# Abstract Non-Interference in a fragment of Java Bytecode

Damiano Zanardini
Dipartimento di Informatica, Università di Verona
Strada Le Grazie 15, I-37134 Verona, Italy
zanardini@sci.univr.it

## ABSTRACT

This work presents a program analyzer for checking Abstract Non-Interference in a fragment of Java bytecode. Abstract Non-Interference is an information flow property which is weaker and more general than standard Non-Interference, since it can allow some selected parts of secret information to flow into the public part of a program. The motivation for such a weakening is that some flows are indeed useful in real-life applications. The amount of allowed flows is encoded into abstract domains, which characterize the degree of precision of a potential attacker in observing data; flows are forbidden as long as they can be observed and exploited by attackers. Abstract values describe possible values of programs in different executions. Basic features of Java bytecode are considered; advanced topics, such as method calls, objects and exceptions, are also discussed. A program is said to be secure if analysis computes a state which does not contain private information in public places; information flows can exist only as long as the attacker has not enough observational power to see and exploit them.

## 1. INTRODUCTION

Non-Interference [7] is an information flow [10] property of programs, meaning that it is not possible to distinguish two executions by their public output if they only differ on the private part of the input. The addressed problem is to avoid that an attacker, which can only see the public part of states, could guess anything about private data. Standard information flow properties are often too restrictive and, consequently, quite useless in real applications: in practice, some information about secret data should be allowed to flow. Several efforts have been made to formalize weaker and more flexible properties. Abstract Non-Interference (ANI) [6] uses Abstract Interpretation [3] to parameterize information flows with respect to a degree of precision: untrusted observers are supposed to see only an abstract representation of public data and flows can exist as long as observers are not able to see them. ANI can describe a number of security properties by simply choosing suitable abstract domains which correctly represent the precision of attackers in observing data.

Abstract Non-Interference was originally defined as a theoretical, non-algorithmic framework; in general, even describing abstract domains can be undecidable. This paper studies a practical way to enforce ANI in the real, widely-used framework of Java. Non-Interference does not refer to single executions of programs; rather, it deals with the differences that two or more computations may have. Consequently, we are interested in pairs of values: a pair may contain the values of a variable in any two computations whose initial states are distinguishable only in their private part, i.e. indistinguishable by attackers. Abstract values can then be stated as abstractions of pairs of values, identifying the set of all the possible values for data. For example, standard NI holds if, when having two executions with the same input value for public data, public variables can only have output values of the form $(v, v)$ for any $v$, meaning that the value of public entities must be the same (although not known). In ANI analysis we follow the same pattern with abstract values: a pair $(s_1, s_2)$ means that a variable has, in any two executions, a value contained resp. into $s_1$ and $s_2$. An abstract value is *public* if attackers cannot distinguish the two components of any pair. A program is secure if the final abstract state contains only public abstract values (sets of symmetric pairs $(s, s)$, where $s$ belongs to the abstract domain defining the precision of the attacker).

### 1.1 Related work

A comprehensive survey of bytecode verification is in [9]. The problem of (standard) information flow for Java bytecode has been recently addressed in [5] and [1] by using, respectively, boolean functions and types. The analysis outlined in this paper is similar to security type inference [12]; however, it is not limited to syntactical analysis. Abstract Non-Interference has been defined for higher-order functional languages [13]. A notion of security type is used, which is similar to the abstract values of this paper. The analysis for ANI uses a notion of *ternary relation* similar to [8]; this allows deciding properties displayed by pairs of computations instead of considering single executions.

## 2. PRELIMINARIES: ANI

Due to space limitations, introduction to the well-known theories of Non-Interference [7] and Abstract Interpretation [3, 4] is omitted. The reader is also supposed to know the basics of bytecode semantics and control flow analysis.

The program property of Non-Interference can be weakened (abstracted) by modeling secrecy relatively to some observable property: a secure program is one that preserves secrecy only as regards of a particular amount on information, the one which attackers can observe.

*Example 1.* In the program `while (h>0){ l+=2; h--;}`, an attacker can guess, by observing the output for the public variable `l`, the value of `h` (private): there is a dangerous information flow from private to public data. However, it is often the case that the attacker can see public data only partially: e.g. if it cannot see anything but parity, then it cannot exploit the flow of information since the parity of `l` is not affected by `h` (there is no abstract flow from `h` to `l`).

The concrete domain of computations is the set of all properties (e.g. $\wp(\mathbb{N})$ for integers: a property is described by the set $P \subseteq \mathbb{N}$ of the values satisfying it). The ability of attackers in observing data is described by abstract domains: $\rho$ is unable to distinguish two values $v_1$ and $v_2$ such that $\rho(\{v_1\}) = \rho(\{v_2\})$. A program $P$ is secure for the pair $(\eta, \rho)$ (written $[\eta]P(\rho)$) if no information flows are detected by an attacker which can see public input and output data up to a level of precision characterized resp. by the domains $\eta$ and $\rho$:

$$\eta(l_1) = \eta(l_2) \quad \Rightarrow \quad \rho([\![P]\!](h_1, l_1)_L) = \rho([\![P]\!](h_2, l_2)_L)$$

if the input values $l_1$ and $l_2$ cannot be distinguished, then it is not possibile to guess the value of $h_i$ by observing the output. For a given $\eta$, the *secret kernel* is the most concrete $\rho_s$ such that $[\eta]P(\rho_s)$, i.e. the most powerful attacker for which program secrets are not revealed. ANI can describe several kinds of attackers by using suitable abstract domains; standard NI is the special case $[\lambda x.x]P(\lambda x.x)$.

# 3. THE ALGORITHM

The algorithm for checking ANI can be seen either as a type system or an abstract semantics. Concrete entities taking part to an execution, such as values, operand stacks and environments, have their abstract counterpart.

## 3.1 Abstract domains

Abstract domains represent what can be seen of data. An attacker can see an approximation of public data and nothing of private data. A *security policy* is a description, in terms of abstract domains, of the observing power of attackers; it specifies the degree of precision a program is asked to be secure for. For example, the policy for standard Non-Interference is that *nothing* can be inferred about private data by an attacker which is able to see *all* public information. In ANI, the attacker can see only *something* of public input and *something* of public output. Abstract domains describe these *something*.

Our analysis works on a single domain $\rho$ which should contain all the information about the policy; $\rho$ is obtained by computing the reduced product of all the domains; in addition, in order to be more precise in computing boolean guards, a further abstract domain $\rho_z$ modeling *zero* is added:

$$\rho_z(\{0\}) = \{0\} \qquad \rho_z(X) = \text{if } 0 \notin X \text{ then } \mathbb{N} \setminus \{0\} \text{ else } \mathbb{N}$$

In the following, $\rho$ will be the domain collecting all the information about the security policy.

## 3.2 Abstract computations

Information flow properties refer to compared executions. We are interested in abstracting pairs of values: the values that a program entity can possibly have in two different computations. Saying that the variable $x$ has (concrete) value $(4, 5)$ means that it has value 4 in the first execution and 5 in the second one.

*Definition 1.* A *prevalue* $F$ is a function mapping variables into security pairs, i.e. elements of $\rho \times \rho$. Ordering, lub $\sqcup$ and glb $\sqcap$ are defined pointwise. A prevalue can be given a set representation as $F = \{x_{s_1 s_2}, y_{t_1 t_2} \dots\}$, meaning that $F(x) = (s_1, s_2)$, $F(y) = (t_1, t_2)$ and $F(z) = (\top, \top)$ for any variable $z$ not occurring in the set; then $\emptyset$ is the trivial prevalue $\lambda x.\top$. An *abstract value* $V \in \mathbb{V}$ is a set of security pairs; it has the form $V = \{(s_1, s_2)^F\}$, where $s_1, s_2 \in \rho$ and $F$ is a prevalue.

Therefore, abstract computations are performed on abstract values in order to infer approximate properties. Abstract values as defined above can be seen as types which are assigned to variables. The correspondence between types and abstract values was outlined in [2]: types are representations of sets of values (e.g. `int` for integer numbers) and can be the codomain of some uco.

Prevalues provide additional information, allowing to improve the analysis process by *remembering* the value of a variable at the beginning of a block; this can be useful particularly in analyzing loops, as shown in Sec. 4. Let $x$ have abstract value $V$; $(s_1, s_2)^F \in V$ means that there exist two computations such that the value of $x$ is represented by $(s_1, s_2)$ when, at the beginning of the block, variable values (not only $x$) are described by $F$. Ordering on $\mathbb{V}$ is $V_1 \leq V_2 \equiv \gamma(V_1) \subseteq \gamma(V_2)$ where $\gamma(V)$ is the set of pairs of values which are represented by $V$: a greater value is a description of a larger set of pairs.

An abstract value is *public* if it contains only pairs $(s, s)^F$, where $s$ is an element of $\rho$ (otherwise it is private): if a variable has a public value, then an attacker cannot distinguish the two computations. In facts, since $\rho$ is obtained by reduced product, it is at least as precise as the observational power of the attacker on output; then flows which are not detected by $\rho$ will not be seen by any observer.

## 3.3 Public and private data

When an abstract state is *public*, an attacker cannot distinguish between any two concrete states which are represented by it. At the beginning of an abstract computation, initial values encode the requirement that public variables are not distinguishable, while private values can have any value; the value of a variable depends on no other variables. Then $x$ is given the abstract value $\{(s_1, s_2)^{x_{s_1 s_2}} \mid s_1, s_2 \in \rho\}$ if it is private, $\{(s_1, s_2)^{x_{s_1 s_2}} \mid \exists s \in \rho_x^{in}.s_1, s_2 \subseteq s\}$ if public (where $\rho_x^{in}$ is the input precision on $x$): the abstract value of public variables is equal (by $\rho_x^{in}$) in any two computations (but can be distinguished by $\rho$, which is more precise) while no constraint is applied on a value in a single execution.

## 3.4 Analysis of basic blocks

In basic blocks (i.e. a sequence of linear instructions) control flow is fixed and the state can be modified by pushing/popping values on the stack or by writing values into locations. Block analysis builds a final abstract state $\langle \mathbf{j}.\underline{S'}.E' \rangle$ out of an initial state $\langle \mathbf{i}.\underline{S}.E \rangle$. Dangerous information flows arise when $E'$ is non-public even if $E$ is. Rules for linear instructions are shown in Figure 1; they are simply the abstract translation of the concrete semantics. $\mathrm{G}_{\mathrm{out}}(bl, G)$ is

$$
\begin{array}{rcl}
[\![\texttt{push } n]\!]\langle \mathbf{i}.\underline{S}.E\rangle & = & \langle \mathbf{i}+\mathbf{1}.\{\rho(n),\rho(n)\}^{\emptyset}|S.E\rangle \\
[\![\texttt{prim } \otimes]\!]\langle \mathbf{i}.\underline{V'|V''|S}.E\rangle & = & \langle \mathbf{i}+\mathbf{1}.\underline{(V'\otimes V'')|S}.E\rangle \\
[\![\texttt{pop}]\!]\langle \mathbf{i}.\underline{V|S}.E\rangle & = & \langle \mathbf{i}+\mathbf{1}.\underline{S}.E\rangle \\
[\![\texttt{load } x]\!]\langle \mathbf{i}.\underline{S}.E\rangle & = & \langle \mathbf{i}+\mathbf{1}.\underline{E(x)|S}.E\rangle \\
[\![\texttt{store } x]\!]\langle \mathbf{i}.\underline{V|S}.E\rangle & = & \langle \mathbf{i}+\mathbf{1}.\underline{S}.E[x\leftarrow V]\rangle
\end{array}
$$

**Figure 1: Rules for linear instructions**

defined as the state at program point $n+1$ (where $n$ is the length of the block) of $bl$ when the initial state is $G$: $\mathrm{G}_{\mathrm{out}}(bl,\langle \mathbf{i}.\underline{S}.E\rangle) = \langle \mathbf{i}+\mathbf{n}.\underline{S'}.E'\rangle$. When performing an integer binary operation $\otimes$ on two values on top of the operand stack, the value $V'\otimes V''$ of the result is a function of the values $V'$ and $V''$ of the operands:

$$
\begin{array}{c}
V'\otimes V'' \;=\; \bigcup_{v'\in V',v''\in V''}\{v'\otimes v''\} \\
(s'_1,s'_2)^{F'}\otimes(s''_1,s''_2)^{F''} \;=\; (s'_1\otimes s''_1, s'_2\otimes s''_2)^{F'\sqcap F''} \\
s'\otimes s'' \;=\; \alpha(\{v\otimes u \mid v\in\gamma(s'), u\in\gamma(s'')\})
\end{array}
$$

where $F\sqcap G$ is the greatest lower bound of $F$ and $G$, meaning that the result value is valid when both hold.

Since our concrete domain (the integer numbers) is infinite, an abstract domain on it must either contain an infinite number of elements or have some elements representing an infinite set of numbers. In the first case some operations are impossible since, for example, the abstract value for a variable would not have a finite representation. In the second, operations on abstract values should deal with infinite sets (for example, the set of even numbers), then computing $s\otimes t$ can be impractical. Some domain-dependent rules can be provided, specifying the results of abstract computations (for example, *even plus odd equals odd* for the parity domain).

## 3.5 Analysis of regions

A program with jumps cannot be described by a single block; instead, it is representable as a graph of basic blocks. Here we call *region* a subgraph of the CFG of a program. Region analysis is compositional: the function REG computes the abstract final state for a region, given a state $G\in\mathbb{G}$:

$$\mathrm{REG}:\mathbb{R}\times\mathbb{G}\mapsto\mathbb{G}$$

- $\mathrm{REG}(bl,G) = \mathrm{G}_{\mathrm{out}}(bl,G)$: a block is analyzed as a sequence of instructions.
- $\mathrm{REG}((\pi_1;\pi_2),G) = \mathrm{REG}(\pi_2,(\mathrm{REG}(\pi_1,G)))$: the state after executing two sequentially composed regions is obtained by functional composition.
- The analysis of a conditional region is more interesting:

$$\mathrm{REG}(\mathrm{if}(\pi_g,\pi_1,\pi_2),G) = \mathrm{M}(\mathrm{BG}(\mathrm{OV}(\pi_g,G)),\mathrm{REG}(\pi_1,G),\mathrm{REG}(\pi_2,G))$$

The states $\mathrm{REG}(\pi_1,G)$ and $\mathrm{REG}(\pi_2,G)$ are those obtained by analyzing the two branches of the conditional statement. These states are merged in the sense that static analysis cannot, in general, decide which branch will be executed. Merging two environments involves the pointwise merging of abstract values:

$$\mathrm{M}(B,E_1,E_2) = \lambda x.\ \mathrm{M}(B,E_1(x),E_2(x))$$

and the merging of values is defined as:

$$
\begin{array}{c}
\mathrm{M}(B,V',V'') = \bigcup\{\mathrm{M}(B,p',p'') \mid p'\in V', p''\in V''\} \\
\mathrm{M}(B,(s'_1,s'_2)^{F'},(s''_1,s''_2)^{F''}) = \{(s_1,s_2)^{F_0} \mid (b_1,b_2,F)\in B\}
\end{array}
$$

where $s_i = (b_i\ ?\ s'_i\ ;\ s''_i)$ and $F_0 = F\sqcap(\mathrm{M}(b_1,b_2,F',F''))$

$$
\begin{array}{rl}
\mathrm{M}(b_1,b_2,F',F'') = \lambda x. & ((b_1\ ?\ \mathrm{fst}(F'(x))\ ;\ \mathrm{fst}(F''(x))), \\
& (b_2\ ?\ \mathrm{snd}(F'(x))\ ;\ \mathrm{snd}(F''(x))))
\end{array}
$$

Merging depends on the abstract value of the boolean guard,

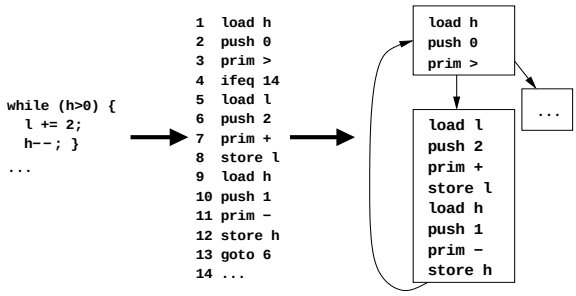


**Figure 2: The program and its transformations**

describing the possible configurations of concrete guards in the two computations (for example $\{(\mathrm{T},\mathrm{T}),(\mathrm{F},\mathrm{F})\}$ means that the value of the guard is equal but unknown):

$$
\begin{array}{c}
\mathrm{BG}(V) = \bigcup_{(s_1,s_2)^X\in V}\{(b_1,b_2,X) \mid b_1\in\mathrm{BV}(s_1), b_2\in\mathrm{BV}(s_2)\} \\
\mathrm{BV}(\{0\}) = \{\mathrm{T}\} \qquad \mathrm{BV}(s\not\supseteq\{0\}) = \{\mathrm{F}\} \\
\mathrm{BV}(s\supsetneq\{0\}) = \{\mathrm{T},\mathrm{F}\}
\end{array}
$$

$\mathrm{OV}(\pi,G)$ is the abstract value appearing on top of the stack after the analysis of $\pi$ in $G$. After merging, the resulting abstract state is one with, as top-environment, the merging of the two environments.

- $\mathrm{REG}(\mathrm{loop}(\pi_g,\pi_l),G) = \text{W-M}(\mathrm{BG}(\mathrm{OV}(\pi_g,G)),\pi_l,G)$. Conceptually, loops can be translated into infinite sequences of conditional statements. The informal expression

$$\mathrm{M}(\mathrm{BG}(\mathrm{OV}(\pi_g,G)),\ G,\ \mathrm{REG}(\pi_l,G),\ \mathrm{REG}(\pi_l;\pi_l,G),\dots)$$

means that any number of iterations can be performed and this number can be different in the two computations when the guard is private. Computing REG for a loop involves a fixpoint computation, encoded in W-M. Given a limit number $K$: if, after at most $k\le K$ steps, the variable values are *smaller* or equal to those at iteration $k-1$, then monotonicity of the operators on abstract values guarantees that any other step will not yield greater results. If the fixpoint is not reached within $K$ steps, private values are taken as the result (a widening operator is applied).

# 4. AN EXAMPLE

In this section, the analysis of a small piece of code is shown, allowing to see some important characteristics of the algorithm. The example program and its translation into the control flow graph are shown in Figure 2; `h` is a (non-negative) private variable, while `l` is public (non-negative). The program is clearly noninterfering if the security policy is encoded into the parity domain: the value of `h` affects the final value of `l` but leaves unchanged its parity, then an attacker which is only able to see the parity of public data cannot infer how many iterations of the loop were executed. It can be easily seen that this program has not the standard Non-Interference property, since an information flow exists between the two variables. More formally: $\rho$ is the reduced product of the domains on input and output; since the parity domain $\rho_p$ describes both input and output, we have $\rho = \rho_p\prod\rho_p\prod\rho_z = \rho_p\prod\rho_z$ which contains the elements $e$ (even), $o$ (odd) and $z$ (zero). $kk$ stands for $(k,k)$. The loop has a private guard (it depends on the value of `h`), then, in principle, the number of iterations could leak information about the private variable by reading `l`. However, the parity of `l`, although unknown, is not affected by the number of iterations (it is an invariant of the loop); we want to take

advantage of this fact, making the analysis smart enough to detect some form of invariance.

After analyzing the guard, a private abstract value is on top of the stack: $V_g = \{zz, ze, zo, ez, ee, eo, oz, oe, oo\}$ (prevalues are omitted) means that the values of the guard can be, in both computations, anything among *even*, *odd* and *zero*. The application of BG to $V_g$ yields $\{tt, ff, tf, ft\}$. Computing the abstract value for `l` gives ($\pi$ = loop body):

$$\begin{array}{rcl} \text{initial state} & : & G = \{(zz)^{l_{zz}}, (ee)^{l_{ee}}, (oo)^{l_{oo}}\} \\ \text{after any iteration} & : & \{(ee)^{l_{zz}}, (ee)^{l_{ee}}, (oo)^{l_{oo}}\} \\ \text{W-M}_1(b, \pi, G) & = & \{(ee)^{l_{zz}}, (ez)^{l_{zz}}, (ze)^{l_{zz}}, \\ & & (zz)^{l_{zz}}, (ee)^{l_{ee}}, (oo)^{l_{oo}}\} \\ \text{W-M}_2(b, \pi, G) & = & \{(ee)^{l_{zz}}, (ez)^{l_{zz}}, (ze)^{l_{zz}}, \\ & & (zz)^{l_{zz}}, (ee)^{l_{ee}}, (oo)^{l_{oo}}\} \end{array}$$

Since we have no knowledge of the actual path taken by possible computations, we should consider the possibility that, for instance, the number of loop iterations is positive in the first computation and zero in the second. Here, prevalues come into play: in *crossing* pairs from the two branches we consider pairs with compatible prevalues: for example, $(ee)^{l_{ee}}$ will be only merged with $(ee)^{l_{ee}}$ and not with $(oo)^{l_{oo}}$ because their prevalues are not consistent. Therefore, the fixpoint of W-M is immediately found and the final abstract value for `l` is

$$\{(ee)^{l_{zz}}, (ez)^{l_{zz}}, (ze)^{l_{zz}}, (zz)^{l_{zz}}, (ee)^{l_{ee}}, (oo)^{l_{oo}}\}$$

This corresponds to say that, by only knowing that the initial parity of `l` is equal in both computations, it is possible to rule out many unreachable final states, thus obtaining ANI. In this case prevalues are useful, allowing to be more precise than simply merging the final values of the two branches. Obviously, this is not a complete method, many situations remaining where false flows are detected in a secure program.

# 5. CORRECTNESS OF THE ALGORITHM

The analysis of a program starts in a public state (see Sec. 3.3); it yields, eventually, an abstract state containing the abstract values for all variables; the program is said to be *secure* if the final state is also public.

## 5.1 Basic blocks

In this section we want to prove that, when a block is analyzed starting from a public state, the final state is also public whenever the program satisfies ANI. In order to formalize publicity, SV (*same value*) predicate is defined; it characterizes pairs of concrete entities both representable by an abstract entity.

*Definition 2.* $x_1$ and $x_2$ are SV-related to $X$ if they both belong to its concretization:

$$\mathrm{SV}_X(x_1, x_2) \equiv (x_1, x_2) \in \gamma(X)$$

This definition can be applied to values (so that $\mathrm{SV}_V(v_1, v_2)$ holds if the concrete values are represented by the same security type $V$), stacks (pointwise) and environments (for which only the public part has to be considered) by replacing the correct definition of representability.

The following lemma (whose proof is quite straightforward) claims that, whenever a block is executed in two indistinguishable states, the final states are still indistinguishable.

LEMMA 1. *Let $l$ be linear, $[\![l]\!]\langle \mathbf{j}.\underline{\sigma_i}.\varepsilon_i\rangle = \langle \mathbf{j}+\mathbf{1}.\underline{\sigma_i'}.\varepsilon_i'\rangle$.*

$$[\![l]\!]\langle \mathbf{j}.\underline{S}.E\rangle = \langle \mathbf{j}+\mathbf{1}.\underline{S'}.E'\rangle \ \wedge\ \mathrm{SV}_S(\sigma_1, \sigma_2) \ \wedge\ \mathrm{SV}_E(\varepsilon_1, \varepsilon_2) \implies \mathrm{SV}_{S'}(\sigma_1', \sigma_2') \quad \wedge \quad \mathrm{SV}_{E'}(\varepsilon_1', \varepsilon_2')$$

Only direct information flows may arise inside a block; this is the case when private abstract values are pushed on the operand stack, possibly manipulated by arithmetic operators and, eventually, stored into the public part of the environment. This corresponds, in Java source code, to assigning to a public variable an expression depending on private data.

## 5.2 Regions

LEMMA 2. *Let $\pi$ be the CFG of a program $P$ satisfying*

$$[\![P]\!]\langle \mathbf{1}.\underline{\sigma_1}.\varepsilon_1\rangle = \langle \mathbf{j}.\underline{\sigma_1'}.\varepsilon_1'\rangle \quad \wedge \quad [\![P]\!]\langle \mathbf{1}.\underline{\sigma_2}.\varepsilon_2\rangle = \langle \mathbf{j}.\underline{\sigma_2'}.\varepsilon_2'\rangle$$

*where $\exists G = \langle \mathbf{1}.\underline{S}.E\rangle$. $\mathrm{SV}_S(\sigma_1, \sigma_2) \ \wedge\ \mathrm{SV}_E(\varepsilon_1, \varepsilon_2)$. Then*

$$\mathrm{SV}_{S'}(\sigma_1', \sigma_2') \quad \wedge \quad \mathrm{SV}_{E'}(\varepsilon_1', \varepsilon_2')$$

*where $\langle \mathbf{j}.\underline{S'}.E'\rangle = \mathrm{REG}(\pi, G)$.*

This lemma can be also stated as *if the initial (abstract) state for a program satisfies* SV *(is public) for a pair of concrete executions, then the final (abstract) state also satisfies* SV *for the two final concrete states.* We proved that the abstract state computed by the algorithm is in SV relation with (the state after) any pair of concrete computations. This means that, if the inferred abstract state has public values for all public variables, then the program is secure.

# 6. ADVANCED FEATURES

## 6.1 Procedures

Static class methods (i.e. procedures whose definition refers to a class rather than to a particular object) can be seen as usual procedures or functions in purely imperative programming languages. When a method is called via the instruction `invokestatic` *c.m* a new frame is pushed on the framestack with an empty operand stack and a local environment containing actual parameters and local variables. The actual parameters are put in the environment after popping them from the operand stack of the caller; when the method terminates, the current top-value of the operand stack is pushed on the stack of the caller, unless the method has `void` return type. The abstract counterpart for this mechanism is ($m$ has `int` return type):

$$[\![\texttt{invokestatic}\ c.m]\!]\frac{\langle \mathbf{i}.V|S.E\rangle}{\phi} = \frac{\langle \mathbf{1}.\underline{S'}.E'\rangle}{\frac{\langle \mathbf{i}.\underline{S}.E\rangle}{\phi}}$$

$$[\![\texttt{return}]\!]\frac{\langle \mathbf{j}'.V'|S'.E'\rangle}{\frac{\langle \mathbf{i}.\underline{S}.E\rangle}{\phi}} = \frac{\langle \mathbf{i}+\mathbf{1}.V'|S.E\rangle}{\phi}$$

Interference is present when either private information is put into a public non-local variable during method execution, or the (private) result of the method is dangerously assigned to a public variable in the code of the caller.

## 6.2 Classes and objects

Common theoretical definitions of Non-Interference consider a simplified notion of program execution in which memory is static and variables can be created only at the beginning of the program. States can change only by variable assignment; two states are indistinguishable if all variables have indistinguishable values. On the other hand, objects involve dynamic memory allocation, so that state indistinguishablity must take into account the possibility that different parts of memory are allocated.

*Remark 1.* What should be intended for interference depends on which properties we are interested in. In defining a

security policy it is not enough to ask "how much an attacker can see of the variable $x$?", because there is not a fixed set of variables during execution. For example, if an attacker is able to see the amount of free memory, then he could acquire private information if an object creation depends on a private variable, as in the following code fragment:

```
while (h>=0) { a[rnd]=new C(); h--; }
```

where `rnd` is a random integer value; in this case information does not flow into a public variable, but the value of `h` affects the state of the memory.

A reasonable choice in defining the policy uses Java modificators and has class granularity: *a class has ANI property if the execution of its (possibly `static`) methods does not reveal anything about `private` (possibly `static`) class components to an external attacker with a given degree of precision.* Thus, an attacker should not acquire information on private class members or personal data of any object of the class. This characterization does not answer to the above remark: it should be specified whether the attacker is able to see memory properties other than variable values. This is, however, a matter of abstraction and can be formalized, in principle, with Abstract Interpretation.

Since a Java program is in facts a collection of class declarations, this definition guarantees ANI for the whole program if every class is secure; analysis should consider mutual dependencies between class component.

### 6.3 Exceptions

Exceptions can be included in ANI analysis by considering them as calls to particular methods executed conditionally on a boolean guard which captures the arising of some semantic violation. For example, the access to an array element `x=a[h]` can be a source of dangerous information flow, even if `x` is private, because the throwing of an `ArrayOutofBounds` exception gives information about `h` (namely, that its value is not within array bounds). This is equivalent to the following pseudocode

*if* $0 \leq h < length(a)$ *then* $x \leftarrow a[h]$ *else manage_exception*

which can be easily translated into bytecode and checked with the usual type analysis, *manage_exception* being a call to the code handling the exception. This can be applied either to implicit or explicit (managed with the `try ... catch` statement) exceptions.

## 7. CONCLUSIONS AND FUTURE WORK

This analysis attempts to make automatic the checking of non-trivial information flow properties. Like any static mechanism for deciding semantic properties, it is necessarily incomplete, since secure programs can be rejected as dangerous because of deceptive flows. Nevertheless, it is possible to rule out some false cases of interference (such as `l=h; l=0`) which are problematic in compositional type systems (such a program is rejected because of a dangerous subprogram). Abstract Non-Interference uses abstract domains on infinite sets, such as the integer numbers; then it is not surprising that checking for ANI can be undecidable, since operations on abstract values (such as $V' \otimes V''$) possibly involve infinite sets (Sec. 3.4).

A more realistic approximation of bytecode could be considered, with a larger set of instructions and Object-Oriented mechanisms; ANI analysis should be linked to the security mechanisms of Java for data hiding (e.g. the modifiers `private` and `public`). A complete treatement of object would involve the redefinition of Non-Interference and many different design choices; this is an important research direction. ANI was proposed in a purely theoretical flavour, relying on describing the observing power of attackers by means of abstract domains. Yet, which relaxed information flow properties are really useful in program analysis is a matter of research: realistic domains should be provided in order to encode critical security policies. Bytecode has the nice property to be directly executed on the client machine; yet, it is higher-level than machine code. Since our analysis works on the same language which will be used by the client, there is no need of a compiler which is guaranteed to preserve the security properties of the source program. This fact is relevant when considering Proof-Carrying code [11] as an application of ANI analysis. A user shall trust and execute a program only if type checking yields a public result; proofs of *ad hoc* rules for operations on abstract values shall be provided by the code producer together with the program.

## 8. REFERENCES


[1] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In *Proc. ACM TLDI*. ACM Press, 2005.

[2] P. Cousot. Types as abstract interpretations, invited paper. In *Proc. ACM POPL*. ACM Press, 1997.

[3] 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 POPL*. ACM Press, 1977.

[4] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proc. ACM POPL*. ACM Press, 1979.

[5] S. Genaim and F. Spoto. Information Flow Analysis for Java Bytecode. In *Proc. VMCAI*, volume 3385 of *LNCS*. SV, 2005.

[6] R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. In *Proc. ACM POPL*. ACM Press, 2004.

[7] J. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*. IEEE Computer Society Press, 1982.

[8] S. Hunt. *Abstract Interpretation of Functional Languages: From Theory to Practice*. PhD thesis, Dept. of Computing, Imperial College of Science Technology and Medicine, 1991.

[9] X. Leroy. Java bytecode verification: an overview. In *Proc. CAV*, volume 2102 of *LNCS*. SV, 2001.

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

[11] G. Necula. Proof-Carrying Code. In *Proc. ACM POPL*. ACM Press, 1997.

[12] D. Volpano and G. Smith. A type-based approach to program security. In *Proc. TAPSOFT'97*, volume 1214 of *LNCS*. SV, 1997.

[13] D. Zanardini. Higher-Order Abstract Non-Interference. In *Proc. TLCA*, LNCS. SV, 2005.