* is said to occur, and the program must be rejected as insecure. The algorithm is sound (a sketch of the soundness proof is provided), i.e., programs are always rejected if it is not possible to guarantee that illicit flows will never occur. As shown in the examples, there are cases in which programs are safely detected as secure even though the classical notion of NI is not satisfied (this means that the security property of interest has been relaxed w.r.t. Non-Interference).

1.2 Related work

Standard Information Flow [13, 20, 3] for an Object-Oriented programming language considers data propagation from *high-security* (often called *private*) to *low-security* (*public*) fields. This security property is enforced by means of types [15] or logic [2], the latter approach being potentially more precise, at the cost of being more difficult and expensive. The present work enforces a different security property — which can be seen as a weakening of Standard Information Flow; see the rest of this section — for which, in some sense, a different approach is required. Unlike the type-based

technique, it is not always *compositional* (Section 4.3); this is admitted in order to improve the precision of the result by taking advantage of specific qualities of the security property under study.

The foundational work on ANI [10] provided a set-theoretic definition of *secrecy* in a simple imperative language. Attempts have been made to make ANI algorithmically verifiable; a compositional *proof system* was proposed [11], which checks ANI by inferring secrecy assertions via *Hoare triples*. Assertions are combined syntactically to derive safety proofs, and the technique also outlines how to derive attackers which do not violate a given security policy. ANI has been extended to higher-order functional languages [23]. In that work, some ideas for an analyzer were provided as a type system: data were equipped with *security types* describing their location w.r.t. the high-low boundary. Hence, the same discussion on compositionality holds when comparing that idea with the present paper. Moreover, there are a number of issues concerning the Object-Oriented paradigm which make a difference between the two works. For example, the system *attacker*+*attacked program* was not described in the former work: in that framework, a functional program simply runs at the same time as an attacker tries to somehow get information from spying its execution, but it is not specified where the attacker "lives". On the other hand, the latter approach provides a reasonable characterization of the execution environment where the program and the attacker interact (Section 2.5). Finally, the earlier proposal deals with numerical and functional (on numbers) values, while the new Object-Oriented approach takes program classes into account and ignores primitive types. Overall, we believe that, on the strictly technical side, the relation between the two papers is quite loose. Another previous paper by the author [24] deals with Java bytecode; however, that work was quite limited, since it only used simple numerical domains and did not consider many features which are specific to OO languages.

Type/class inference for Object-Oriented languages [17, 22] aims at verifying that data cannot belong, at runtime, to the wrong class, or at detecting that a *virtual* call can be optimized by the compiler to a *non-virtual* (i.e., already resolved at compile-time) one if the runtime class of the caller is statically decidable to be unique. The connection between type information and data properties has been underlined [7] in the framework of Abstract Interpretation [8]: a type is seen as an abstract property, and type systems can be described as abstract semantics of programs.

Since Information Flow and Data Dependencies [5, 1] are closely related, ANI involves a notion of *abstract Data Dependency* [19], whose computation in the general case has been recently discussed in the framework of Program Slicing [14, 26].

To the best of our knowledge, the content of Section 2.5 is new.

A first version of the present article appeared in conference proceedings [25]. It must be merely considered as a preliminary approach to the problem.

1.3 Introductory examples

This section introduces the problem by means of two examples. It must be noted that such examples present scenarios where classes have been declared according to design choices which were not driven by secrecy requirements — in other words, the Information-Flow analysis works on preexisting classes which where designed for other purposes. On the other hand (as mentioned in Section 1.1), one can think of a design which considers *from the beginning* secrecy requirements by enriching the class hierarchy in order to enforce Non-Interference at the level of the properties of interest.

[1] Hiding implementation details

Suppose an abstract class MySet be user-defined, which models a *set* container. The declaration of this container leaves unspecified how to implement the internal data structure data, which contains the elements of the set.

```
public abstract class MySet {
  // the usual methods for getting, setting, etc.
  ...
  // gets all elements in the container
  public abstract Object getElems();
}

// hash-table implementation
public class MySet_Hashtable extends MySet {
  protected Hashtable data;
  ...
  public Enumeration getElems() { return data.elements(); }
}

// linked-list implementation
public class MySet_List extends MySet {
  protected Node data; // the node of the linked list
  ...
  public Array getElems() {
    // returns all the elements as an array
    return data.elements();
  }
}
```

The declaration of each subclass of MySet decides how data is implemented, and provides a specific implementation for the method getElems.

If the goal is to *hide the implementation* of the container from external users — which amounts, in this case, to considering the data field as *high-security* — then this code is ill-designed. In fact, the return type of getElems is Array if data is a Node, and

Enumeration if data is a Hashtable. An external user which calls this method would be able, by observing the type of the return value — e.g., by means of the instanceof operation — to guess if the set has been implemented as an array or as a hash table.

```
public class Attacker {
  public static void main(String args[]) {

    // the attacker cannot directly access the container, but can
    // access the result
    Object ret = getElems();

    if (ret instanceof Enumeration) {
      System.out.println("It's implemented as a hash table");
    } else if (ret instanceof Array) {
      System.out.println("It's implemented as a linked list");
    }
  }
}
```

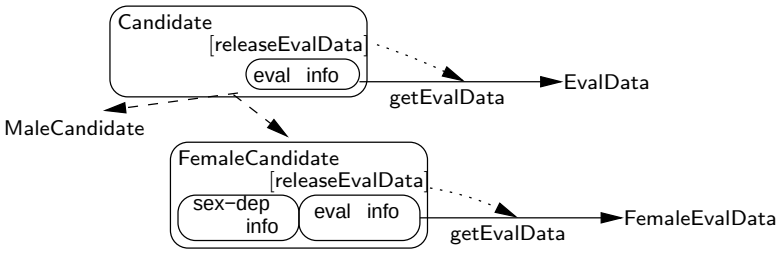
Note that the interest is *not* in protecting the information stored in the set: single elements can be directly fetched by calling getElem with some key. Instead, the information to be kept hidden is the implementation of the data structure: concretely, the type of the high-security data field. As a matter of fact, observing the execution time of some operations on the data structure (or other covert channels) may also leak information about the implementation; however, this is outside the scope of this paper.

[2] Protecting confidential information

Consider the recruitment unit of an enterprise. People applying for a job should fill in their personal data in a questionnaire. Since the evaluation is (partially or totally) manually performed, only a subset of data (experience, skills, spoken languages, etc.) can be used or made visible in the process, in order to prevent the human operator from discriminating candidates on the basis of, say, gender or race. In this example, gender discrimination is considered.

Let personal information be stored in objects of class Candidate (Figure 1). Some data in this class are gender-dependent, and are going to be used, for bureaucratic reasons, only *after* the candidates have been evaluated. This means that such information is present from the beginning in the database of candidates, but should be kept hidden from evaluators. In the spirit of Object-Oriented programming, it makes sense to declare two subclasses FemaleCandidate and MaleCandidate of Candidate in order to consistently and efficiently store gender-dependent data. Information which is

Fig. 1 The introductory example



supposed to be used during recruitment can be fetched by calling the method

$$\mathsf{getEvalData(CandidateID\ id)EvalData}^{*3}$$

which accesses a Candidate object by its id, and returns an EvalData object containing data to be evaluated.

It cannot be excluded that the system has been previously designed for different uses. Therefore, two subclasses FemaleEvalData and MaleEvalData of EvalData may have been implemented. Also, a private method (to be invoked by the method getEvalData(id))

$$\mathsf{Candidate.ReleaseEvalData()EvalData}$$

can have been declared in the superclass, and overridden by

$$\mathsf{FemaleCandidate.ReleaseEvalData()FemaleEvalData}$$
$$\mathsf{MaleCandidate.ReleaseEvalData()MaleEvalData}$$

in subclasses. This choice is completely sensible in an Object-Oriented design. Yet, the return type of getEvalData(id) would reveal gender information. Such code is not ill-designed in itself, but is not adequate if security requirements are set which include forbidding the gender of candidates to be revealed during recruitment.

§2 Preliminaries

This section gives the main notions which are necessary to understand the rest of the paper. Since the language is basically a subset of Java, a basic knowledge of Java is assumed. Section 2.4 merges the content of Sections 2.2 and 2.3 by applying

*3 In this example and in the rest of the paper, the full notation for method signatures will take the form methodName(Param1class param1, ..., ParamKclass paramK)ReturnClass or className.methodName(Param1class param1, ..., ParamKclass paramK)ReturnClass.

Abstract Interpretation to Information Flow. Section 2.5 argues about how to bring ANI to a Java-like framework.

2.1 The Java$_{IF}$ language

The programming language to which the presented framework is applied is a subset Java$_{IF}$ of sequential Java. Using a new name is not really meant to define a new language, but only to distinguish a subset of an existing one for the sake of the analysis.

Java$_{IF}$ includes *classes*, *objects*, and *inheritance*. Each class is declared to contain *static fields* and *methods*, and *instance* fields and methods. It is possible to declare *abstract* classes and abstract methods, with the usual behavior. *Primitive types* comprise integers and booleans (Section 3.2 discusses how primitive types actually do not play a role in the presented framework); *arrays* are also supported.

The public, private and protected *modifiers* are included, with the usual semantics; moreover, Java$_{IF}$ admits two additional *security modifiers* flowH and flowL, which are related to the security level of data. A security modifier can be set either in the field declaration, as an additional modifier, or by means of a separate seclevel declaration. The security level of fields can be readjusted in subclasses.

Example 2.1

```
class C {
  E f;
  // f is set to be low-security after being declared
  seclevel f flowL;
  // the security level can also be given at declaration-time
  private flowH E g;
}
class D extends C {
  seclevel f flowH; // f is high-security when accessed from D
  seclevel g flowL; // g is low-security when accessed from D
}
```

This code states that f is to be considered as low-security when accessed from C, and high-security when accessed from D. As for g, it is the other way around. Both declaration styles are semantically equivalent.

seclevel does *not* create a new field: in the example, the subclass will only have one field f and one field g, both inherited from C, whose attached modifier is different w.r.t. the superclass. On the other hand, it is also possible, as in Java, to declare a new field in D which has the same name f (and, possibly, a different security level); in this case, both fields can be accessed from an object of D. This amounts to saying that a security modifier can be *overridden*, but the field itself cannot.

Another feature of Java$_{IF}$ is, as in Java, the possibility to retrieve the *runtime type* of an object by means of an instanceof operation: objectReference instanceof type returns true when objectReference can be cast to type. In the presented framework, such an operation can be seen as something that an attacker can carry out in order to discover the runtime type of objects and break their secrets (Section 2.5).

The language does not include advanced Java features as *concurrency*, *interfaces*, *inner classes*, *exceptions*, and the support for *native* code.

2.2 Information Flow

It is a reasonable requirement for a computing system that a part of the manipulated data must be kept hidden from the observation of external, untrusted agents. If a user wants to keep these data *confidential* during the execution of a program, (s)he can take as a requirement that information cannot *flow* at runtime from private (i.e., which is only visible to trusted users) to public data. An untrusted user, which can only see public data, should not be able, by observing the external behavior of the computing system, to guess anything about what is hidden (private).

Such a policy permits programs to use and manipulate the private part of data, as long as the visible output does not reveal information about it. In *Non-Interference* (NI for short), introduced by Goguen and Meseguer [13], a program $\mathcal{P}$ is *secure* if any two runs only differing in their *private* input (i.e., indistinguishable by an untrusted user) cannot be distinguished by only observing the *public* output. Formally,

$$\forall h_1, h_2, l_1, l_2. \quad l_1 = l_2 \implies [\![\mathcal{P}]\!]^{\mathsf{L}}(h_1, l_1) = [\![\mathcal{P}]\!]^{\mathsf{L}}(h_2, l_2)$$

where $[\![\mathcal{P}]\!]^{\mathsf{L}}(h, l)$ is the public (*low-security*[*4], marked as L) part of the semantics $[\![\mathcal{P}]\!]$ on the input (h, l), divided into a private (*high-security*, marked as H) part h and a public part l. In other words, there must be no *illicit information flow* from h to l, or, equivalently, h and l do not *interfere*. An Information-Flow analyzer should track all the possible flows of information in the program execution, and reject the program if there can exist flows from the private input to the public output.

Example 2.2
In the imperative paradigm, the assignment $l := h^2$, where h is private and l is public, generates a forbidden flow since the (allowed) observation of l gives information about the (hidden) previous value of h. This flow will lead to a rejection of the program, unless its effects are canceled by the following part of the execution.

[*4] See also the first footnote in Section 1.

2.3 Abstract Interpretation

Abstract Interpretation (AI for short) [8] is a theory for systematically deriving approximated program semantics. *Abstraction* is the main notion in defining approximations: the abstraction $\alpha(x)$ of a piece of data x only keeps a subset of the information related to x — for example, an integer number could be abstracted to its sign, thus losing everything about its absolute value. Usually, an *abstract semantics* is an abstraction of the *collecting semantics*, which, for every program point and for every variable, collects all the possible values the variable can hold in any execution. For example, if x can have, depending on the input, (concrete) numerical values from the set $\{2k|k{>}0\}$, then the abstract semantics obtained with the sign abstraction will assign to x the *abstract* value "positive" at that program point. However, it must be noted that the process of obtaining the abstract semantics may lose information, and x could fail to be detected as positive (see Example 2.4).

An *abstract domain* is a mathematical object which describes the result of abstracting *concrete* values. It can be worded either in terms of Galois connections or upper closure operators [9]. The present discussion takes the latter alternative.

Given a set C of concrete values (the integer numbers, in the example above), the *concrete domain* is usually the power-set $\wp(C)$. Then, given the partially ordered set $\langle \wp(C), \subseteq \rangle$ (where $\subseteq$ is just set inclusion), an *upper closure operator* (uco for short) on $\langle \wp(C), \subseteq \rangle$ is a function $\rho : \wp(C) \mapsto \wp(C)$ which is *monotonic*, *idempotent* and *extensive* (i.e., $\forall x.\ \rho(x) \supseteq x$). For example, the uco ρ_s for the sign abstraction is such that $\rho(\{2,3,4\}) = \{n|n{\geq}0\}$ (i.e., all positive numbers plus 0), $\rho(\{-20,-5\}) = \{n|n{<}0\}$ (i.e., all negative numbers), and $\rho(\{-7,3\}) = \mathbb{Z}$ (since concrete values include numbers with different sign).

The set of all ucos on $\wp(C)$ is $\mathrm{UCO}\,(\wp(C))$. An upper closure operator is uniquely determined by the set of its fixpoints, called *abstract values*: these are the elements x of $\wp(C)$ such that $\rho(x) = x$. Such a set is isomorphic[*5] to the abstract domain $\mathcal{A}$ which approximates the concrete domain $\wp(C)$. A set $X \subseteq \wp(C)$ (i.e., a set of sets of concrete values) is the set of fixpoints of a uco iff it is a *Moore-family*, i.e., if X is equal to its Moore closure $\mathcal{M}(X) = \{\cap S \mid S \subseteq X\}$ (where $\cap$ is the intersection on all elements of S). In the following, $\rho(v)$ will stand for $\rho(\{v\})$ whenever $\{v\} \in \wp(C)$ is a singleton.

A concrete computation $f_{\wp(C)}$ on $\wp(C)$ can be abstracted to an abstract one $f_{\mathcal{A}}$ on $\mathcal{A}$ by providing the abstraction of values (as exemplified before with the sign ab-

[*5] The use of ucos instead of Galois connections allows one to get rid of the abstract domain by using a subset of the concrete domain, isomorphic to it.

straction) and operators. The abstraction of such a computation is *sound* if the abstract result is always a correct approximation of the concrete result: $\forall x.\ f_{\wp(C)}(x) \subseteq f_{\mathcal{A}}(x)$, where $\subseteq$ means *more concrete* or *more precise*, i.e., identifying a smaller set of concrete values.

Example 2.3
Let $f_{\wp(C)}(x,y) = x{+}y$, and let $\mathcal{A}$ represent the sign abstraction. Then, $f_{\mathcal{A}}(x,y)$ will take the abstraction of x and y, and apply an abstract version $\oplus$ of $+$ to get the result. Let $positive$ be $\{n|n{\geq}0\}$, and $negative$ be $\{n|n{<}0\}$. The abstract operator is defined as $positive{\oplus}positive = positive$, $negative{\oplus}negative = negative$, $positive{\oplus}negative = negative{\oplus}positive = \mathbb{Z}$. In this case, $f_{\mathcal{A}}$ is a sound abstraction of $f_{\mathcal{C}}$, since the abstract result will always include the concrete result. For example, let $x = 5$ and $y = 7$; then $f_{\wp(C)}(x,y) = 12$, and $f_{\mathcal{A}}(x,y) = \rho(\{x\}) \oplus \rho(\{y\}) = positive \oplus positive = positive$, which includes 12 since $\rho(\{12\}) = positive$.

Abstraction formalizes the idea that $\mathcal{A}$ is simpler than $\wp(C)$, being (isomorphic to) a subset of its. On the other hand, a computation $f_{\mathcal{A}}$ on $\mathcal{A}$ can be less precise than its corresponding concrete computation $f_{\mathcal{C}}$, since values $V \in \mathcal{C} \setminus \mathcal{A}$ cannot be used.

Example 2.4
Consider the abstraction of Example 2.3: if $x = 5$ and $y = 7$, then $f_{\wp(C)}(x,y) = 12$ while $f_{\mathcal{A}}(x,y) = positive$, which is clearly less precise. Moreover, if $x = 5$ and $y = -8$, then $f_{\wp(C)}(x,y) = -3$ and $f_{\mathcal{A}}(x,y) = \mathbb{Z}$, which is even less precise than the abstraction $\rho(-3)negative$. This circumstance is called *incompleteness*[12].

If $\langle \mathcal{C}, \top, \bot, \vee, \wedge \rangle$ is a complete lattice, then $\langle \mathrm{UCO}(\mathcal{C}), \mathrm{TOP}, \mathrm{ID}, \vee', \wedge' \rangle$, ordered point-wise, is also a complete lattice where $\mathrm{ID} = \lambda x.x$ describes the identity abstraction (i.e., the most concrete domain $\mathcal{A} = \mathcal{C}$, which does not lose any information), and $\mathrm{TOP} = \lambda x.\top$ is the trivial abstraction mapping $\mathcal{C}$ into a singleton $\mathcal{A} = \{\top\}$.

The *reduced product* $\sqcap$ [9] of a set of domains $\{\mathcal{A}_i\}$ is the most abstract domain which is at least as concrete as any A_i: formally, $\sqcap_i \mathcal{A}_i$ is the Moore closure $\mathcal{M}(\cup_i \mathcal{A}_i)$. The intuition is that the reduced product collects all and only the information contained in any of the $\mathcal{A}_i$.

Example 2.5
Let the *parity abstraction* ρ_p obey, as expected, $\rho_p(\{-2,6,8\}) = \{2n|n{\in}\mathbb{Z}\}$ (the even numbers), $\rho_p(\{-1,3\}) = \{2n+1|n{\in}\mathbb{Z}\}$ (the odd numbers), and $\rho_p(\{1,2\}) = \mathbb{Z}$ (any number). Combining the sign abstraction with the parity abstraction by means of the

reduced product gives an abstract domain which can distinguish between positive even numbers, positive odd numbers, negative even numbers, and negative odd numbers.

Notation will be often abused by referring to ρ as the set of its fixpoints, i.e., $x \in \rho$ if x belongs to the domain $\mathcal{A}$ generated by ρ. In the present work, the role of Abstract Interpretation is twofold: (i) it provides the basis for defining the security property; and (ii) it gives the background for developing the static analyzer.

2.4 Abstract Non-Interference

Non-Interference can be weakened by modeling secrecy relatively to some observable property, or *abstraction* of data[10]. The observational power of an attacker is limited to an abstraction, and a *secure* program is one which preserves confidentiality (i.e., such that no illicit flows may arise) only w.r.t. the information the attacker can observe. Let the concrete domain be the set $\wp(\mathbb{Z})$ of all properties on integers, where a property is identified by the set $P \subseteq \mathbb{Z}$ of values satisfying it. Upper closure operators describe the ability of an attacker in observing data: if the attacker has precision ρ, then (s)he cannot distinguish v_1 and v_2 if $\rho(v_1) = \rho(v_2)$ (i.e., if the values have the same property w.r.t. ρ).

$\mathcal{P}$ is secure for two abstract domains η and ρ, written $[\eta]\mathcal{P}(\rho)$, if no flows are detected by observing public input and output data only up to a precision characterized by, respectively, η and ρ:

$$\forall h_i, l_i. \quad \eta(l_1) = \eta(l_2) \implies \rho([\![\mathcal{P}]\!]^{\mathsf{L}}(h_1, l_1)) = \rho([\![\mathcal{P}]\!]^{\mathsf{L}}(h_2, l_2))$$

where l_i and h_i are, respectively, values assigned to some public and private program variables. This means that, if l_1 and l_2 cannot be distinguished by η, then it is not possible to guess h data from the (abstracted by ρ) output. Standard NI is a special case of ANI, corresponding to $[\mathrm{ID}]\mathcal{P}(\mathrm{ID})$ (neither the input nor the output are approximated).

Unfortunately, flows may be detected, which are caused by a change in the public instead of the private input. These flows are called *deceptive* since they are not really dangerous. A more general version of ANI, namely (η, ϕ, ρ)-secrecy[*6], rules out deceptive flows by computing $[\![\mathcal{P}]\!]$ on an abstraction of the input:

$$\forall h_i, l_i. \quad \eta(l_1) = \eta(l_2) \implies \rho([\![\mathcal{P}]\!]^{\mathsf{L}}(\phi(h_1), \eta(l_1))) = \rho([\![\mathcal{P}]\!]^{\mathsf{L}}(\phi(h_2), \eta(l_2)))$$

Example 2.6

Consider Example 2.2: the assignment $l := h^2$ is no longer dangerous if the attacker can only observe the sign of integers, since knowing the sign of l does not give any

[*6] This version is called *abstract*, opposed to the *narrow* version introduced in the previous definition.

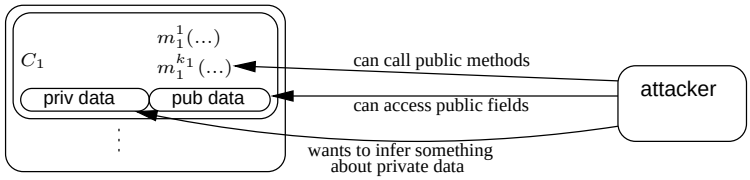


Fig. 2 The program and the attacker

information on the input sign of h. Therefore, there is no *abstract flow* from h to l, and the program can be accepted.

In the following, we deal with (ρ, ρ, ρ)-secrecy: the class hierarchy data may belong to will identify the abstract domain to be considered. It is important to point out that, in this case, the semantics $[\![\mathcal{P}]\!]$ is an abstract one, which only keeps track of type information, as in (η, ϕ, ρ)-secrecy.

2.5 A model of Object-Oriented Information Flow

Information Flow properties have been mostly defined on simple languages. Yet, advanced features in present-day Object-Oriented languages make a correct definition much harder to achieve in such an arduous framework. For example, it is no longer clear what a high-security or low-security variable is, since variables can be dynamically declared. Particularly, in order to reasonably adapt the Non-Interference notion

> *what an attacker can see of the output does not allow him or her to acquire (abstract) information about the secret input*

to Java$_{IF}$, it is necessary to clarify (i) what a running program is; (ii) what attackers are and can do; (iii) which data have to be protected (Figure 2).

[1] Attacked programs

The Java$_{IF}$ program whose secrecy we want to investigate is considered to be a collection of cooperating classes, which can be directly accessed according to Java access-control rules. The focus is on detecting whether the legal interaction with a class C may disclose its confidential data. An interaction can take the form of an invocation of a public method m of C, or the access to some public field f. A class is said to be *secure* if all possible interactions are harmless, i.e., if it is not possible to acquire knowledge of its high-security data by interacting with its public (in the sense of the Java access-control policy) part and reading low-security information.

[2] Attackers

In this framework, attackers are programs which can interact with external classes C and aim at breaking their secrets by calling their methods or accessing their fields. In other words, the system *attacker+attacked program* is a complete program, in the classic terminology, where the main method lies in the attacker, and the attacked program is rather to be seen as a library.

An attacker can only read C data (by field accessing) before and after calling any method m. If a dangerous flow arises in m, then accessing data before and after its execution should provide the attacker with confidential information. On the other hand, it is not possible to observe extra information such as the amount of allocated memory or the execution time, which would entail additional flows.

Example 2.7

Let $\mathcal{P}$ allow dynamic variable declaration, with h and h1 high-security.

```
if (p(h)) {
  h1 = x;
} else {
  C i = new C();
  h1 = y;
}
```

This fragment is secure if the observer can only see data before and after the statement. In fact, i is no longer visible outside the *else* branch, so that it is not possible to guess which path has been taken, and the value of h. However, two executions with different values for h could be told apart if, for example, the attacker could see the intermediate value of variables, or the amount of public allocated memory — in fact, executing the else branch results in allocating additional memory.

[3] The security level of data

Java comes with field modifiers public, private and protected, which model security requirements by regulating the direct access to data. That is, a program is rejected at compile time if, for example, a private field is accessed from an object of a class which is not the one where it was declared. However, private information can be indirectly *propagated* via the return value of public methods. This is, indeed, one of the keys of the idea of *data encapsulation*, where getYYYY methods are usually designed to bring out internal (but not meant to be hidden completely) data.

Non-Interference analysis enforces the property that secret information should not *propagate* through the computation. Because of encapsulation, secret information

cannot be identified with private data; therefore, a different strategy is needed to identify high-security information. New, ad-hoc modifiers are added to the language as code annotations: flowH and flowL specify, at the level of classes, that a field is to be regarded, respectively, as high- or low-security (if no such information is provided, then the field is considered as low-security). Our security policy requires that there are no (abstract) flows from flowH fields to flowL fields or return values. Thus, flowH and flowL are somehow orthogonal to private and public, and program analysis must enforce, say, that an attacker cannot access a public flowH field[*7].

If f is declared as flowH in D, then all the data stored in the field f of D objects are considered as secret. Due to this choice, it is easy to represent high and low data: each class has a set of high-security fields and a set of low-security fields, and both can be easily and statically decided from inspecting the program syntax. Given D where f is flowH, if one wants some instance o of D *not* to protect its field f, then a subclass D' of D, where f is declared as flowL, must be added, and o must be an instance of D'. This can be the case when the security policy requires to consider a subset of D instances in a special way as regards security, in spite of being structurally identical to any other D instances (note that, in OO languages, structural identity is usually what makes two objects belong to the same class, so that, in the presented setting, classes may acquire an additional meaning).

Example 2.8
In the class D above, {D.l} is the set of low data, while {D.h} is the set of high data. (The content of) Any field o.h of an object o of class D is considered as high-security.

Importantly, this representation loses information about *which* objects of a given class flow into the observable part of the program. However, it automatically (and soundly) deals with *sharing* [21], since every object is somehow considered to implicitly share with other objects in the same class.

§3 Class-Oriented Abstract Non-Interference

Abstract Non-Interference is parametric on *how* data can be observed by untrusted users, i.e., the degree of precision attackers have when reading low-security information. Given a universe of values, properties are sets of values which share some common behavior. In an Object-Oriented framework, *classes* have a similar purpose: they identify collections of *objects* sharing the same internal structure. Therefore, mod-

[*7] As a matter of fact, this combination of access-control modifiers and security modifiers is legal but somehow weird.

eling properties by means of classes is quite natural, the class being exactly the property of interest for a given object. In the present setting, program classes are the kind of properties an attacker can observe.

Identifying properties with classes reduces property checking to a sort of class-directed program analysis [17]. Classes are ordered by the subclass relation: a class is greater than any of its subclasses. A subclass models a sub-property, i.e., a more precise property since it is "satisfied" (by instantiation) by a smaller set of values.

As for notation, depending on the context, names C, D etc. will denote either sets of semantic values, classes, or properties, while o will stand for objects or values. The subclass relation $C' \sqsubset C$ is basically $C' \subseteq C$[*8] if C' and C are considered as properties or sets of values. The predicate $o{:}C$ (or, sometimes, $o \in C$) holds if o has class C (i.e., if o can be stored in a variable whose declared type is C). On the other hand, $o{::}C$ holds iff o is an instance of C (typically, an object created by a $\mathsf{new}\ \mathsf{C}()$ instruction): this amounts to saying that $o{:}C$ and there is no $D \sqsubset C$ s.t. $o{:}D$. Finally, $C_{\downarrow}$ is defined as the set of subclasses $\{D | D \sqsubseteq C\}$ of C.

3.1 Class hierarchies as abstract domains

Consider a class C (which, as discussed above, can also be regarded as a set of objects), and let the concrete domain be $\wp(C)$. A *class hierarchy* rooted at C (i.e., where the class C is a superclass of all the other classes in the hierarchy [*9]) identifies a subset of properties $D \in \wp(C)$, i.e., an abstract domain ρ.

Lemma 3.1
Let a class hierarchy rooted at C be given, and let the concrete domain be $\wp(\mathcal{O})$ where $\mathcal{O}$ is a set of instances of the class C or its subclasses. Let the function $\rho : \wp(\mathcal{O}) \mapsto \wp(\mathcal{O})$ be defined as follows: $\rho(O)$ is the smallest set $O' \supseteq O$ of objects such that O' is the set of all objects having class D for some class D in the hierarchy. Then, ρ is an upper closure operator on $\wp(\mathcal{O})$.

Proof The function ρ is clearly monotone: the larger the object set, the greater the minimal class such that every object has this class. It is also idempotent since the result of applying ρ once is a set O' of values (all the values, actually) which have some specific class D. Then, applying ρ once again gives the same O' because, trivially. $O' \supseteq O'$, and it already corresponds to the class D. Finally, ρ is extensive because of

[*8] Inclusion is not strict since C instances which are not C' instances are not guaranteed to exist.

[*9] Usually, we may want to ignore the Object superclass, thus obtaining a set of hierarchies instead of a single hierarchy rooted at Object.

the condition $O' \supseteq O$. ■

Example 3.1
Let C be the set of integers. In Java or Java$_{IF}$, it can be modeled by the class Integer. Let the semantic properties of sign be modeled by ρ_s: this domain can be represented by the class hierarchy $\{\mathsf{Integer},\ \mathsf{Pos}\sqsubset\mathsf{Integer},\ \mathsf{Neg}\sqsubset\mathsf{Integer}\}$, where, for example, objects of class Pos should contain a positive number. Subclasses should be equipped with methods which encode their behavior with respect to the abstract property. For example, if o:Pos, the method invocation $o.\mathsf{sum}\,(\mathsf{new}\ \mathsf{Pos}(n))$, which sums o and n, should be declared to have return type Pos, while $o'.\mathsf{sum}\,(\mathsf{new}\ \mathsf{Neg}(n))$ should be declared to return an Integer with no sign indication if o':Integer (since the sign of the return value is not statically decidable). Clearly, the definition of these subclasses should avoid that, say, a positive number is written in the value field of a Neg object, which would break the required policy.

Given a set of values $D \in \wp(C)$, $\rho(D)$ is the smallest class $D' \in \rho$ such that $D \subseteq D'$. Since no idea of *multiple inheritance* is allowed (note that Java$_{IF}$ does not support interfaces), generally there is no subclass C of both C_1 and C_2 which models $C_1 \cap C_2$ (unless $C_1 \sqsubset C_2$ or $C_2 \sqsubset C_1$). Therefore, to have ρ closed under intersection, as required by the definition of upper closure operators, the Bot_ρ empty class is supposed to exist for every class hierarchy ρ. As a consequence, abstract domains which are representable in this model are those where classes are either related by $\sqsubset$, or disjoint[*10].

The declaration of a class as abstract (i.e., which cannot be directly instantiated) can give additional information to the class-based analysis. For example, given an abstract class C with $C_\downarrow = \{C, C_1, C_2\}$, the information o:C boils down to o::$C_1 \vee o$::C_2 since o is guaranteed not to be an instance of C. Although nothing changed from the set-theoretic point of view, since $C = C_1 \cup C_2$, this additional information can help in the inference of non-dependency results, as shown in Section 5.

3.2 Primitive types

Since values are identified with objects, the reader may ask whether primitive types can be accounted for in this framework. Actually, focusing only on classes means to *abstract away* the value of data with primitive types.

Example 3.2
Let the class C have fields f of type D and g of type int, and o be declared as C. In this

[*10] Languages like C++, which allow multiple inheritance, permit representing a wider range of domains.

case, asking the runtime type of o means asking if the object belongs to some subclass of C. Also, asking the type of $o.f$ gives information about which subclass of D the stored value belongs to. However, there is no way to know, with such questions, the value of $o.g$ nor any internal values of primitive type in $o.f$.

Clearly, it is possible to describe properties on primitive data by encapsulating them into classes. For example, int can be embedded into Integer, and properties of numbers can be represented by subclasses, as shown in Example 3.1.

3.3 The link to the Abstract Non-Interference theory

This section describes how ANI is related to the present work, and provides basic ideas in order to understand how the analyzer works. The program $\mathcal{P}$ whose secrets are to be broken takes the form, at runtime, of a collection $\mathcal{O}$ of objects and $\mathcal{C}$ of classes which are available in the execution environment. $\mathcal{E}$ denotes the set of *execution states*, containing the *activation stack* and the *heap* according to the concrete semantics. The *visible part* (form the point of view of an attacker) of $\varepsilon \in \mathcal{E}$ consists of low-security instance fields of objects in $\mathcal{O}$, and low-security static fields of classes in $\mathcal{C}$.

Let $\varepsilon_1, \varepsilon_2 \in \mathcal{E}$; the *indistinguishability condition* $\text{IC}\ (\varepsilon_1, \varepsilon_2)$ holds if

- for every class C, objects $o_1, o_2{:}C$ (note the use of : instead of ::) and low field f, values $o_1.f$ in ε_1 and $o_2.f$ in ε_2 belong to the same class ($\varepsilon(o.f)$ is the value of the field f of the object o in the state ε):

$$\exists D.\ \varepsilon_1(o_1.f){::}D \wedge \varepsilon_2(o_2.f){::}D$$

- for every class D and low static field g, $D.g$ is an instance of the same class in both states.

Given two sequences of values $\overline{v}_1$ and $\overline{v}_2$, the indistinguishability condition $\text{IC}\ (\overline{v}_1, \overline{v}_2)$ is taken to be class equality for every two corresponding elements.

The meaning of the condition is that two states cannot be distinguished by someone who can only check the class of low-security data. An attacker may try to break the secrets of $\mathcal{P}$ by invoking $o.m$ of $o \in \mathcal{O}$ (or a static $C.m$ of $C \in \mathcal{C}$), and observe the final state of the execution. The *Abstract Non-Interference condition* $\text{ANI}\ (m)$ holds for m if

$$\text{IC}\ (\varepsilon_1, \varepsilon_2) \wedge \text{IC}\ (\overline{v}_1, \overline{v}_2) \quad \Rightarrow \quad \text{IC}\ (\llbracket o.m\ (\overline{v}_1) \rrbracket\ (\varepsilon_1), \llbracket o.m\ (\overline{v}_2) \rrbracket\ (\varepsilon_2))$$

where $\llbracket o.m\ (\overline{v}) \rrbracket\ (\varepsilon)$ is the state obtained by executing $o.m$ with parameters $\overline{v}$ in ε (return value included). This means that, after executing m in two states only differing (at the abstract level) in their high-security part, the low-security part of the output states

cannot be distinguished. The attacked program $\mathcal{P}$ is secure (written $\text{ANI}(\mathcal{P})$) if this condition holds for every possible interaction between the program and the attacker.

This definition is a special case of (an Object-Oriented adaptation of) the Abstract Non-Interference original setting. Next section will describe how the analyzer tries to prove $\text{ANI}(m)$.

§4 Analysis of Non-Interference for class information

This section describes the analysis (called *class-flow analysis*) in its main features. It is basically a formalization based on Abstract Interpretation, implemented as a prototype in the `Ciao` [4] Prolog system. The algorithm performs an *abstract execution* of a method which could be called from outside the program under study (possibly by an untrusted user), and tries to detect illicit abstract flows from high-security fields to low-security fields or the returned value. As pointed out before, a program is secure if no method execution provokes illicit flows.

The abstract semantics relies on the notion of *abstract states* $\sigma \in \mathcal{S}$. An abstract state σ consists of two parts: (1) a *global environment* maps each static or instance field f of the class C to an abstract value; and (2) a frame-stack stores the abstract value of local variables in the usual way (i.e., maps a local variable in the frame on top of the stack to an abstract value).

An *abstract value* V takes the form $\mathcal{X}_\phi \in \mathcal{V}$, where $\mathcal{X}$ is a set of classes, and $\phi \in \{\mathsf{H}, \mathsf{L}\}$ is a *security flag* indicating the security level (high or low) of the corresponding piece of data. Before starting the execution, fields which are considered as flow$\mathsf{H}$ by the policy are given the value $(C_\downarrow)_\mathsf{H}$, where C is the declared class (remember that $C_\downarrow$ is C plus all its subclasses), while flow$\mathsf{L}$ fields are given the value $(C_\downarrow)_\mathsf{L}$. As regards notation, $\sigma(C.f)$ is the abstract value of the field f of the class C in σ, while $\sigma(x)$ refers to the abstract value of the variable x in the active frame of σ. Saying that $\sigma(x)$ (resp., $\sigma(C.f)$) is $(D_\downarrow)_\mathsf{L}$ means that x (resp., the content of the field f of some instance of C) can be instance of any class in $D_\downarrow$, and does not contain any information which has been propagated from high-security data. On the other hand, saying that $\sigma(x)$ (resp., $\sigma(C.f)$) is $(D_\downarrow)_\mathsf{H}$ means, again, that the class is guaranteed to be one among $D_\downarrow$, but does not guarantee anything about the security level (the individual piece of data could also be low, but H is taken as a sound approximation).

Ordering $\leq$ and *least upper bound* $\sqcup$ on abstract values are defined as

$$\begin{array}{rcl} {\mathcal{X}'}_{\phi'} \leq {\mathcal{X}''}_{\phi''} & \equiv & \mathcal{X}' \subseteq \mathcal{X}'' \wedge \phi' \leq \phi'' \\ \phi' \leq \phi'' & \equiv & \phi'{=}\mathsf{L} \ \vee \ \phi''{=}\mathsf{H} \end{array}$$

$$\begin{array}{rcl} {\mathcal{X}'}_{\phi'} \sqcup {\mathcal{X}''}_{\phi''} & = & (\mathcal{X}' \cup \mathcal{X}'')_{\phi' \sqcup \phi''} \\ \phi' \sqcup \phi'' & = & \begin{cases} \mathsf{L} & \text{if } \phi' = \phi'' = \mathsf{L} \\ \mathsf{H} & \text{otherwise} \end{cases} \end{array}$$

In standard type theory, types are a partial order. A type can be seen as an abstraction of a set of values [7]. When x is given type τ, and τ has several subtypes $\tau_1, .., \tau_k$, it is not possible, from the type representation, to guess which τ_i the variable x belongs to. On the other hand, our choice of representing abstract values as sets of classes can be more precise, and also allows us to exploit `abstract` declarations.

Example 4.1
Let D be declared as `abstract`, and the (instantiable) classes D_1 and D_2 be its subclasses. Suppose the abstract value $\{D_1, D_2\}_\phi$ be computed for an object o. This means that the analysis cannot infer which subclass o belongs to. Nevertheless, this is more precise than the typing D obtained with a type-based representation, but still correct since D has no direct instances. In other words, both approaches identify the same set of values. In fact, a typing D for o implies that, according to the information provided by the type representation, the object can belong to any type among D, D_1 and D_2. This type models the abstract value $(D_\downarrow)_\phi = \{D, D_1, D_2\}_\phi$, which is less precise than $\{D_1, D_2\}_\phi$ since it also includes D. See Section 5 for an application.

Remark 4.1
We note that, due to how high- and low-security data are defined (Section 2.5), there is no need to keep a representation of the concrete heap (i.e., an *abstract heap*) in the abstract states. In fact, consider two objects o_1 and o_2, instances of the class C, such that the objects stored in the field f of both are instances of the same class, but $o_1.f$ is high while $o_2.f$ is low. The security policy does not distinguish between o_1 and o_2: to its purposes, all instances of C have a field f which has to be (conservatively) considered as high, since there is at least one object whose field f is actually high — or cannot guaranteed to be low. In other words, the security level is a property which refers to *the field f of the class C*, rather than *the field f of an instance o of the class C*, and the abstract state does not need an abstract heap in order to represent this information accordingly.

$$
\begin{aligned}
[\![x]\!]^\sharp_\phi(\sigma) &= \langle \sigma(x)^\phi,\ \sigma \rangle \\
[\![o.f]\!]^\sharp_\phi(\sigma) &= \text{LS}\left(\left\langle \sqcup \{\sigma'(C.f) \mid C \in \mathcal{X}^o\}^\phi,\ \sigma' \right\rangle\right) \\
&\text{where} \quad \langle {\mathcal{X}^o}_{\phi_o},\ \sigma' \rangle = [\![o]\!]^\sharp_\phi(\sigma) \\
[\![C.f]\!]^\sharp_\phi(\sigma) &= \text{LS}\left(\langle \sigma(C.f)^\phi,\ \sigma \rangle\right) \\
[\![\mathsf{new}\ C(p_1,..,p_k)]\!]^\sharp_\phi(\sigma) &= (\text{ANALYZE}_{\sigma_k}(C(V_1,..,V_k)))^\phi \\
&\text{where} \quad \sigma_0 = \sigma \\
&\qquad \langle V_i,\ \sigma_i \rangle = [\![p_i]\!]^\sharp_\phi(\sigma_{i-1}) \\
[\![o.m(p_1,..,p_k)]\!]^\sharp_\phi(\sigma) &= \text{LS}\left((\text{ANALYZE}_{\sigma_k}(V_o.m(V_1,..,V_k)))^\phi\right) \\
&\text{where} \quad \langle V_o,\ \sigma_0 \rangle = [\![o]\!]^\sharp_\phi(\sigma) \quad \text{and} \quad \langle V_i,\ \sigma_i \rangle = [\![p_i]\!]^\sharp_\phi(\sigma_{i-1}) \\
[\![C.m(p_1,..,p_k)]\!]^\sharp_\phi(\sigma) &= \text{LS}\left((\text{ANALYZE}_{\sigma_k}((\{C\}_\mathsf{L}).m(V_1,..,V_k)))^\phi\right) \\
&\text{where} \quad \sigma_0 = \sigma \quad \text{and} \quad \langle V_i,\ \sigma_i \rangle = [\![p_i]\!]^\sharp_\phi(\sigma_{i-1})
\end{aligned}
$$

Fig. 3 Rules for analyzing expressions

In Java and most imperative or Object-Oriented languages, illicit flows come from (i) assigning a high-security value to a low variable or field (*explicit flows*); or (ii) executing conditional statements or loops whose guard depends on high data (*implicit flows*). In detecting explicit flows, the main issue comes to be the computation of the security level of expressions. An illicit flow is soundly assumed to exist if a high-security value is computed for an expression which will be assigned to a low-security field or variable. Implicit flows are dealt with by means of a *global analysis* which remembers the security level under which a command is executed: if an assignment to low data depends on a high-security guard, then its effects may lead to an illicit flow even if the assigned value is low-security.

Next sections illustrate the formalization in its main parts.

4.1 The computation of Abstract Values

In the assignment $x = e$, the structure of the expression e might be quite intricate: e.g., it can be a method call with side effects. Therefore, computing the security level ϕ_e of an expression is not trivial at all. Figure 3 shows how abstract values can be computed. The *expression semantics* $[\![]\!]^\sharp$ takes e, $\sigma \in \mathcal{S}$, and ϕ, and returns a pair $(value, resulting\ state)$. The goal of this abstract semantics is to find out whether an expression contains information which must be regarded as confidential.

The operator $[]^\phi$ *raises* the security level of a value: $(\mathcal{X}_{\phi'})^{\phi''} = \mathcal{X}_{\phi' \sqcup \phi''}$. The flag ϕ in $[\![e]\!]^\sharp_\phi(\sigma)$ means that e is computed under the security level ϕ. The level under which a computation is performed depends on whether executing the piece of code is

conditional on a high guard. ϕ comes to be the highest level of any guard the code depends on (i.e., the guard of any conditional or iterative construct which surrounds the program point under study). Every updating of low data under H has to be seen as an illicit implicit flow (see above). Note that the least upper bound which is performed in evaluating instance fields (second equation in the figure) is not necessary in the case of static fields, since the class C in $C.f$ is statically decided. The function $(\text{ANALYZE}_\sigma\,(V.m(\ldots)))^\phi$ is defined in the next section.

The function LS (where the name stands for *lower singleton*) is only applied to the first element of the pair (i.e., the abstract value); it lowers the security flag of a value when the class set is a singleton: $\text{LS}\,(\langle\mathcal{X}_\phi,\,\sigma\rangle)$ is $\langle\mathcal{X}_\mathsf{L},\,\sigma\rangle$ if $\mathcal{X}$ is a singleton, and $\langle\mathcal{X}_\phi,\,\sigma\rangle$ otherwise. This operation is important since it shows how abstract data dependencies may differ w.r.t. their concrete counterpart.

Example 4.2
Let $\{C\}_\phi$ be the result of computing $[\![e]\!]^\sharp_{\phi'}\,(\sigma)$, and o be the object computed by runtime (concrete) execution. Thus, o::C certainly holds by soundness of the abstract semantics. Since type information is all an attacker can see, the security content of o can be considered as L even if e contains high data. In fact, illicit flows do not occur since the class of o is constant, so that it is indistinguishable by the attacker. This is consistent with the definition of Abstract Non-Interference for Object-Oriented programs, since the uniqueness of the class implies that the right-hand side of the indistinguishability condition (Section 3.3) is trivially satisfied. Typically, $\mathcal{X}$ is a singleton $\{C\}$ if C has no subclasses; however, it can be the case, as in new $C()$, that the class is unique although C does have subclasses.

As an example, consider the $[\![o.f]\!]^\sharp_\phi\,(\sigma)$ rule in Figure 3. In this case, the object o is first evaluated to get a set of classes $\mathcal{X}^o$ (the classes o might belong to at runtime) and a security flag ϕ_o. For every class in $\mathcal{X}^o$, the abstract state σ contains information about the security of the field f. Since it is not known statically which class will be needed, a least upper bound of all instances is computed. The flag of the result is possibly raised if the computation is performed under a high guard. Finally, LS lowers the flag to L if every $C.f$ can be statically said to be instance of the same, unique class.

4.2 The analysis of Statements

The analysis of a method body needs to execute abstract statements and compute abstract expressions. In the following, $\sigma\,[x \leftarrow V]$ is the *updated* state σ' such that $\sigma'(x) = V$ and $\sigma'(y) = \sigma(y)$ for every $y \neq x$. Similarly, $\sigma\,[C.f \leftarrow V]$) is the up-

$$
\begin{aligned}
\textsc{exec}_\sigma\,(C\ x) &= \sigma\left[x \leftarrow (C_\downarrow)_{\mathsf{L}}\right] \\
\textsc{exec}_\sigma\,(x = e)^\phi &= \sigma'\left[x \leftarrow \mathcal{X}_{\phi'}\right] \\
&\text{where} \quad \langle \mathcal{X}_{\phi'},\ \sigma' \rangle = [\![e]\!]^\sharp_\phi\,(\sigma) \\
\textsc{exec}_\sigma\,(o.f = e)^\phi &= \sigma''\left[\mathcal{X}.f \twoheadleftarrow \mathcal{X}'_{\phi'}\right] \\
&\text{where} \quad \langle \mathcal{X}_\phi,\ \sigma' \rangle = [\![o]\!]^\sharp_\phi\,(\sigma) \text{ and } \langle \mathcal{X}'_{\phi'},\ \sigma'' \rangle = [\![e]\!]^\sharp_\phi\,(\sigma') \\
\textsc{exec}_\sigma\,(S_1;\ S_2)^\phi &= \textsc{exec}_{\sigma'}\,(S_2)^\phi \\
&\text{where} \quad \sigma' = \textsc{exec}_\sigma\,(S_1)^\phi \\
\textsc{exec}_\sigma\,(\mathbf{if}(b)\ s_1\ \mathbf{else}\ s_2)^\phi &= \textsc{exec}_{\sigma_1}\,(s_1)^{\phi \sqcup \phi_b} \sqcup \textsc{exec}_{\sigma_2}\,(s_2)^{\phi \sqcup \phi_b} \\
&\text{where} \quad (\phi_b, \sigma_1, \sigma_2) = [\![b]\!]^\sharp\,(\sigma)
\end{aligned}
$$

Fig. 4 Some rules for analyzing statements

dated state obtained by storing V in $C.f$. The set extension $\sigma\,[\mathcal{X}.f \leftarrow V]$ is also easily defined. Moreover, the *upgraded* state $\sigma' = \sigma\,[v \twoheadleftarrow V]$ satisfies $\sigma'(v) = \sigma(v) \sqcup V$ and $\sigma'(w) = \sigma(w)$ for any $w \neq v$. Figure 4 shows how statements are executed at the abstract level, instead of being replaced. State upgrading (instead of simple updating) is used since it is not known which instances $C.f$ will be actually assigned to at runtime. Therefore, the initial value of any field must be kept for the sake of soundness, and combined with the assigned value.

Example 4.3
Let o be given statically a value $\{D, D_1\}$ with $D_1 \sqsubset D$. Let $o.f$ be H before the assignment $o.f = e$, and e be L. *Updating* the state would result in a final L flag for both $D.f$ and $D_1.f$. This is unsound if the runtime class of o is D_1, since $D.f$ would be harmfully considered as L. Yet, *upgrading* is sound since the H flag is kept for $D.f$ as a result of the least upper bound.

For a guard b, the function $[\![b]\!]^\sharp\,(\sigma)$ computes its flag ϕ in σ, giving also two abstract states corresponding to the two branches. The state σ_1 includes the information which can be extracted from assuming, as in the "then" branch, that b was evaluated to true, while σ_2 is the dual for false. States σ_1 and σ_2 are different only when something can inferred from the truth value of the guard. For example, if b is x $\mathsf{instanceof}$ C, assuming $\neg b$ means that its class is not a subclass of C. The $\mathsf{instanceof}$ guards are the only case where this additional information can be obtained. Otherwise, class information is not enough for correctly describing the conditional. Therefore, σ_1 and σ_2 are both (conservatively) assumed to be the least upper bound of the abstract states computed for each branch. In some cases, this improves the precision of the analysis;

however, it can be simply ignored if a simpler analysis is needed.

Example 4.4
Let b be $(\mathsf{h} == \mathsf{null})$, and the class set inferred at this program point for the high identifier h be $\{D\}$. If the definition of $[\![]\!]^\sharp$ is followed, then the flag of h should be L, thus computing L as the final flag of b. However, it is easy to see that this treatment is unsound. In fact, the (concrete) boolean value of the guard is far from being totally determined by the class of v, since $\{\mathsf{null}\}$ is a strict subset of the set of possible runtime values for v. Consequently, knowing that h has class D does not give information about the branch which is taken, and no additional hypothesis can be used when examining each of the branches.

The superscript in the expression $\textsc{exec}_\sigma\,(s)^\phi$ means that the effects of s on σ will be raised by ϕ (similarly to the definition of $[\![]\!]^\sharp$): when V is computed in σ, its flag ϕ_V is raised to $\phi_V \sqcup \phi$. This is important in dealing with implicit flows, originating from a high-security guard, where $\phi = \mathsf{H}$.

4.3 The analysis of Methods

The function $(\textsc{analyze}_\sigma\,(V.m(\dots)))^\phi$, already used in Figure 3, deals with the security analysis of *methods*. A method can be either invoked as a statement — when there is no return value, or it is ignored — or as part of an expression. Let V and $V_1..V_k$ be abstract values obtained for the calling object and the actual parameters by previous abstract computations. The result of $(\textsc{analyze}_\sigma\,(V.m(V_1..V_k)))^{\phi'}$ is obtained by the least upper bound of the abstract execution in σ of each instance $C.m$ such that $V = \mathcal{X}_\phi$ and $C \in \mathcal{X}$. More formally:

$$(\textsc{analyze}_\sigma\,(\mathcal{X}_\phi.m(V_1,..,V_k)))^{\phi'} = \sqcup_{C\in\mathcal{X}}\left((\textsc{analyze}_\sigma\,(C_\phi.m(V_1,..,V_k)))^{\phi'}\right)$$

where $\sqcup$ works on both the return value and (by point-wise extension) the final state:

$$\begin{array}{rcl}\langle \mathcal{X}_{1\phi_1}, \sigma_1\rangle \sqcup \langle \mathcal{X}_{2\phi_2}, \sigma_2\rangle & = & \langle \mathcal{X}_{1\phi_1} \sqcup \mathcal{X}_{2\phi_2} \quad , \quad \sigma_1 \sqcup \sigma_2\rangle \\ (\sigma_1 \sqcup \sigma_2)(id) & = & \sigma_1(id) \sqcup \sigma_2(id)\end{array}$$

Importantly, when static methods and constructors are concerned, only one instance $C.m$ needs to be considered since the class C is fixed.

The notation C_ϕ means that the flag ϕ of the caller is kept in the analysis of each C method instance (it is stored in the local variable this). This is another difference with respect to standard Information-Flow analysis, and it is also applied to field access. Usually, in a compositional, syntax-based type-system approach, the

analysis of $o.f$ or $o.m(\dots)$ leads to an H flag whenever o is H. On the contrary, in the present formulation, the H content of o is not a sufficient condition for considering $o.f$ or $o.m(\dots)$ as high-security. For example, let $o{:}C$ be high-security, and $C_{\downarrow} = \{C, C_1, C_2\}$. In all classes, f has class D, and $D_{\downarrow} = \{D\}$ (i.e., D has no subclasses). In this case, regardless of whether f is declared as high or low, $o.f$ is to be considered as L since its class is constant.

When a developer or a user wants to analyze a method m[*11] in the class C with respect to class-based Information Flow, she or he will call

$$(\text{ANALYZE}_{\sigma_0}\,(\{C\}_{\mathsf{L}}.m(V_1,..,V_k)))^{\mathsf{L}}$$

where:

- the initial abstract state σ_0 is such that (1) every field of every class is assigned to an abstract value according to its declared type and the security policy (Section 4); and (2) there are no local variables (since we are entering a method);
- $\{C\}$ is a singleton because the analysis focuses on a specific instance of m;
- the abstract value $\{C\}_{\mathsf{L}}$ corresponding to the caller has the flag L because, in principle, it will be "provided" by the attacker when calling m;
- for the same reason, every V_i is $(D^i_{\downarrow})_{\mathsf{L}}$ where D^i is the class of the corresponding formal parameter;
- the abstract execution will be carried out under the flag L, since the attacker is supposed to have access to m.

Such abstract execution computes a *denotation* δ_m for m, i.e., the behavior of the method with respect to the security property. This is well known in many areas of semantics and static analysis: basically, δ_m maps states to states (plus the return value), and $\delta_m(\sigma)$ is the final abstract state (plus the return value) when executing m in σ. If m calls another method m', then it is not possible to provide a denotation for m without first analyzing m' (or many instances of it, if the class of the caller is not statically decidable). Therefore, the analysis needs to

- studying m using a *bottom value* $\langle \emptyset_{\mathsf{L}}, \sigma_{\perp} \rangle$ as the temporary denotation for m' — i.e., in the place of $\left(\text{ANALYZE}_{\sigma}\left(\mathcal{X}_{\phi}.m'(\overline{V})\right)\right)^{\phi'}$ — where $\sigma_{\perp}(id) = \emptyset_{\mathsf{L}}$ for every id; here, σ, $\mathcal{X}$, ϕ, ϕ' and the abstract values $\overline{V}$ for the actual parameters refer to the program point where m' is invoked in the body of m;
- analyze m' starting from σ, $\mathcal{X}$, ϕ, ϕ', and $\overline{V}$, thus giving a new result $\delta_{m'}$ for $(\text{ANALYZE}_{\sigma}\,(\mathcal{X}_{\phi}.m'(\dots)))^{\phi'}$;
- going back to m and recomputing its denotation with the newly obtained $\delta_{m'}$

[*11] Even if there is more than one m declared in C, they are different by their signature.

instead of the bottom value used in the first step.

Since m' may, in turn, call other methods, and even m itself (*mutual recursion*), the analyzer needs to implement a *fixpoint*, as usual in similar analysis techniques. A *queue* is kept to manage the evaluation order of method instances. An element of the queue is a pair $\langle method\ description, initial\ information \rangle$, where the method description is basically the method signature, and the initial information is the abstract information corresponding to the calling point of the method. When the call to m' is found in m, the signature of m' and the abstract information at the call point (i.e., σ, $\mathcal{X}$, ϕ, ϕ', and $\overline{V}$ as defined above) are inserted in the queue (in other words, m' is marked as "to-be-analyzed"). When the analysis of m has finished (but is still incomplete because there was no final denotation for m'), the algorithm extracts a signature from the queue and analyzes it starting from the corresponding initial information. After m' is extracted and analyzed (which can involve inserting other methods, possibly m, in the queue, and iterating the process), a denotation $\delta_{m'}$ is computed. If the new denotation is not smaller (point-wise, w.r.t. the ordering on abstract states) than the previous one obtained for m', then the system inserts in the queue all the methods which called m' with the same initial information, because the new denotation obtained for m' can lead to a change in the denotation of such methods. On the other hand, if the new denotation is smaller than or equal to the old one (i.e., no *new* information was acquired), then nothing is inserted in the queue. More formally:

$$\begin{aligned} \delta' \leq \delta'' &\equiv \forall \sigma.\, \delta'(\sigma) \leq_{\mathcal{S}} \delta''(\sigma) \\ \sigma' \leq_{\mathcal{S}} \sigma'' &\equiv \forall id.\, \sigma'(id) \leq \sigma''(id) \end{aligned}$$

The process goes on until the queue is empty, i.e., when no denotation has to be recomputed due to a change in another denotation. This means that a fixpoint has been reached. Termination is guaranteed because denotations are always increasing (*monotonicity*) with respect to the ordering above, and abstract domains are finite (since the number of classes cannot be infinite).

4.4 Soundness

In order to discuss the soundness of the formalization, it is necessary to understand what the security level of data means *in the concrete world*. We concentrate on the evaluation of expressions, depicted in Section 4.1.

Let ρ be the upper closure operator which maps a value o to the smallest class C containing it (i.e., such that o::C). It can be easily extended point-wise to sequences. As for states, the abstraction $\sigma = \rho(\varepsilon)$ of a concrete execution state ε maps every local

variable v to its smallest class, and, for a field $C.f$,

$$\sigma(C.f) = \cup\{D \mid \exists \varepsilon.\sigma = \rho(\varepsilon) \wedge \exists o \text{ in the heap of } \varepsilon.\ \varepsilon(o)::C \wedge \varepsilon(o.f)::D\}$$

Consider two concrete states ε_1 and ε_2, and their abstract counterparts $\sigma_1 = \rho(\varepsilon_1)$ and $\sigma_2 = \rho(\varepsilon_2)$. The Abstract-Non-Interference condition (derived from definitions in Section 3.3) for an expression e amounts to saying that evaluating e in σ_1 and σ_2 yields two objects (o_1 and o_2, respectively) of the same class (i.e., there exists a class D such that $o_1::D$ and $o_2::D$), provided all *low* fields $C.f$ have the same class in ε_1 and ε_2. More formally: for every ε_1 and ε_2, $\sigma_1 = \rho(\varepsilon_1)$ and $\sigma_2 = \rho(\varepsilon_2)$

$$(\forall C.f\ low.\ (\sigma_1)(C.f) = (\sigma_2)(C.f)) \Rightarrow [\![e]\!]\,\sigma_1 = [\![e]\!]\,\sigma_2$$

If the above condition holds, then e is said to be *exp-low*. It must be pointed out that, in the spirit of the *abstract* version of ANI (Section 2.4), the semantics $[\![]\!]$ which is used to evaluate e is an abstract one, only dealing with class information. The soundness of the analysis of (terminating) expressions is stated in the following proposition.

Proposition 4.1
Let $\mathcal{X}_\phi$ be the result of computing $[\![e]\!]^\sharp_{\phi_0}(\sigma)$. Then, for every concrete state ε whose abstraction $\rho(\varepsilon)$ is σ, the following holds for $[\![e]\!](\varepsilon)$ provided the concrete computation actually terminates:

1. there exists $D \in \mathcal{X}$ such that $[\![e]\!](\varepsilon)::D$ (here, the semantics is concrete); and
2. ϕ is L only if e is *exp-low*.

Proof

- First, ϕ_0 is taken to be L, so that there is no need to consider the operator $[]^{\phi_0}$. This choice does only affect item 2. of the proposition.
  - Item 1. holds easily by observing that, as far as the computation of the class set is concerned, the algorithm describes a standard class analysis, which over-approximates the set of runtime classes.
  - Item 2. follows from observing that $\phi = \mathsf{L}$ may hold if (i) e has only low sub-expressions, so that combining them is low; or (ii) the class of the e is unique and decidable. In both cases, e is clearly *exp-low*.
- In the case of $\phi_0 = \mathsf{H}$, the operator $[]^{\phi_0}$ must also be taken into account. Here, there two possibilities:
  - LS does not lower the result to L: then soundness is trivially guaranteed;
  - the result, which was raised by $[]^{\mathsf{H}}$, is lowered by LS: in this case, the class of e is guaranteed to be fixed, so that e is *e-low*.

If e is or contains a method call, then the proof needs soundness for the analysis of method execution (see below).

■

This amounts to saying that (1) the set of classes inferred by the abstract semantics includes the real, concrete class of the expression in any compatible concrete state (where "compatible" means that the abstraction of ϵ is included in σ); and (2) the analyzer considers the expression as low-security (i.e., where no high-security information has been propagated) only if the expression really is.

The rest of the soundness proof comes from the the soundness of standard AI-based tools for program analysis, mentioned in the discussion of Section 4.3 about the interprocedural part of the formalization (combining denotations). In particular, the analysis of statements and method execution obeys the general definition of a sound abstract semantics for an Object-Oriented language [18, 22].

§5 An example

The code in Figure 5 shows an important main feature of the analysis: it is possible to accept programs which would be rejected by a standard Non-Interference analyzer. We focus on $\mathsf{Aclass.flow(Dsup, Dsub1)Esup}$: it computes an expression by calling three methods. We study whether the return value depends on high information when d1 and d2 are high.

First, we note that d1 may belong to any of the Dx classes, except the abstract classes Dsup and Dsub2sub. Therefore, getC() must be evaluated for all non-abstract classes. The key point in getC() is that, although the declared return class is Csup, it can be statically inferred to be either Csub1 or Csub2. Therefore, by least upper bound, the abstract evaluation of d1.getC() yields $\{\mathsf{Csub1}, \mathsf{Csub2}\}_{\mathsf{L}}$. This was possible since Dsup is abstract, so that Csup — i.e., the return type of Dsup.getC — is not to be dealt with. Here, it is possible to see how using sets of classes can indeed make a difference (see Section 4): excluding Csup.getE() from the computed instances allows one to get rid of Csup.getE, which returns Esup. On the other hand, both Csub1.getE() and Csub2.getE() return Esub2 (in the former, analysis of guards permitted to exclude the *else* branch).

Consequently, Esub2.chooseOne() is the only chooseOne instance to be considered. This shows another use of using class sets: we do not need to consider the downward closure of Esub2 — as it would have happened using a less expressive representation of abstract values — which would have involved a flow from the high-security field oneHighCfield via Esub2sub.chooseOne(). Instead, no illicit flows are

```
class Aclass {
  Esup flow(Dsup d,Dsub1 d1) { return ((d.getC()).getE(d1)).chooseOne(); } }

class Csup {
  flowL Esup oneLowEfield;  flowH Esup oneHighEfield;
  Esup getE(Dsup d) { return oneHighEfield; } }

class Csub1 extends Csup {
  Esup getE(Dsup d) { if (d instanceof Dsub1)
    { return new Esub2(d) } else { return oneHighEfield }}}

class Csub2 extends Csup {
  Esup n(Dsup d) { return new Esub2(d); } }

class Csub3 extends Csup {
  Esup getEaux(Esup e) { return new Esup(); }
  Esup getE() { return getEaux(oneHighEfield); } }

abstract class Dsup {
  abstract Csup getC(); }

class Dsub1 extends Dsup {
  Csup getC() { return new Csub1(); } }

class Dsub1sub1 extends Dsub1 {}

class Dsub1sub2 extends Dsub1 {}

class Dsub2 extends Dsup {
  Csup getC() { return new Csub2(); } }

abstract class Dsub2sub extends Dsub2 {}

class Dsub2subSub1 extends Dsub2sub {
  Csup getC() { return new Csub2(); } }

class Dsub2subsub2 extends Dsub2sub {
  Csup getC() { return new Csub2(); } }

class Esup {
  flowH Csup oneHighCfield;  flowL Csup oneLowCfield;
  Csup chooseOne() { return oneLowCfield; } }

class Esub1 extends Esup {}

class Esub2 extends Esup {
  Esub2(Dsup d) {} }

class Esub2sub extends Esub2 {
  Csup chooseOne() { return oneHighCfield; } }
```

Fig. 5 The Java$_{IF}$ code

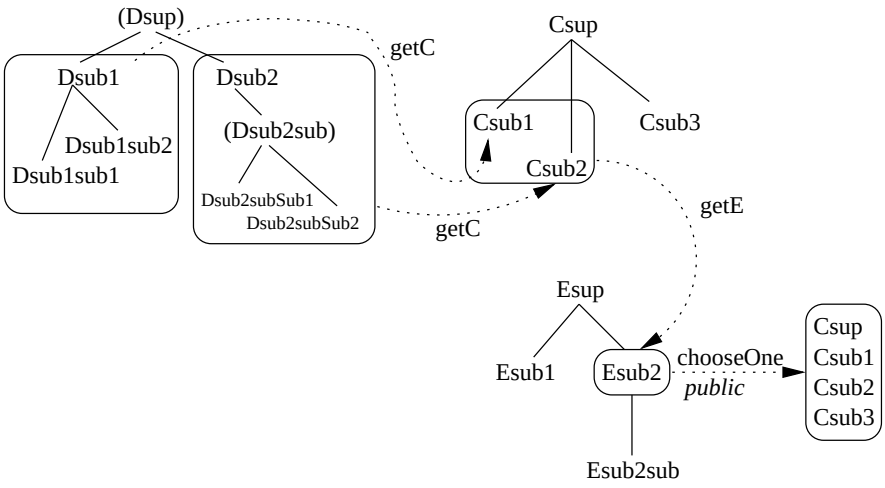


Fig. 6 The abstract behavior of methods

detected because the low-security oneLowCfield is returned by the Esub2 instance of chooseOne(); the final value comes to be $\{\mathsf{Csup}, \mathsf{Csub1}, \mathsf{Csub2}, \mathsf{Csub3}\}_{\mathsf{L}}$. Figure 6 shows the analysis in its main steps.

§6 Conclusions and future work

The present work introduces an application of Abstract Non-Interference to a non-trivial subset of Java, where the class of data is the property of interest. By defining abstract properties as classes, the task of detecting illicit information flows can be reduced to finding low-security (or *public*, as in most part of the literature) data whose type after a method execution depends on the initial type of some high-security (*private*) data. This is a sort of type-based dependency analysis, which tracks how class information propagates. The result is substantially different from standard Non-Interference verification, since

- it models a weaker property; i.e., it does not distinguish between any two instances of the same class;
- it is not completely syntax-based, so that data can be considered as low-security even if they are (syntactically) made of high-security sub-parts (e.g., consider o and f in $o.f$: usual type systems are typically syntax-based, so that $o.f$ cannot be low-security unless o is).

As shown in examples (Section 1.3), this security framework can model requirements such as hiding the implementation details of a data structure, or preventing external

users from observing information contained in the class of data.

6.1 Future work

The main direction of future work is towards the implementation of a real analyzer; the current tool is a prototype which can be improved and optimized in several ways (e.g., the efficiency of the fixpoint, the internal representation of abstract values, etc.). This would lead to a more efficient and possibly more precise analysis, capable of soundly accepting (i.e., proving the correctness of) a wider set of programs.

Moreover, a larger subset of Java would be worth considering, in particular the use of exceptions, which can be an important source of illicit flows. The definition of Abstract Non-Interference in such an enlarged context deserves further work, also from the theoretical point of view.

Finally, the present framework could be part of a *Proof-Carrying-code* [16] architecture. In a Proof-Carrying-code schema for Java, the code user may want to be sure that the bytecode program (s)he receives is safe, and the program is not executed unless the producer provides a correctness proof for the desired (and required) security property. The inclusion in a Proof-Carrying-code architecture would involve translating the analysis — or its result on source code, if the soundness of the compiling process w.r.t. the security property is also proven — to the level of bytecode, since the user will be interested in verifying low-level programs rather than source code.

Acknowledgment This work was funded in part by the Information & Communication Technologies program of the European Commission, Future and Emerging Technologies (FET), under the IST-15905 *MOBIUS* project and the ICT-231620 *HATS* project, by the Spanish Ministry of Science and Innovation (MICINN) under the TIN-2005-09207 *MERIT* and the TIN-2008-05624 *DOVES* project, the HI2008-0153 (Acción Integrada) project, the UCM-BSCH-GR58/08-910502 Research Group and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project and the S2009TIC-1465 *PROMETIDOS* project.

References

1) M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 147–160, San Antonio, Texas, USA, January 1999. ACM Press.

2) T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In S. Jones, editor, *Proc. ACM Symp. on Principles of Programming Languages*, Charleston, South Carolina, USA, January 2006. ACM Press.

3) A. Banerjee and D. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programming*, 2(15):131–177, March 2005.

4) F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao System. Reference Manual (v1.13). Technical report, School of Computer Science (UPM), 2006. Available at `http://www.ciaohome.org`.

5) I. Cartwright and M. Felleisen. The semantics of program dependence. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, pages 13–27, Portland, Oregon, USA, 1989. ACM Press.

6) S. Chong and A. Myers. End-to-end enforcement of erasure and declassification. In *Proc. IEEE Computer Security Foundations Symposium*, Pittsburgh, Pennsylvania, USA, June 2008.

7) P. Cousot. Types as abstract interpretations, invited paper. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 316–331, Paris, France, January 1997. ACM Press.

8) P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 238–252, Los Angeles, California, USA, 1977. ACM Press.

9) P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, USA, 1979. ACM Press.

10) R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In N. Jones and X. Leroy, editors, *Proc. ACM Symp. on Principles of Programming Languages*, pages 186–197, Venice, Italy, January 2004. ACM Press.

11) R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *Proc. Conf. on Computer Science Logic*, volume 3210 of *Lecture Notes in Computer Science*, pages 280–294, Karpacz, Poland, 2004. Springer-Verlag.

12) R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the Association for Computing Machinery*, 47(2):361–416, 2000.

13) J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20. IEEE Computer Society Press, 1982.

14) I. Mastroeni and D. Zanardini. Data Dependencies and Program Slicing: from Syntax to Abstract Semantics. In *Proc. Workshop on Partial Evaluation and Program Manipulation*, pages 125–134, San Francisco, California, USA, January 2008. ACM Press.

15) A. Myers. JFlow: practical mostly-static information flow control. In *Proc. ACM Symp. on Principles of Programming Languages*, pages 228–241, San Antonio, Texas, USA, January 1999. ACM Press.

16) G. Necula. Proof-Carrying Code. In *Proc. ACM Symp. on Principles of Programming Languages*, Paris, France, January 1997. ACM Press.

17) J. Palsberg and M. Schwartzbach. Object-oriented type inference. In A. Paepcke, editor, *Proc. Conf. on Object-Oriented Programming Languages, Systems, and Applications*, volume 26 of *ACM SIGPLAN Notices*, pages 146–161, Phoenix, Arizona, USA, November 1991. ACM Press.

18) Uday S. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *In Proc. ACM Conference on Lisp and Functional Programming*, pages 289–297. ACM Press, 1988.

19) X. Rival. Abstract dependences for alarm diagnosis. In K. Yi, editor, *Proc. Asian Symp. on Programming Languages and Systems*, volume 3780 of *Lecture Notes in Computer Science*, pages 347–363, Tsukuba, Japan, November 2005. Springer-Verlag.

20) A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

21) S. Secci and F. Spoto. Pair-Sharing Analysis of Object-Oriented Programs. In C. Hankin, editor, *Proc. Symp. on Static Analysis*, volume 3672 of *Lecture Notes in Computer Science*, pages 320–335, London, UK, August 2005. Springer-Verlag.

22) F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Transactions on Programming Languages and Systems*, 25(5):578–630, September 2003.

23) D. Zanardini. Higher-Order Abstract Non-Interference. In P. Urzyczyn, editor, *Proc. Int. Conf. on Typed Lambda Calculi and Applications*, volume 3461 of *Lecture Notes in Computer Science*, Nara, Japan, April 2005. Springer-Verlag.

24) D. Zanardini. Abstract Non-Interference in a fragment of Java bytecode. In *Proc. ACM Symp. on Applied Computing*, Dijon, France, April 2006.

25) D. Zanardini. Analyzing Non-Interference with respect to Classes. In *Proc. Italian Conf. on Theoretical Computer Science*, Roma, Italy, October 2007. World Scientific.

26) D. Zanardini. The Semantics of Abstract Program Slicing. In *Proc. Int. Workshop on Source Code Analysis and Manipulation*, Beijing, China, September 2008. IEEE Computer Society Press.

27) S. Zdancewic and A. Myers. Robust declassification. In *Proc. IEEE Computer Security Foundations Workshop*, pages 15–23, Cape Breton, Nova Scotia, Canada, June 2001.