The Non-Interference Protection in a Bytecode Program Logic∗

Daria Walukiewicz-Chrząszcz and Aleksy Schubert

Institute of Informatics
University of Warsaw
ul. Banacha 2
02-097 Warsaw
Poland

Abstract Many information-flow type systems have been developed that allow to control the non-interference of information between the levels of classification in the Bell-LaPadula model. We present here a translation of typing information collected for bytecode programs to a bytecode program logic. This translation uses the syntax of a bytecode specification language BML. A translation of this kind allows including the check of the non-interference property in a single, unified verification framework based on a program logic and thus can be exploited within a foundational proof-carrying code infrastructure. It also provides a flexible basis for various declassification strategies that may be useful in a particular code body.

1 Introduction

The application of formal specification methods at the level of the Java bytecode has several advantages. (1) This allows to provide descriptions and verify properties of programs written in the bytecode itself. (2) It allows to do a unified formalised development for languages other than Java, but compiling to the Java bytecode. In particular, it allows to conduct a unified formal verification in projects with several source code languages. (3) Proofs for bytecode programs may enable several optimisations in JIT compilers. (4) As bytecode is the language which is actually executed, it is possible to couple with programs their proof carrying-code (PCC) certificates. (5) Since Java programs are distributed in their bytecode version, it is possible for a software distributor to develop its own certificate to ensure a particular property its clients are interested in. These reasons led to a proposal of a bytecode program logic [BH06] and, based on this foundation, a specification language for the bytecode — Bytecode Modeling Language (BML) [BHP07]. The latter language is based on the design-by-contract principles and is derived from a Java specification language called Java Modeling Language [JP01,LBR98,LPC+05] which has wide tool support [BCC+03].

∗ This work was partly supported by Polish government grant 177/6.PR UE/2006/7 and Polish government grant 3 T11C 002 27
A software verification effort can usually be justified when the software should give strong software security guarantees. In this context the application of frameworks based on the Bell-LaPadula model [BL75] is standard practice. The model is the starting point for many information-flow type systems for while-programs (e.g. [HS06,SM03,HRLS06]) as well as for other formal languages (e.g. [HY05,Hen04]). The aim of the type systems is to establish the non-interference property i.e. the property that one cannot deduce facts about more confidential information based on the less confidential one.

In this paper we consider the non-interference property of bytecode programs. As a starting point we take an information-flow type system for Java bytecode [BRN06,BPR06] that guarantees non-interference for sequential bytecode programs with objects, methods and exceptions. The main contribution of the current paper is a translation of the type system into the bytecode program logic developed within the MOBIUS project [BH06,Con06] such that the correctness of the typing is equivalent to the verifiability of the resulting bytecode program logic formulas. This translation goes through the BML specification language [BCC03] which has a direct translation to the aforementioned program logic. This solution has the advantage that the translation can be easily adopted by verification tools based on the logic and further explored in a proof-carrying code infrastructure (e.g. the one envisioned in the MOBIUS project, see [Con06]).

It is worth pointing out that the translation has a few desirable features. First of all, it is possible to use a toolset based on logical methods rather than typed ones. The translation may be especially profitable when the single expressive logic based toolset is within the trusted code base. Second, the resulting model of non-interference is more flexible when the non-interference property must be weakened, because the declassification is needed (e.g. in case the code encrypts confidential data). The translation we provide is designed so that it is relatively straightforward to adjust it to declassification needs.

The paper is structured as follows. In Section 2 we fix the notation and present the basic notions which are exploited in the paper. This is followed in Section 3 by an exposition of the translation of the type based system to the logic based one. This translation is supplemented by a proof that the resulting specifications guarantee the non-interference property in Section 4. The formal development is concluded by a proof that the non-interference property holds even when the bytecode program is extended with other specifications. This is presented in Section 4.1. At last we present the final remarks in Section 5 where we sketch the way the declassification can be modelled.

2 Preliminaries

In this section we fix the notation used throughout the paper. We, generally, follow here the papers [Con06,BPR06]. Sometimes, we give here additional informal descriptions of the notions which are not directly used in the translation, but are essential for understanding the principles of the construction.
Basic notation We use the expression \( \text{dom}(f) \) to denote the domain of the (partial) function \( f \). Similarly, \( \text{rng}(f) \) denotes the image of \( f \). To indicate that \( f \) is a partial function from a set \( A \) to \( B \) we write \( f : A \rightarrow B \). The powerset of a set \( A \) is written \( P(A) \). We write \( \textbf{k} \) to denote a vector of values. The notation \( |\textbf{k}| \) expresses the length of the vector and \( \textbf{k}(0), \ldots, \textbf{k}(|\textbf{k}| - 1) \) are subsequent elements of the vector.

Java bytecode programs and specifications A Java bytecode program \( P \) is a set of classes with one singled out method \( \text{main}_P \). A class \( C \) is a set of fields and methods. Each field \( f \) has a name \( f_n \) and a type \( f_t \). Similarly, a method \( m \) has a name \( N(m) \), a signature \( T_m \) and the body \( B_m \). We assume that the method names are unique within a single program (possibly due to the standard Java prefixing with an object or class name). A method body is a sequence of bytecode instructions. The instructions are indexed by program points. For each method \( m \) we distinguish the set of all program points in the method \( \mathcal{P} \mathcal{P}_m \) (we omit the subscript \( m \) when \( m \) is clear from the context).

An annotated program \( \hat{P} \) has additionally for each class \( C \)

- a list \( \text{Ghost}_C \) of model and ghost fields (i.e. fields which can occur only in specifications),
- a list \( \text{Inv}_C \) of (static and object) class invariants,
- a list \( \text{Constr}_C \) of (static and object) history constraints,
- a method specification table \( \mathcal{M}_C \).

The method specification table \( \mathcal{M}_C \) associates with each method \( m \):

- a method specification \( S_m = (R_m, T_m, \Phi_m) \) where \( R_m \) is the precondition of the method \( m \), \( T_m \) is the postcondition of the method, and \( \Phi_m \) is the method invariant;
- a local specification table \( G_m \) which assigns to each label in the method body \( B_m \) an additional assumption that may be used in the proof of the program verification clause associated with the label;
- a local annotation table \( Q_m \) which assigns to each label in the method body \( B_m \) an additional assertion,
- a local instruction table \( \text{Ins}_m \) which assigns to each label \( l \) in the method body \( B_m \) a sequence of bytecode instructions that operate on ghost variables which is supposed to be executed before the instruction at the label \( l \) and the respective specification \( Q_m(l) \).

Security policy The security policy framework we employ in this paper is based on assumption that the attacker can observe the input/output of methods only (e.g. methods that perform network operations). This, however, is extended to the values of fields and heaps as otherwise it is difficult to guarantee statically the non-interference property. We also assume that the attacker is unable to observe the termination of the programs.

\footnote{The separation of the identities for the method and its name serves to model the inheritance.}
Formally, a security policy is expressed in terms of a finite partial order ($S, \leq$). This order allows to describe the capabilities of the attacker and the program to be analysed:

- A security level $k_{\text{obs}}$ determines the observational capabilities of the attacker (she can observe fields, local variables and return values whose level is less or equal than $k_{\text{obs}}$).
- A policy function $f_t$ assigns to each field its security level. This allows us to express the non-interference property we are interested in.
- A policy function $\Gamma$ that associates to each method identifier $N(m)$ and security level $k \in S$ a security signature $\Gamma_{N(m)}[k]^2$. This signature gives the security policy of the method $m$ called on an object of the level $k$. The set of security signatures for a method $m$ is defined as $\text{Policies}_f(m) = \{ \Gamma_{N(m)}[k] \mid k \in S \}$. The security signature has the shape $k_p \xrightarrow{k_h} k_r$:
  - The vector $k_p$ describes the security levels appropriate for the local variables of the method (in particular it assigns also the levels to the input parameters), $k_p[0]$ is the upper bound on the security level of an object that calls the method.
  - The value $k_h$ describes the lower bound in the security levels of the heap operations performed by the method.
  - The vector $k_r$ describes the security levels for the method results (both normal and exceptional ones); it is a list of the form $\{ n : k_n, e_1 : k_{e_1}, \ldots, e_n : k_{e_n} \}$, where $k_n$ is the security level of the return value and $e_i$ is the class of an exception that might be thrown in the method and $k_{e_i}$ is the upper bound on the security level of the exception. We use the notation $k_r[n]$ and $k_r[e_i]$ for $k_n$ and $k_{e_i}$.

For simplicity, the orders we actually use here are suborders of natural numbers. However, it is straightforward to extend the definitions to arbitrary finite orders. This can be achieved either by a more complicated definition of the inequality $\leq$ we use in the formulas below or by using ghost objects for which the inequality can be encapsulated in one of their methods. This last solution allows also to consider certain infinite orders.

Non-interference In order to define the non-interference property one has to provide Java Virtual Machine semantics and some notions of indistinguishability with respect to the observer $k_{\text{obs}}$. This is all done in [BPR06]. Moreover in [BPR06] there is a definition stating when a method is safe with respect to a signature. For our needs it is enough to recall the definition of a safe program.

**Definition 1.** (safe programs)
A program $P$ is safe with respect to a table of method signatures $\Gamma$ if for each method $m$ in $P$, $m$ is safe with respect to all the security signatures in $\text{Policies}_f(m)$.

\(^2\) In [BPR06] a less precise notation $\Gamma_m[k]$ is used.
Just to give the insight, the safety with respect to a security signature $k_p \xrightarrow{k_h} k_r$ means that

- the method can write only to variables on the security level greater than $k_h$,
- two invocations of the method such that the parameters at levels less than or equal to $k_{\text{obs}}$ are the same either both return with values of the same level $k \leq k_{\text{obs}}$ and the values are equal, or both return with values of level $\not\leq k_{\text{obs}}$.

**Non-structured programs** The bytecode programs organise the control flow by means of jump instructions. In order to reason on the information flow of such programs an additional structural information is needed. As we translate typings in an information flow system [BPR06], we need the same descriptions of the bytecode program structure.

We use a binary successor relation $\mapsto^\tau \subseteq PP \to PP$ defined on the program points. This relation is parametrised by a tag $\tau$ since bytecode instruction may have several successors as it may execute normally (then the tag is $\emptyset$) or it may trigger an exception (then the tag is the class of the exception). Intuitively, $j$ is a successor of $i$ ($i \mapsto j$) if performing one step execution from a state whose program point is $i$ may lead to a state whose program point is $j$. We write $i \mapsto j$ if $i$ corresponds to a return instruction (or $i \mapsto^\tau$ if $i$ corresponds to an instruction that may throw an exception).

We assume that a bytecode program $P$ comes equipped with a control dependence regions structure cdr which consists of a pair of partial functions $(\text{region}, \text{jun})$. The role of the functions is to arrange the program into compact parts for which the analysis of program invariants can be conducted separately. The region function describes the internal parts of these regions while jun the connections between them. The types of the functions are the following:

$$\text{region}_m : PP \times (\{\emptyset\} + C) \to PP \quad \text{jun}_m : PP \times (\{\emptyset\} + C) \to PP$$

The functions can be axiomatised by the SOAP (Safe Over Approximation) properties [BPR06, Section 6] ensuring that the control dependence regions structure correctly describe information flow in a program $P$.

**Typable programs** To check that a program is safe one may use a type system presented in [BPR06]. In this type system, every method is checked against its signatures separately. The type system is parametrized by:

- a table $\Gamma$ of method signatures,
- a global policy $ft$ that provides security levels of fields,
- a cdr structure $(\text{region}_m, \text{jun}_m)$ for every method $m$

Moreover, we suppose that the following functions are given and that they are correct:

1. A function classAnalysis which given a program point returns the set of exception classes of exceptions that may be thrown at the program point.
2. A function \texttt{excAnalysis} which given a method name \( N(m) \) returns the set of exception classes that are possibly thrown by \( m \).
3. A function \texttt{nbLocals} which given a method name \( N(m) \) returns the number of its local variables.
4. A function \texttt{nbArguments} which given a method name \( N(m) \) returns the number of its arguments.
5. A function \texttt{Handler}_m which for a given point \( i \) in the method \( m \) and an exception \( e \) returns the point where the handler of the exception starts.

**Definition 2.** (typable programs) 
Program \( P \) is \textit{typable} with the policy \((k_{abs}, \texttt{ft}, \Gamma)\) and \texttt{cdr} satisfying SOAP if every method \( m \) from \( P \) is typable with respect to \texttt{ft}, \( \Gamma \), \texttt{region}_m and all signatures in \texttt{Policies}(\( m \)).

**Definition 3.** (typable methods) 
Method \( m \) is \textit{typable} with respect to \texttt{ft}, \( \Gamma \), \texttt{region}_m and a signature \( \texttt{sgn} \) if there exists a security environment \( \texttt{se} : PP \to S \) and a function \( \texttt{st} : PP \to S^* \) such that \( \texttt{st}(0) = \epsilon \) and for all \( i, j \in PP, e \in \{\emptyset\} \cup C \):

1. if \( i \xleftarrow{e} j \) then there exists \( s \in S^* \) such that \( \Gamma, \texttt{ft}, \texttt{region}_m, \texttt{se}, \texttt{sgn}, i \xleftarrow{e} \texttt{st}(i) \implies s \) and \( s \sqsubseteq \texttt{st}(j) \),
2. if \( i \xleftarrow{e} \) then \( \Gamma, \texttt{ft}, \texttt{region}_m, \texttt{se}, \texttt{sgn}, i \xleftarrow{e} \texttt{st}(i) \implies \)

where \( \sqsubseteq \) denote the point-wise partial order on type stack with respect to the partial order taken on security levels. Two exemplary rules for .. \( \xleftarrow{e} \) .. \( \implies \) .. relation, for \texttt{ifeq} and \texttt{invokevirtual}, are given in Figure 2. The set of all rules is given in [BPR06].

\[
\begin{align*}
\text{P}_m[i] = \texttt{ifeq} j \quad \forall j' \in \text{region}(i, \emptyset), \ k \leq \text{se}(j') \\
\Gamma, \text{ft}, \text{region}, \text{se}, k_p \Rightarrow k_r, i \xrightarrow{e} k :: \text{st} \implies \text{lift}_{k}(\text{st}) \\
\Gamma, \text{ft}, \text{region}, \text{se}, k_p \xrightarrow{k_h} k_r, i \xrightarrow{e} k :: \text{st} \implies \text{lift}_{k}(\text{st}) \\
\text{P}_m[i] = \texttt{invokevirtual} N(m') \quad \Gamma_{N(m')}[k] = k'_p \xrightarrow{k'_h} k'_r \\
k \cup k_h \cup \text{se}(i) \leq k'_h \quad \text{length}(\text{st}_1) = \text{nbArguments}(N(m')) \\
k \leq k'_p[0] \quad \forall i \in [0, \text{length}(\text{st}_1) - 1], \ \text{st}_1[i] \leq k'_p[i + 1] \\
k_e = \cup \{k'_r[e] \mid e \in \text{excAnalysis}(N(m'))\} \\
\forall j \in \text{region}(i, \emptyset), \ k \sqsubseteq k_e \leq \text{se}(j) \\
\Gamma, \text{ft}, \text{region}, \text{se}, k_p \xrightarrow{k_h} k_r, i \xrightarrow{e} \text{st}_1 :: k :: \text{st}_2 \implies \text{lift}_{k_h,k_e}(\cup (\text{se}(i)) :: \text{st}_2) 
\end{align*}
\]

Figure 1. The rule for \texttt{ifeq} and a rule for \texttt{invokevirtual}

Intuitively, the function \texttt{se} gives for each program point \( i \) a security level \( k \) such that the instruction at \( i \) cannot store to locations of a level lower than \( k \); the function \texttt{st} associates with each program point a stack of security levels such that the operands on the actual stack cannot be at level higher than the one indicated by \( st \); at last \texttt{sgn} is the currently used security signature for the analysed method.
Note that st and se are chosen for particular signature sgn. This signature, in turn, comes from Policies \( \Gamma \) and for each security level \( s \in S \) we have a single signature. In this light we may consider se and st as collections indexed by elements of \( S \) only.

The main theorem of [BPR06] states that typable programs are safe.

**Theorem 1.** (from typable to safe programs)

Let \( P \) be a Java bytecode program, \((k_{obs}, ft, \Gamma)\) a desired security policy, and cdr a control dependence regions structure satisfying SOAP. If \( P \) is typable with respect to \((k_{obs}, ft, \Gamma)\) and cdr then \( P \) is safe with respect to \( \Gamma \).

**Proof.** See [BPR06, Section 6].

3 Translation from the information flow system

The translation of the information flow system into the bytecode program logic is done with the use of the BML syntax. The BML formulas can be translated to the actual bytecode program logic with the use of the translation in [Con06]. Here is a brief summary of the syntax elements which are useful in the presentation of the translation to BML.

We use the modifier \textit{ghost} to indicate that a particular variable is not a program variable, but a specification variable. Other Java type modifiers such as \texttt{public}, \texttt{private}, \texttt{final}, \texttt{static} have the same meaning as in the case of Java declarations. We use the BML syntax to denote the logical connectives i.e. \&\& means the logical conjunction, || the logical alternative, and ! the logical negation. The logical implication is denoted as \( \Rightarrow \). We also use the BML syntax to access and initialise elements of arrays and to denote types of variables. We also use here the general quantifier. The syntax of the quantifier expression is as follows:

\[
(\forall \text{variable declaration}; \text{bound on the quantification}; \text{actual formula})
\]

where the \textit{variable declaration} has the same form as a one line variable declaration in Java and introduces the variables over which the quantification takes place. The \textit{bound on the quantification} is a formula the goal of which is to restrict potentially infinite domain of the quantification to be finite and it can be any boolean BML expression. At last, the \textit{actual formula} is the formula we are interested in. A \textit{bound on the quantification} \( B \) and an \textit{actual formula} \( A \) are understood as the implication from \( B \) to \( A \).

In order to express in the bytecode program logic that there is no information flow in the program \( P \) we need to add some formula annotations to \( P \) and to extend the method specification tables for \( P \) with the formulas which encode the conditions from Definition 3. Both can be done separately for each method. The translation uses extensively ghost fields.

\textbf{Macros} For the needs of the translation we assume the following additional numbers are calculated statically. Some of them correspond to the program \( P \) and others to methods in \( P \):
1. maxEx the number of all exception types in P.
2. maxS the maximal security level used to type-check P; the level \( 0 \notin S \) will be used to mark the undefined value. Let \( S_0 = S \cup \{0\} \).
3. maxNbArg the maximal number of arguments of all methods in P.
4. lm the maximal label of the method \( m \).
5. maxStack the maximal height of the stack in the execution of method \( m \).

These values are inlined in the specifications below (i.e. we do not use any special ghost variable definitions to refer to them).

Data to translate The annotations we need are of two kinds. The first group consists of data used in the information-flow type system, namely:

- a table \( \Gamma \) of method signatures,
- a global policy \( \mathit{ft} \) that provides security levels of fields,
- a cdr structure \( (\mathit{region}_m, \mathit{jun}_m)^3 \),
- functions \( \mathit{classAnalysis}, \mathit{excAnalysis}, \mathit{nbLocals}, \mathit{nbArguments}, \mathit{Handler} \).

The second group comes from the Definition 3. For each policy signature of a given method these are:

- a security environment \( \mathit{se} : PP \rightarrow S \),
- a function \( \mathit{st} : PP \rightarrow S^* \).

Security level of fields can be stored in ghost fields in the corresponding class. For each class field \( f \) (both static and instance) we define:

\[
\text{public final ghost } S \ \mathit{gft}_f = \mathit{ft}(f); 
\]

this variable allows to consult the security level of the field \( f \). The domain of the functions \( \Gamma, \mathit{excAnalysis}, \mathit{nbLocals}, \mathit{nbArguments} \) is the name space for methods names. They are stored in ghost fields of the class where the given method name appears highest in the class hierarchy. The other data is stored as local ghost variables of the actual methods.

In each class we add the following ghost field definitions initialised according to the static data mentioned above. In the definitions below we use a fixed correspondence between the exception types and the natural numbers \( 0, \ldots, \maxEx - 1 \). For each method \( m \) (both static and instance) with the identifier \( N(m) \) we define a set of ghost variables. These variables will be used as constants; they will never be changed. The initial values of all the ghost variables we use here are defined to correspond directly to the values of real values/functions.

\[
\text{public static final ghost } \text{int } \mathit{gnbArguments}_m N(m) = \mathit{nbArguments}(N(m)); \\
\text{public static final ghost } \text{int } \mathit{gnbLocals}_m N(m) = \mathit{nbLocals}(N(m)); \\
\text{public static final ghost } \text{boolean } [\maxEx] \ \mathit{gecxAnalysis}_m N(m) = \\
\]

\(^3\) We actually do not provide the definition for \( \mathit{jun}_m \) as the function does not occur in the typing rules in [BPR06].
The last two definitions make use of some additional values which are defined and explained below. The information contained in \texttt{gexcAnalysis\_N(m)} is defined with the use of:

\[
e_i = \begin{cases} 
\text{true} & \text{when } \text{excAnalysis}(N(m)) \text{ says that the exception number } i \text{ is risen in } m, \\
\text{false} & \text{otherwise.}
\end{cases}
\]

The security signature \( \Gamma_m[i] = k_p \xrightarrow{k_h} k_r \) allows us to give the values for \( s_{i,j} \).

\[
s_{i,j} = \begin{cases} 
 k_p(j) & j < |k_p|, \\
 k_h & j = |k_p|, \\
 k_r(j - |k_p| - 1) & j > |k_p|.
\end{cases}
\]

This definition allows us to explain the meaning of \( gsgn[i][j] \) in the following way. For a given security level \( i \), \( gsgn[i][0] \) is the security level of the object that calls the method \( m \), \( gsgn[k][1\ldots n\text{bLocals}\_N(m)] \) are security levels of parameters and local variables (note that \( \text{nbLocals}(N(m)) = |k_p| - 1 \)), the value \( gsgn[k][\text{nbLocals}\_N(m) + 1] \) is the level of heap operations, the value \( gsgn[k][\text{nbLocals}\_N(m) + 2] \) is the level of a normal return value and

\[
gsgn[k][\text{nbLocals}\_N(m) + 3\ldots n\text{bLocals}\_N(m) + 3 + \text{maxEx} - 1]
\]

are the security levels in which corresponding exceptions might be propagated (note that \( \text{maxEx} > |k_r| - 1 \)).

We define also local ghost variables associated with the method \( m \):

\[
\text{ghost boolean } [\text{|im|]}[\text{|maxEx|]} \text{ class Analysis} = \{
\{ c_{0,0}, \ldots, c_{0,\text{maxEx}-1} \}, \\
\ldots
\{ c_{m-1,0}, \ldots, c_{m-1,\text{maxEx}-1} \}
\};
\]

where

\[
c_{i,j} = \begin{cases} 
\text{true} & \text{when } \text{classAnalysis}(i) \text{ says that the exception number } j \text{ can be risen in } m \text{ at } i, \\
\text{false} & \text{otherwise.}
\end{cases}
\]

\[
\text{ghost boolean } [\text{|im|}] [\text{|maxEx+1|}] [\text{|im|]} \text{ region} = \{
\{ r_{0,0,0}, \ldots, r_{0,0,\text{im}-1} \}, \\
\ldots
\{ r_{m-1,0,0}, \ldots, r_{m-1,0,\text{im}-1} \}
\};
\]

\[
\text{public static final ghost S0 } [\text{maxS}] [\text{gnbLocals\_N(m)+3+\text{maxEx}}]\ gsgn\_N(m) = \{
\{ s_{0,0}, \ldots, s_{0,\text{gnbLocals}\_N(m)+3+\text{maxEx}-1} \}, \\
\ldots
\{ s_{\text{maxS}-1,0}, \ldots, s_{\text{maxS}-1,\text{gnbLocals}\_N(m)+3+\text{maxEx}-1} \}
\};
\]
where

\[
    \begin{align*}
    r_{i,j,k} &= \begin{cases} 
        \text{true when } k \in \text{region}_m(i,e) \text{ and} \\
        \text{the exception number corresponding to } e \text{ is } j,
    
    \text{true when } k \in \text{region}_m(i,\emptyset) \text{ and} \\
    j = \maxEx, \n
    \text{false otherwise.}
    \end{cases}
    \end{align*}
\]

Note that we use the index \( \maxEx \) on the second coordinate to encode the region information for the normal execution.

\[
    \begin{align*}
    \text{ghost int } [lm][\maxEx] \ g\text{Handler} &= \{ \\
    \{ h_{0,0}, \ldots, h_{0,\maxEx-1} \}, \\
    \ldots \\
    \{ h_{\maxEx-1,0}, \ldots, h_{\maxEx-1,\maxEx-1} \}
    \}. 
    \end{align*}
\]

where \( h_{i,j} = \text{Handler}_m(i,e) \) with \( e \) corresponding to the exception number \( j \). Handler function is encoded as a twodimensional array of boolean values, since the value of a handler is not used in the type-system.

As noted below Definition 3, we may assume that \( \text{se} \) and \( \text{st} \) are indexed with security levels from \( S \). We use the notation \( \text{se}_i \) and \( \text{st}_i \) for \( i \in S \) to refer to the elements of the indexed families.

\[
    \begin{align*}
    \text{ghost S } [\maxS][lm] \ g\text{se} &= \{ \\
    \{ v_{0,0}, \ldots, v_{0,\maxS-1} \}, \\
    \ldots \\
    \{ v_{\maxS-1,0}, \ldots, v_{\maxS-1,\maxS-1} \}
    \}; 
    \end{align*}
\]

where \( v_{i,j} = \text{se}_i(j) \).

\[
    \begin{align*}
    \text{ghost S0 } [\maxS][lm][\maxStack] \ g\text{st} &= \{ \\
    \{ t_{0,0,0}, \ldots, t_{0,0,\maxStack-1} \}, \\
    \ldots \\
    \{ t_{0,\maxS-1,0}, \ldots, t_{0,\maxS-1,\maxStack-1} \}, \\
    \ldots \\
    \{ t_{\maxS-1,0,0}, \ldots, t_{\maxS-1,0,\maxStack-1} \}
    \}. 
    \end{align*}
\]
where $t_{i,j,k} = n$ whenever $(k+1)$-st element of the sequence $st_i(j)$ is $n$. Note that the function $st_i$ gives security levels for the stack positions so that the length of each $st_i(j)$ is less that the maximal stack length $maxStack$. We also assume that for each $i, j$ the elements of $gst[i][0][j]$ are zero which corresponds to the fact that the operand stack at the beginning of a method is empty.

*Translating non-interference property* Once we have all the needed annotations, we may translate the property described in Definition 3. We do it for each method $m$ separately and we decide to use the local annotation table $Q_m$ (as in [Con06, Chapter 3]).

$Q_m$ is a finite partial map which for a program label $i$ occurring in $m$ gives an assertion $Q_i(c0, c)$. If a program point $i$ in $m$ is annotated with $Q_m$ then $Q_i(c0, c)$ is supposed to hold in every state $c$ encountered at label $i$ during any execution of $m$ with the initial state $c0$ satisfying $R_m(c0)$ (i.e. the precondition of the method).

Let us describe how to extend a given specification $Q_m$ so that it ensures the non-interference property. According to Definition 3 we need to state that for every signature of the method (i.e. security level $s$), every label $i$, every exception $e$, and every $j$, such that $i \mapsto e j$ (or $i \mapsto e$) some properties hold. For every $i$, these properties are expressed by formulas $N(i)(s)$. We define $QNI_m$, the local annotation table extended with non-interference checking, as

$$QNI_m(i) = \lambda c0 \in \text{State} \ rarr c' \in \text{State}$$
$$Q_i(c0, c) \land \land N(i)(1) \land \ldots \land N(i)(\text{maxS}). \quad (1)$$

The formulas $N(i)(s)$ have similar form; it is

$$\forall \text{int } e, j; 0 \leq e \land \land e \leq \text{maxEx} \land \land 0 \leq j \land \land j < \text{lm}$$
$$ \ldots \land \ldots (\text{Reg}_{i}^{\text{inst}(i), s}(p_1) \\| \ldots \| \text{Reg}_{i}^{\text{inst}(i), s}(p_k))) \land \land \ldots$$

where $\text{inst}(i)$ is the instruction at the label $i$ in the method body $m$. Note also that $i \mapsto e j$ (as well as $i \mapsto e$) is a static information which can be easily defined directly as a subformula to be inserted at the place or with help of an additional ghost array.

Every $\text{Reg}_{i}^{\text{inst}(i), s}(p_1)$ corresponds to one of the possibly applicable typing rules for instruction $\text{inst}(i)$ in case $i \mapsto e j$ (or $i \mapsto e$) where $p_i$ are parameters of $\text{inst}$.
For example $\text{Reg}^{\text{ifeq}}_1(j)$ corresponds to ifeq and equals to

\begin{align*}
1 & \quad e \equiv \text{maxEx} \quad \&\& \quad (\forall \text{forall int } j' : 0 \leq j' \quad \&\& \quad j' < \text{lm}; \\
2 & \quad \text{region}[i][\text{maxEx}][j'] \Rightarrow gst[s][i][\text{cntr}] \leq gse[s][j']) \quad \&\& \\
3 & \quad (\forall \text{forall int } p : 0 \leq p \quad \&\& \quad p \leq \text{cntr } - 1; \\
4 & \quad gst[s][i][p] \supseteq gst[s][i][\text{cntr}] \leq gst[s][j][p]) \quad \&\& \\
5 & \quad (\forall \text{forall int } p; \text{cntr } \leq p; \\
6 & \quad gst[s][j][p] = 0)
\end{align*}

The two last lines state that $\text{lift}_k(st) \subseteq st(j)$, where $st$ is $st(i)$ without its top element; the last one explicitly checks that $st(j)$ is one element shorter than $st(i)$.

For invokevirtual there are three different typing rules. Let us present the translation for the one presented on Fig. 2; $\text{Reg}^{\text{invokevirtual}_{st}}_1(N(m'))$ equals:

\begin{align*}
1 & \quad e \equiv \text{maxEx} \quad \&\& \quad (\forall \text{forall int } n, k, k, k, s < \text{maxNbArg} \quad \&\& \quad 0 \leq k, k, k, k, k < \text{maxS}; \\
2 & \quad n = \text{gnsArguments}_N(m') \quad \&\& \\
3 & \quad \text{cntr } \geq n \quad \&\& \\
4 & \quad k = gst[s][i][\text{cntr } - n] \quad \&\& \\
5 & \quad k = \text{region}[i][\text{maxEx}][j'] \\
6 & \quad (\forall \text{forall int } p; 0 \leq p \quad \&\& \quad p \leq n - 1; \\
7 & \quad gse[s][i][\text{cntr } - p] \leq gssN[m']\text{gnsLocals}[m' + 3 + e] | \\
8 & \quad \text{gnsAnalysis}[N(m')][n]]) \\
9 & \quad k \sqcup gssN[m'][s][\text{gnsLocals}[m'] + 1] \sqcup gse[s][i] \leq gssN[m'][k][\text{gnsLocals}[m'] + 1] \quad \&\& \\
10 & \quad k \leq gssN[m'][k][0] \quad \&\& \\
11 & \quad (\forall \text{forall int } p; 0 \leq p \quad \&\& \quad p \leq n - 1; \\
12 & \quad gst[s][i][\text{cntr } - p] \leq gssN[m'][k][p + 1]) \quad \&\& \\
13 & \quad (\forall \text{forall int } j'; 0 \leq j' \quad \&\& \quad j' < \text{lm}; \\
14 & \quad gssN[m'] \text{region}[i][\text{maxEx}][j'] \\
15 & \quad (\forall \text{forall int } p; 0 \leq p \quad \&\& \quad p < \text{cntr } \quad \&\& \\
16 & \quad gssN[m'][k][\text{cntr } - n] \quad \&\& \\
17 & \quad (k \sqcup k_e \leq gse[s][j']) \quad \&\& \\
18 & \quad (\forall \text{forall int } p; 0 \leq p \quad \&\& \quad p < \text{cntr } \quad \&\& \\
19 & \quad gssN[m'][k][\text{cntr } - n] \quad \&\& \\
20 & \quad (k \sqcup k_e \sqcup gssN[m'][k][n + 2] \sqcup gse[s][i]) \leq \\
21 & \quad gssN[m'][k][\text{cntr } - n + 1] \quad \&\& \\
22 & \quad (\forall \text{forall int } p; \text{cntr } - n + 1 \leq p; \\
23 & \quad gssN[m'][k][p] \quad (0)))))
\end{align*}

We recall that $N(m')$ stands for the method identifier of the called method $m'$. In order to shorten the formula we introduce shorthands $n, k$ for the number of the parameters for the method to be called and the security level of the object that contains the called method, respectively. We also introduce the shorthand $k_e$, with the informal use of $\sqcup$, to denote the least upper bound of security levels.
corresponding to exceptions allowed by the excAnalysis:

\[ k_e \triangleq \bigcap \{ gsgn_{\text{N}}(m') [k] \mid \text{gexcAnalysis}_{\text{N}}(m') \} \]

Again, it is easy to define the appropriate formula which expresses this property as we can enumerate all the values that are involved here.

It is worth pointing out that the subformulas starting from the line 12 state that \( \text{lift}_{k, k_e}(k'_r [n] \sqcup se(i)) :: st_2 \subseteq st(j) \).

### 4 Proof of non-interference

In our approach the non-interference property is obtained in two steps. The first one is Theorem 1 which provides the link between safety and typability of a program. The second one relates typability and the fact that the program verifies correctly in the bytecode program logic. We additionally provide a result which allows to mix the specifications that result from our translation with specifications that come from other sources (e.g. are written by hand).

Please recall that like in [BPR06] we assume that functions classAnalysis, excAnalysis, nbLocals, nbArguments, Handler are correct.

The goal of the translation we provided is to obtain the following theorem.

**Theorem 2.** (typechecking and verifiability)

Let \( P \) be a Java bytecode program, \((k_{\text{obs}}, ft, \Gamma)\) a desired security policy, and \(cdr\) a control dependence regions structure satisfying SOAP. Let \( TR \) be the translation defined in Section 3 that adds base logic annotations to \( P \). For every security environment family \( \{se_i : PP \rightarrow S\}_{i \in S} \) and a family of functions \( \{st_i : PP \rightarrow S^*\}_{i \in S} \) such that \( st_i(0) = \epsilon \) for all \( i \),

\[ \Rightarrow \text{if the annotated program } TR(P, k_{\text{obs}}, ft, \Gamma, cdr, se, st) \text{ verifies correctly then } P \text{ with the policy } (k_{\text{obs}}, ft, \Gamma) \text{ and } cdr \text{ is typable,} \]

\[ \Leftarrow \text{if the program } P \text{ with the policy } (k_{\text{obs}}, ft, \Gamma) \text{ and } cdr \text{ is typable with se, st, and all } Q_i(c0, c) \text{ in } QNI_{m_i} \text{ in (1) on page 11 are empty then the annotated program } TR(P, k_{\text{obs}}, ft, \Gamma, cdr, se, st) \text{ verifies correctly.} \]

**Proof.** We present here a sketch of the proof only.

(\( \Rightarrow \)) Suppose that the annotated program \( TR(P, k_{\text{obs}}, ft, \Gamma, cdr, se, st) \) verifies correctly. We want to show that \( P \) is typable, i.e. that every method \( m \) in \( P \) is typable with respect to every signature in Policies\( (m) \). We need to verify that the condition in Definition 3 is fulfilled. Let sgn be a signature corresponding to security level \( s \). We take se and st as above. \( st_i(0) = \epsilon \) is guaranteed by initialization of gst. Since \( P \) verifies correctly the formula \( N(i)(s) \) holds for every \( i \). The conditions (1)-(2) from Definition 3 are guaranteed by the fact that all the typing rules are faithfully modelled in the logic. We provide here a motivation that the typing rule is faithfully modelled in case the instruction at the label \( i \) is ifeq \( j \) or invokevirtual \( N(m') \). In other cases the proof is similar.
ifeq. Note that in this case we must ensure the point (1) only if $i$ has a successor in the method. We have to ensure that the typing condition is fulfilled. This is guaranteed by the first $\forall$forall subformula of (3). The condition $s \subseteq st(j)$ is guaranteed in turn by the second $\forall$forall subformula of (3).

**invokevirtual** Let us do only the part of the proof and show that $i \mapsto^0 j$ and $\text{Reg}_{i}^{\text{invokevirtual},s}(N(m'))$ imply that the premises of the typing rule for invokevirtual from Figure 2 are satisfied. In $\text{Reg}_{i}^{\text{invokevirtual},s}(N(m'))$ first few lines introduce constants used later; $n$ is the number of arguments of the method $N(m')$, $k$ is a security level of the object that calls the method and $k_s$ for the upper bound of all security levels corresponding to exception classes that belong to excAnalysis($N(m')$). Let us analyse the premises of invokevirtual rule and justify to which part of the formula $\text{Reg}_{i}^{\text{invokevirtual},s}(N(m'))$ they correspond:

- $k \cup k_h \cup se(i) \leq k'_h$ is satisfied because of the subformula in lines 9-10,
- $k \leq k'_h[0]$ corresponds to line 11,
- $\forall i \in [0, \text{length}(st_1) - 1]$, $st_1[i] \leq k'_h[i + 1]$ corresponds to lines 12-14,
- $\forall j \in \text{region}(i, \emptyset)$, $k \cup k_e \leq se(j)$ corresponds to lines 15-18

Because of the formula in line 4 the stack $st(i)$ is of length at most $n + 1$. The condition $\text{lift}_k((k'[n] \cup se(i)) :: st_2) \subseteq st(j)$ is expressed in lines 19-24: lines 21-22 take care of the top element of $\text{lift}_k((k'[n] \cup se(i)) :: st_2)$, 19-20 of all other elements and 23-24 ensure that $st(j)$ is not to high.

($\Leftarrow$) Suppose that the program $P$ with the policy $(k_{\text{obs}}, ft, \Gamma)$ and $\text{cdr}$ is typeable with $se$ and $st$. We have to ensure that each $\text{QNI}(m,i)$, for $i$ being a label in the method $m$, holds. As $Q_1(c0,c)$ is empty, it is enough to check that each $N(i)(j)$ holds for $j \in S$. Each of the $N(i)(s)$ has similar structure presented in (2). It is enough to show that one of the corresponding $\text{Reg}_i^{\text{inst}(i),s}(p_i)$ holds in case $i \mapsto^e j$ (or in case $i \mapsto^c$). As the method is typeable, we know that $\Gamma, ft, \text{region}_m, se, sgn, i \vdash^e st(i) \implies s (or \Gamma, ft, \text{region}_m, se, sgn, i \vdash^c st(i) \implies )$ can be inferred. This is done with one of the rules, say $l$-th. Now, we have to make sure that the corresponding translation formula $\text{Reg}_i^{\text{inst}(i),s}(p_i)$ holds. We show this in case $\text{inst}(i)$ is ifeq $j$.

- the subformula $e == \text{maxEx}$ in the pattern (3) holds as the only rule for ifeq $j$ works only when we deal with the normal execution,
- the second subformula of (3) holds as the typing rule guarantees that $\forall j' \in \text{region}(i, \emptyset)$, $k \leq se(j')$,
- the third subformula of (3) holds as the typability requires that $\text{lift}_k(st(j)) \subseteq st(j)$, where $st$ is $st(i)$ without its top element,
- the fourth subformula of (3) holds as the $st(j)$ is not determined for indices greater than the top of the operand stack.

This finishes the proof in this case. The cases of other instructions are similar.

This theorem says that whenever a program with annotations proposed in Section 3 successfully verifies it also successfully typechecks. This property implies the following result:
Theorem 3. (non-interference)
If TR(P, k_{obs}, ft, Γ, cdr, se, st) verifies correctly then the program P is safe.

Proof. Suppose that TR(P, k_{obs}, ft, Γ, cdr, se, st) verifies correctly. By Theorem 2 it means that P is typable with the policy (k_{obs}, ft, Γ) and cdr. As Theorem 1 states, this means that P is safe.

4.1 Proof of stability

Theorem 4 below states that we can safely extend the specifications so that the non-interference property is preserved.

Definition 4. (specifications in conflict)
We say that specifications are in conflict with the translation TR whenever any element of the ghost arrays or variables defined in Section 3 is set.

Theorem 4. (stability)
Let P' be a specification extension of TR(P, k_{obs}, ft, Γ, cdr, se, st) that does not conflict with TR(P, k_{obs}, ft, Γ, cdr, se, st). If P' verifies correctly then P satisfies the non-interference property.

Proof. We present here a sketch of the proof only. When the specification extension does not conflict with the translation TR(P, k_{obs}, ft, Γ, cdr, se, st) then the values of all the variables used in the translation are the same. In this light, the logical values of the formulas are the same as in case there are no additional specifications. By Theorem 3 we obtain non-interference for P.

5 Final remarks

Declassification Our translation modifies the local assertion table Q_m so that each typing rule is checked at the instruction label it handles. This enables an easy method to declassify information by means of the assignment to a ghost variable. Usually, the declassification should occur when a value on a high security level on the stack at a program point i should be stored in a low level field. The current rules prevent this, but they exploit the information stored in an entries of gsgn and gst arrays that correspond to i. It is enough to store there different values just at the instruction that requires the declassification and revert them back at the next one.

Finite range of levels The original order used in the information flow type system has not been restricted to be finite. Our translation relies crucially on the fact that the order is finite. In practice, however, it is very difficult to check the non-interference in case of essentially infinite policies—in particular such policies should be effectively enumerable and thus the checking that a policy is fulfilled becomes rather an algorithm verification task than static checking.
Conclusion We presented here a translation of the information-flow type system defined in [BRN06] and further explored in [BPR06] to the bytecode program logic presented in and [Con06,BH06]. The translation is in fact through the BML specification language syntax which should facilitate quicker adoption of the translation by tools. This translation gives possibility to combine the logical verification with the properties obtained using a type system. It seems also that our formalisation can enable an easy control over the declassification.

References


