ANALYZING NON-INTERFERENCE WITH RESPECT TO CLASSES

D. ZANARDINI

CLIP, Universidad Politécnica de Madrid, E-28660 Boadilla del Monte, Madrid, Spain
E-email: damiano@clip.dia.fi.upm.es

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: a property models the set of numbers satisfying it. The present work proposes an Object-Oriented, Java-based formulation of ANI, where data take the form of objects, and the observed property comes to be their class. Relevant data are stored in fields; the execution of a program is taken to be the invocation of some (public) method by an external user; a class is *secure* if, for all its public methods, the class of its public data after the execution does not depend on the initial class of its private data. The relation ANI lies in the representation of abstract domains as class hierarchies: upper closure operators map objects into the *smallest* class they belong to. An analyzer for a non-trivial subset of Java is illustrated, which is sound since programs are never misclassified as secure.

Keywords: Verification; Information Flow; Classes as Properties; Abstract Interpretation

1. Introduction

Abstract Non-Interference (ANI)[8] provided a well-founded and parametric framework where the standard notion of Non-Interference (NI)[10,16] can be relaxed. Such a weakening is useful since many programs do not satisfy NI because it requires the separation between public (information every user can observe) and private (to protect from unauthorized users) data to be complete; i.e., the public output must not depend in any way on the private input (there are no *illicit flows* from private to public data). In practice, it is often the case that some flow should be allowed, as long as (i) attackers cannot detect them; or (ii) there is no need to protect some *aspects* of the revealed information. Several techniques have been proposed for weakening NI; existing approaches either limit the observational power of attackers, or declassify the released information. ANI belongs to the first family: it considers attackers which can only observe properties of data (not exact, *concrete* data). Therefore, an illicit flow may not be visible to attackers since the property does not change; such a flow should be considered as harmless, and the program

is safely accepted as secure. Properties are described by abstract domains.[6]

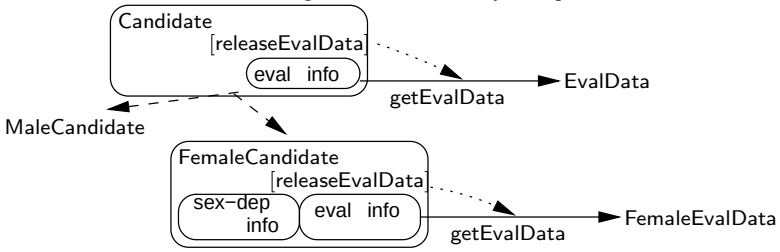
Abstract Non-Interference originally refers to simple imperative languages with global integer variables. Data properties are sets of values (e.g., *to be even* is the set of even numbers). Our work defines information flow in an Object-Oriented framework, modeled on Java. Values take the form of objects. The main idea is to see a class, which represents a collection of objects with the same structure, as an abstract property: to observe a property amounts to see the class of the object the property refers to. 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;[5,14,17] therefore, class-based reasoning in the framework of information flow can be seen as an interesting approach. An analyzer is shown, which checks ANI in method execution, relying on class-based dependencies for a non-trivial subset of Java (exceptions and threads being the main missing features). Its purpose is to check if the output class of non-private fields depends on the initial class of secret fields; in this case, an illicit *information flow* is said to occur. The algorithm is sound (a discussion on soundness is given), i.e., programs are rejected if illicit flows may occur. Examples show methods which are safely detected as secure even though the classical notion of NI is not satisfied.

Information flow security[10,16] relies on *data dependencies*.[1,4] Standard information flow for an OO framework considers data propagation from private to public fields; it is analyzed by means of types[12] or logic,[2] the latter being potentially more precise. The foundational work on ANI 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,[9] which checks ANI by inferring secrecy assertions via Hoare triples. Assertions are combined syntactically to derive safety proofs; it is also shown how to derive attackers which do not violate a security policy. ANI was recently extended to functional languages;[18] in this work, some ideas for an analysis algorithm were provided as a type system. Data are equipped with *security types* describing the behavior w.r.t. the private-public boundary. Type/class inference for OO languages[14,17] aims at verifying that data cannot belong, at runtime, to the wrong class. The link between type information and data properties has been underlined[5] in Abstract Interpretation. Since information flow and dependencies are closely related, ANI involves a notion of *abstract data dependency*,[15] whose computation in the general case is part of ongoing work.[11]

Introductory example Consider the recruitment unit of an enterprise (Fig. 1). People applying for a job should fill in their personal data in a questionnaire; in

order to avoid discriminating candidates on sex, race etc., only a subset of data (experience, skills, spoken languages) can be used in the (non-automatic) evaluation process. Let personal information be stored in objects of class Candidate; some data in this class are sex-dependent, and are going to be used for bureaucratic reasons *after* the candidates have been evaluated; this means that this information exists in the database from the beginning, yet should be kept invisible to evaluators. In the spirit of OO programming, it makes sense to define two subclasses FemaleCandidate and MaleCandidate of Candidate in order to consistently store sex-dependent data. The information which should be used in the evaluation process can be fetched by calling the method getEvalData(CandidateID id)EvalData [a], which accesses a *private* Candidate object by its id and returns an EvalData object containing data to be evaluated. In the same spirit, the system, possibly coming from previous uses, may implement two subclasses FemaleEvalData and MaleEvalData of EvalData, and private methods (to be invoked by getEvalData(id)) Candidate.ReleaseEvalData()EvalData in the superclass, and the redefined FemaleCandidate.ReleaseEvalData()FemaleEvalData and MaleCandidate.ReleaseEvalData()MaleEvalData. Yet, in this case, the class of the result of getEvalData(id) would reveal information about sex. Such a code is not ill-designed w.r.t. the OO programming style; yet, it is not adequate if security requirements include forbidding the sex of candidates to be revealed.

Fig. 1. The introductory example



2. Preliminaries

Abstract Interpretation Abstract Interpretation (AI)[6] is a theory for systematically deriving non-standard, approximated program semantics. *Abstract domains* can be formulated either in terms of Galois connections or closure operators.[7] An

[a] 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.

upper closure operator (uco) on a poset $\langle \mathcal{C}, \leq \rangle$ is a function $\rho : \mathcal{C} \mapsto \mathcal{C}$ monotone, idempotent and extensive. The set of all ucos on $\mathcal{C}$ is $\mathrm{UCO}(\mathcal{C})$. A closure operator is uniquely determined by the set of its fixpoints (called *abstract values*); this set is (isomorphic to) the abstract domain $\mathcal{A}$ approximating the concrete domain $\mathcal{C}$. A set $X \subseteq \mathcal{C}$ is the set of fixpoints of a uco iff it is a *Moore-family*, i.e., $X = \mathcal{M}(X) = \{\wedge S \mid S \subseteq X\}$. In the following, $\rho(v)$ will stand for $\rho(\{v\})$ whenever $\{v\} \in \mathcal{C}$. Abstraction formalizes the idea that $\mathcal{A}$ is simpler than $\mathcal{C}$, being a subset. On the other hand, a computation $f_{\mathcal{A}}$ on $\mathcal{A}$ can be less precise than $f_{\mathcal{C}}$ since values $V \in \mathcal{C} \setminus \mathcal{A}$ cannot be used. $f_{\mathcal{A}}$ approximates $f_{\mathcal{C}}$ by providing the abstract version of constants and operators. It is a *sound* abstraction if the abstract result is always a correct approximation of the concrete result: $\forall x.\ f_{\mathcal{C}}(x) \leq f_{\mathcal{A}}(x)$ (read $\leq$ as *more concrete* or *more precise*). 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 pointwise, is also a complete lattice where $\mathrm{ID} = \lambda V.V$ describes the identity abstraction ($\mathcal{A} = \mathcal{C}$, no loss of information) and $\mathrm{TOP} = \lambda V.\top$ is the trivial abstraction mapping $\mathcal{C}$ into a singleton $\mathcal{A} = \{\top\}$. The *reduced product* $\sqcap$[7] of a set $\{\mathcal{A}_i\}$ is the most abstract among the domains which are more concrete than each A_i: formally, $\sqcap_i \mathcal{A}_i = \mathcal{M}(\cup_i \mathcal{A}_i)$. Notation will be often abused by referring to ρ as the set of its fixpoints; i.e., $V \in \rho$ if V belongs to the domain $\mathcal{A}$ generated by ρ. In the present work, AI plays two roles: (i) providing the basis for defining the security property; (ii) giving the background for developing the static analyzer.

Information Flow If a user wants to keep some data confidential, (s)he can take as a requirement that information cannot go from private to public data. An untrusted user, which can only see public data, should not be able to guess anything about what is protected (private). Such a policy allows programs to use private data as long as the visible output does not reveal information about it. In *Non-Interference* (NI),[10] 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*, L) part of $[\![\mathcal{P}]\!]$ on the input (h, l), divided into a private (*high-security*, H) part h and a public part l. In other words, there must be no *information flow* from h to l: h and l do not interfere.

Abstract Non-Interference Non-Interference can be weakened by modeling secrecy relatively to some observable property. The observational power of an attacker is limited, and a *secure* program preserves secrecy only as regards the information the attacker can observe. Let the concrete domain be the set of all prop-

erties (e.g., $\wp(\mathbb{N})$ for integers, where a property is identified with the set $P \subseteq \mathbb{N}$ of values satisfying it), representing which values can be distinguished by attackers. Ucos describe the ability of an attacker: if one has precision ρ, then (s)he cannot distinguish v_1 and v_2 if $\rho(v_1) = \rho(v_2)$ (i.e., values having the same property w.r.t. ρ). $\mathcal{P}$ is secure for domains η and ρ (written $[\eta]\mathcal{P}(\rho)$) if no flows are detected by observing public input (resp. output) data only up to a precision η (resp. ρ):

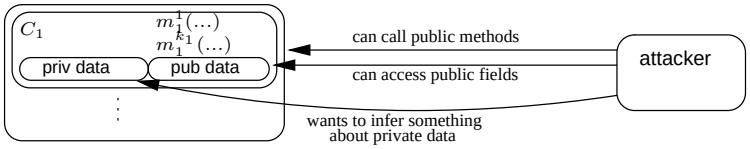
$$\forall h_i, l_i. \quad \eta(l_1) = \eta(l_2) \implies \rho([\![\mathcal{P}]\!]^{\mathsf{L}}(h_1, l_1)) = \rho([\![\mathcal{P}]\!]^{\mathsf{L}}(h_2, l_2))$$

If l_1 and l_2 cannot be distinguished, then it is not possible to guess h from the (abstracted) output. Standard NI is a special case of ANI, equivalent to $[\text{ID}]\mathcal{P}(\text{ID})$ (no abstraction). 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, (η, ϕ, ρ)-secrecy[b], rules out deceptive flows by computing $[\![\mathcal{P}]\!]$ on an abstraction of the input:

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

In the following, we deal with (ρ, ρ, ρ)-secrecy: the class hierarchy will identify the abstract domain to be considered.

Fig. 2. The program and the attacker



A realistic approach to OO Information Flow Most information flow properties were originally defined on simple languages. Yet, advanced features in present-day OO languages make a correct definition much more difficult to obtain. Particularly, in order to reasonably adapt the NI notion *what an attacker can see at the output does not allow him or her to acquire (abstract) information about secret input* to Java, we must clarify (i) what a running program is; (ii) what an attacker is and can do; (iii) which data we want to protect (see Fig. 2 for a picture).

Running programs. A Java program is a collection of cooperating classes; we are interested in detecting if the legal interaction with a class C may disclose its protected data. The *program execution* takes the form of the invocation of a public method m of C; a class is secure if all its public methods are secure, i.e., it

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

is not possible to access its private data by interacting with its public part.

Attackers. We see attackers as programs which can interact with external classes C and aim at breaking their secrets by calling public methods. An attacker cannot observe auxiliary information as the amount of allocated memory or the execution time of, which would involve additional flows.

Public and private data. Java comes with field modifiers **public**, **private** and **protected**, which model security requirements by regulating data access. However, NI analysis does not only deal with *direct* access: secret information should not *propagate* through the computation, as shown in the following code:

```
public class D {
  public C l;    private C h;    public C m() { return h; } }
```

This declaration is legal; yet, the private h is propagated via the return value of m (public); this means that NI may be violated without breaking the access policy. Our security policy requires that there be no (abstract) flow from **private** to **public** fields or return values. If f is declared as private in D, then all the objects stored in the field f of D objects are considered as private; if we want some object $o : D$ not to protect its field f, then it must be an instance of $D' \sqsubset D$, where f is public.

3. Class-Oriented Abstract Non-Interference

ANI is parametric on the degree of precision attackers have in reading public information. Given a universe of values, properties are sets of values sharing some common behavior. In an OO framework, classes have a similar purpose: they identify objects with the same internal structure. Therefore, modeling properties with classes is quite natural. In our setting, program classes are the properties attackers can observe. Identifying properties with classes reduces property checking to a class-directed program analysis.[14] Classes are ordered by the subclass relation: a class is a superset of any subclass; a subclass models a sub-property, i.e., more precise since it is satisfied by less values. Names C, D etc. will denote either sets of semantic values, classes or properties; o stands for objects or values. The subclass relation $C' \sqsubset C$ is basically $C' \subseteq C$ if C' and C are considered as sets of values. The predicate $o : C$ (or $o \in C$) holds if o has class C, while $o :: C$ holds iff $o : C$ and there is no $D \sqsubset C$ s.t. $o : D$. Finally, $C_{\downarrow} = \{D | D \sqsubseteq C\}$.

Class hierarchies as abstract domains Consider an object set C and a concrete domain $\mathcal{C} = \wp(C)$. A class hierarchy rooted at C[c] identifies a subset of properties $D \in \mathcal{C}$, i.e., an abstract domain ρ; $\rho(D)$, for $D \in \wp(C)$, is the smallest class

[c]That is, with C as the greatest class. Usually, we may want to ignore the Object superclass, thus obtaining a set of hierarchies instead of a single hierarchy rooted at Object.

$D' \in \rho$ s.t. $D \sqsubseteq D'$. Since *multiple inheritance* is not allowed in Java, 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 ucos, the Bot_ρ empty class is supposed to exist for every hierarchy ρ. As a consequence, representable domains are those where classes are either related by $\sqsubset$ or disjoint; languages allowing multiple inheritance do not suffer from this limitation.

The link to the Abstract Non-Interference theory 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. $\mathcal{E}$ denotes the set of *execution states*. The visible part of $\varepsilon \in \mathcal{E}$ consists of public instance fields of $\mathcal{O}$ objects, and public static fields of $\mathcal{C}$ classes. Let $\varepsilon_1, \varepsilon_2 \in \mathcal{E}$; the *indistinguishability condition* $\text{IC}\,(\varepsilon_1, \varepsilon_2)$ holds if (i) for every C, $o_1, o_2 : C$ (note the :) and public field f, the value $o_1.f$ in ε_1 and $o_2.f$ in ε_2 belongs to the same class ($\exists D.\ \varepsilon_1(o_1.f) :: D \wedge \varepsilon_2(o_2.f) :: D$); (ii) for every D and public static field g, $D.g$ belongs to the same class in both states.

Given two sequences of values $\overline{v}_1$ and $\overline{v}_2$, $\text{IC}\,(\overline{v}_1, \overline{v}_2)$ 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 public data. An attacker may try to break $\mathcal{P}$ secrets by running $o.m$ of $o \in \mathcal{O}$, or a static $C.m$ of $C \in \mathcal{C}$. The *Abstract Non-Interference condition* $\text{ANI}\,(m)$ holds for m if $\text{IC}\,(\varepsilon_1, \varepsilon_2) \wedge \text{IC}\,(\overline{v}_1, \overline{v}_2)$ implies $\text{IC}\left(\left[\!\left[o.m\,(\overline{v}_1)\right]\!\right]_{\varepsilon_1}, \left[\!\left[o.m\,(\overline{v}_2)\right]\!\right]_{\varepsilon_2}\right)$, where $\left[\!\left[o.m\,(\overline{v})\right]\!\right]_{\varepsilon}$ 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 (in the abstract sense) in their private part, we cannot distinguish the output states. The attacked program $\mathcal{P}$ is secure (written $\text{ANI}\,(\mathcal{P})$) if this condition holds for every public method. It is easy to see that this definition of Abstract Non-Interference is quite close to the original one; in fact, it is an adaptation to our OO framework.

4. ANI analysis for class information

This section describes the *class-flow* analyzer in its main features. It is basically an AI-based tool, implemented in the `Ciao`[3] system, which performs an *abstract execution* of a method and tries to detect illicit abstract flows from private fields to public fields or the returned value. Apart from some minor simplifications, Java is supported in its main features, including static fields and methods, abstract classes and polymorphism. Exceptions are the most important direction of future work. Analysis relies on *abstract states* $\sigma \in \mathcal{E}$, which consist of a global part (the fixed set of class fields) and the framestack storing local environments. We note that, due to how private and public data are defined (Sec. 2), there is no need to keep an abstract *heap*. $\sigma(C.f)$ is the class of the field f of C in σ, while $\sigma(x)$ refers

to the class of x in the active frame of σ. Abstract states map fields and variables to *abstract values* $V = \mathcal{C}_\phi \in \mathcal{V}$, where $\mathcal{C}$ is a set of classes and $\phi \in \{\mathsf{H}, \mathsf{L}\}$ is a *security flag* indicating the security level of data. Private fields are initially $(C_\downarrow)_\mathsf{H}$, where C is the declared class. *Least upper bound* $\sqcup$ is ${\mathcal{C}'}_{\phi'} \sqcup {\mathcal{C}''}_{\phi''} = (\mathcal{C}' \cup \mathcal{C}'')_{\phi' \sqcup \phi''}$, where $\mathsf{L} \sqcup \mathsf{L} = \mathsf{L}$ and $\phi' \sqcup \phi'' = \mathsf{H}$ otherwise. In type theory, types are a partial order, a type being an abstraction of a set of values;[5] when x has type τ, information about which $\tau' \leq \tau$ it can belong to is lost. Using class sets is more precise, and allows exploiting abstract declarations (Sec. 5). For example, let D be abstract, $D_1, D_2 \sqsubset D$. In our setting, $\{D_1, D_2\}_\phi$ may be computed for o declared as D; this is more precise than $(D_\downarrow)_\phi = \{D, D_1, D_2\}_\phi$, but still correct since it describes the same set (D has no instances).

In Java, illicit flows come as (i) assignment of a private value to a public variable or field (*explicit flows*); or (ii) execution of conditional statements or loop whose guard depends on private data (*implicit flows*). In detecting explicit flows, the main point is computing abstract values for expressions; a flow is soundly assumed to exist if a private value is computed for an expression assigned to a public v. 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 public depends on a private guard, then its effect may cause an illicit flow.

Abstract values In $x = e$, e may be quite complicated: e.g., a method call with side effects. Therefore, computing the security content ϕ_e is non-trivial. Fig. 3 shows how abstract values are computed (ANALYZE is shown below). The *expression semantics* $[\![]\!]^\sharp$ takes e, $\sigma \in \mathcal{E}$ and ϕ, and gives a pair (value, resulting state).

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

Fig. 3. Rules for analyzing expressions

The operator $[]^\phi$ *raises* the security level of a value: $(\mathcal{C}_{\phi'})^{\phi''} = \mathcal{C}_{\phi' \sqcup \phi''}$. The

flag ϕ in $[\![e]\!]^\sharp_\phi(\sigma)$ means that e is computed under a level ϕ (see implicit flows).

The function LS is only applied to the first element of the pair (i.e., the abstract value); it lowers the flag of a value if the class set is a singleton: LS $(\langle \mathcal{C}_\phi,\ \sigma\rangle)$ is $\langle \mathcal{C}_{\mathsf{L}},\ \sigma\rangle$ if $\mathcal{C}$ is a singleton, $\langle \mathcal{C}_\phi,\ \sigma\rangle$ otherwise. This function is very important since it shows how abstract data dependencies may differ w.r.t. their concrete counterpart. Let $\{C\}_\phi$ be $[\![e]\!]^\sharp_{\phi'}(\sigma)$, and o be the object computed by runtime (concrete) execution. Thus, $o :: C$ certainly holds (soundness). Even if e contains private data, it can be considered as L; in fact, illicit flows do not occur since the class of o is constant. Typically, $\mathcal{C}$ 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 has subclasses.

Methods and statements The function $\text{ANALYZE}_\sigma\,(V.m(\dots))^\phi$, already used above, performs the security analysis of methods. A method can be invoked as a command (when there is no return value, or it is ignored), or inside expressions. Let V and $V_1..V_k$ be abstract values obtained by previous computations. The result of $\text{ANALYZE}_\sigma\,(V.m(V_1..V_k))^{\phi'}$ is obtained by (the lub of) the abstract execution in σ of each instance $C.m$ such that $V = \mathcal{C}_\phi$ and $C \in \mathcal{C}$. More formally: $\text{ANALYZE}_\sigma\,(\mathcal{C}_\phi.m(V_1,..,V_k))^{\phi'} = \sqcup_{C\in\mathcal{C}}\left(\text{ANALYZE}_\sigma\,(C_\phi.m(V_1,..,V_k))^{\phi'}\right)$

where $\sqcup$ works on both the return value and the final state. The notation C_ϕ means that the flag of the caller is kept in the analysis of each C method instance (it is copied in this). This is another difference with respect to standard information flow analysis, and it is also applied to field access: usually, the analysis of $o.f$ or $o.m\,(\dots)$ leads to an H flag whenever o is H. In our formulation, the H content of o is not a sufficient condition to consider $o.f$ or $o.m\,(\dots)$ as private. For example, let $o : C$ be private, and $C_\downarrow = \{C, C_1, C_2\}$. In all classes, f has class D, and $D_\downarrow = \{D\}$ (i.e., no subclasses). In this case, regardless of whether f is declared as public or private, $o.f$ is considered as L since its class is unique.

The analysis of a method body computes (abstract) commands and expressions. $\sigma\,[x \leftarrow V]$ (resp. $\sigma\,[C.f \leftarrow V]$) is the *updated* state obtained by storing the abstract value V in x (resp. $C.f$); $\sigma\,[\mathcal{C}.f \leftarrow V]$ is the set extension for each $C \in \mathcal{C}$. The *upgraded* state $\sigma' = \sigma\,[v \twoheadleftarrow V]$ satisfies $\sigma'(v) = \sigma(v) \sqcup V$.

Upgrading (instead of updating) is used since it is not known which instances $C.f$ will be actually assigned; therefore, the initial value cannot be forgotten.

Example 4.1. Let o be given $\{D, D_1\}$, and $D_1 \sqsubset D$. Let $o.f$ be H before $o.f = e$, and e be L. *Updating* the state would result in a final L flag for both $D.f$ and $D_1.f$, which 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 H is kept for $D.f$ by lub.

The function $[\![b]\!]^\sharp_g(\sigma)$ computes, for a guard b, its flag ϕ in σ. States σ_1 and σ_2

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

are only different when we can infer something from the truth value: e.g., if b is x instanceof C, assuming $\neg b$ means that its class is not a subclass of C. $\textsc{exec}_\sigma\,(s)^\phi$ means that the effects of s on σ will be raised by ϕ (see def. 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 non-public guards (i.e., $\phi = \mathsf{H}$).

Global analysis and soundness The analyzer has a *global* part, dealing with method invocation and the problems arising from mutual recursion; a *security signature* is maintained for every method instance; it is updated whenever the method is successfully analyzed. If an instance is invoked when another activation is already in the framestack, then the current security signature is used instead of re-analyzing the method. A global fixpoint is performed; its adherence to the usual AI-based techniques for designing program analyzers ensures soundness. As for the intra-body part of the analyzer, the main non-standard issue about soundness is the correctness of LS, argued above: the flag is L only if the class is a constant. Because of this, $[\![]\!]^\sharp$ computes an over approximation $\mathcal{C}_\phi$ of the *optimal* (the one obtained by directly abstracting the concrete result) abstract value, i.e., $\mathcal{C}$ is a superset of the set of possible runtime classes, and ϕ is L only if indeed no flows are possible. Soundness of the use of state upgrading is also motivated above.

5. An example

The code in Fig. 4 shows the main features of the analysis; it is possible to accept programs which would be rejected by a standard NI analyzer. We focus on A.flow(D, D1)E. It computes an expression by calling three methods; we study whether the return value depends on private information when d1 and d2 are private. We note that d1 may belong to any of the Dx classes, but the *abstract* D and D5. Therefore, m() must be evaluated for all non-abstract instances. The key point in m() is that, although the declared return class be C, it can be statically inferred to be C1 or C2. By lub, d1.m() yields $\{\mathsf{C1}, \mathsf{C2}\}_\mathsf{L}$. This was possible since D is abstract, thus C (i.e., the return of D.m) is not to be dealt with. Here, using

```
class A { E flow(D d1,D1 d2) { return ((d1.m()).n(d2)).met(); } }

class C { public E f1;  private E f2;  E n(D d) { return f2; } }
class C1 extends C { private E f2;
  E n(D d) { if(d instanceof D1){ return new E2(d) } else { return f2 }}}
class C2 extends C { private E f2;  E n(D d) { return new E2(d); } }
class C3 extends C { private E f2;
  E nn(E e) { return new E(); }
  E n() { return nn(f2); } }

abstract class D { abstract C m(); }
class D1 extends D { C m() { return new C1(); } }
class D3,D4 extends D1 {}
class D2 extends D { C m() { return new C2(); } }
abstract class D5 extends D2 {}
class D6,D7 extends D5 { C m() { return new C2(); } }

class E { private C fpr;  public C fpb;  C met() { return fpb; } }
class E1,E2 extends E {}
class E3 extends E2 { C met() { return fpr; } }
```

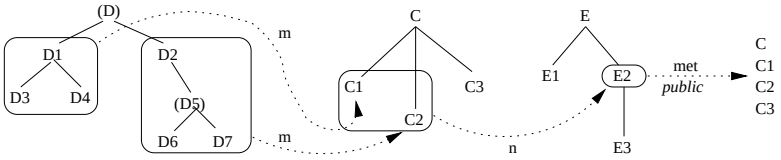


Fig. 4. The Java code (with abuse of syntax) and the abstract behavior of methods

sets of classes can indeed make a difference: excluding $\mathsf{C.n}()$ allows to get rid of $\mathsf{C.n}$, which returns E. On the other hand, both $\mathsf{C1.n}()$ and $\mathsf{C2.n}()$ return $\mathsf{E2}$ (in the first, guard analysis was used to exclude *else*). Finally, $\mathsf{E2.met}()$ is the only met instance to consider. This shows another use of using class sets: we do not need to consider the closure of $\mathsf{E2}$, which would have involved a flow from the private fpr via $\mathsf{E3.met}()$. Instead, no illicit flows occur because the public fpb is returned by the $\mathsf{E2}$ instance of $\mathsf{met}()$; the final value comes to be $\{\mathsf{C}, \mathsf{C1}, \mathsf{C2}, \mathsf{C3}\}_{\mathsf{L}}$.

6. Conclusions and future work

The present work introduces a reformulation of Abstract Non-Interference for a non-trivial subset of Java. By defining abstract properties as classes, detecting illicit information flows can be reduced to finding low security data whose class after a method run depends on the initial class of some high security data. This is a sort of type-based dependency analysis, tracking how class information propagates. The result is substantially different from standard NI verification, since (i) it models a weaker property, i.e., it does not distinguish between values of the same class; and (ii) it is not completely syntax-based, so that data can be considered as public even if they have private syntactic sub-parts (e.g., o and f in $o.f$).

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. This would lead to a more efficient and possibly more precise analysis, capable of soundly accept a wider set of programs. Moreover, a larger subset of Java would be worth considering, in particular the use of exceptions. Finally, this framework could be part of a *Proof-Carrying code*[13] architecture. In a PCC Java framework, the code user wants to be sure that the bytecode program (s)he receives is safe. The program is not executed unless the producer provides a correctness proof for the desired security property. The inclusion into PCC would involve the translation of the analysis (or of its results, by means of soundness of the compiling process) to the bytecode level, since the consumer is interested in verifying low-level programs.

References

1. M. Abadi, A. Banerjee, N. Heintze, and J. Riecke. A core calculus of dependency. In *POPL*, 1999.
2. T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006.
3. F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and G. Puebla. The Ciao System. Reference Manual (v1.13). Technical report, UPM, 2006.
4. I. Cartwright and M. Felleisen. The semantics of program dependence. In *PLDI*, 1989.
5. P. Cousot. Types as abstract interpretations, invited paper. In *POPL*, 1997.
6. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
7. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
8. R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *POPL*, 2004.
9. R. Giacobazzi and I. Mastroeni. Proving abstract non-interference. In *CSL*, 2004.
10. J. Goguen and J. Meseguer. Security policies and security models. In *SSP*, 1982.
11. I. Mastroeni and D. Zanardini. The calculus of Abstract Dependencies. unpbl., 2007.
12. A. Myers. JFlow: practical mostly-static information flow control. In *POPL*, 1999.
13. G. Necula. Proof-Carrying Code. In *POPL*, 1997.
14. J. Palsberg and M. Schwartzbach. Object-oriented type inference. In *OOPSLA*, 1991.
15. X. Rival. Abstract dependences for alarm diagnosis. In *APLAS*, 2005.
16. A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, (1), 2003.
17. F. Spoto and T. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM TOPLAS*, (5), 2003.
18. D. Zanardini. Higher-Order Abstract Non-Interference. In *TLCA*, 2005.