Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 253 (2009) 113–127

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

The Non-Interference Protection in BML

Aleksy Schubert[1,2] Daria Walukiewicz-Chrząszcz[1,3]

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.

Keywords: Java, bytecode, specification, BML, non-interference

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

[1] This work was partly supported by Polish government grant 177/6.PR UE/2006/7 and and Polish government grant 3 T11C 002 27.

[2] Email: alx@mimuw.edu.pl

[3] Email: daria@mimuw.edu.pl

doi:10.1016/j.entcs.2009.11.018

program logic [7] and, based on this foundation, a specification language for the bytecode — Bytecode Modeling Language (BML) [11]. The latter language is based on the design-by-contract principles and is derived from a Java specification language called Java Modeling Language [18,19,20] which has wide tool support [10].

We consider here the non-interference property of bytecode programs. Many type systems to guarantee the non-interference have been proposed for while programs (e.g. [16,17,21]) as well as for other formal languages (e.g. [14,15]). The starting point of our work is the information-flow type system for Java bytecode [4,5] that guarantees the 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 [7,12] such that the correctness of the typing is equivalent to the verifiability of bytecode program logic formulae. Here, the soundness of the type system guarantees the non-interference property of verifiable programs.

It is worth pointing out that the translation has a few desirable features. First of all, once the translation is in place it is possible to use a toolset based on logical methods rather than typed ones. This allows to incorporate the guarantees of the type system into a foundational proof-carrying code (PCC, [2]) framework and use the non-interference property together with other properties originally formulated and expressed in the foundational fashion [4]. Moreover, the wide selection of JML based verification tools and methods [10] is a solid basis to aim for a platform of foundational PCC certificates for Java bytecode. Another desirable feature of this method is the fact that the resulting model of the non-interference is more flexible than the one based on typing. This is important whenever the non-interference property must be weakened, for example when declassification is needed (in particular when the code encrypts confidential data). The translation we provide is designed so that it is relatively straightforward to adjust it to various declassification needs.

The Hoare-like logic available in BML is in fact only first order logic with very weak inventory of relations which allow to compare heaps at different points of program. In particular, it is impossible to express there the *agreement* operator by Amtoft et al [1]. Moreover, BML also does not contain any features of dynamic or algorithmic logic so it is impossible to express the non-interference property by relating the heaps after two different program runs as in [6] or [13]. Therefore, we decided to model the type system derivations in BML and base the safety of the program on the type system soundness. Still another limitation of the approach based on BML is that the self-composition [3] cannot be expressed here (although it is available in MOBIUS logic).

The paper is structured as follows. In Sect. 2, we fix the notation and present the basic notions which are used in the paper. Sect. 3 provides an exposition of the translation of the type based system to the logic based one. This translation is supplemented by a theorem that the resulting specifications guarantee the non-

[4] The translation from this paper does not reduce the trusted logical base to the one of the foundational PCC. To achieve that one has to link the resulting formulae with the non-interference property expressed in the foundational logic e.g. the property expressed in [8] for while programs.

interference property in Sect. 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 Sect. 4.1. At last we present the final remarks in Sect. 5 where we sketch the way the declassification can be introduced.

2 Preliminaries

In this section we fix the notation used throughout the paper. We, generally, follow the papers [4,12]. We also present informal description of some 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 $\mathrm{dom}(f)$ to denote the domain of the (partial) function f. Similarly, $\mathrm{rng}(f)$ is the image of f. We write $f : A \rightharpoonup B$ when f is a partial function from A to B. The powerset of A is $P(A)$. We write $\boldsymbol{k}$ to denote a vector of values. The notation $|\boldsymbol{k}|$ expresses the length of the vector and $\boldsymbol{k}(0), \ldots, \boldsymbol{k}(|\boldsymbol{k}|-1)$ are subsequent elements of the vector. The set of finite sequences over a set A is A^*.

Java bytecode programs and specifications A Java bytecode program P is a set of classes with one singled out method 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 a body B_m [5]. 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 $\mathcal{PP}_m$ (we omit the subscript m when it is clear from the context).

An annotated program $\hat{P}$ has additionally (among others) for each class C a list Ghost_C of model and ghost fields (i.e. fields which can occur only in specifications), and a method specification table M_C. The bytecode program logic we employ here [7] makes use of the method specification table $\mathsf{M}_C(m)$ associated with each method m. This table consists of:

- 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 Φ_m is the method invariant which holds in each accessible state of the method;
- 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 (this corresponds to the BML `assume` annotations);
- a local annotation table Q_m which assigns to labels in B_m further assertions (this corresponds to the BML `assert` annotations);
- a local instruction table Ins_m which assigns to each label l in the method body

[5] The separation of the identities for the method and its name serves to model the inheritance.

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 $\mathsf{Q}_m(l)$ (this corresponds to the BML `set` annotations).

Security policy We use here the security policy framework from [5]. It is based on assumption that the attacker can observe the input/output of methods only. 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.

Formally, a security policy is expressed in terms of a finite partial order $(\mathcal{S}, \leq)$. This order allows to describe the capabilities of the attacker and the program to be analysed:

- A security level k_{obs} determines the observational capabilities of the attacker (she can observe fields, local variables and return values the level of which is less or equal than k_{obs}).
- A policy function ft assigns to each field its security level. This allows us to express the non-interference property we are interested in.
- A policy function Γ that associates to each method identifier $N(m)$ and security level $k \in S$ a security signature $\Gamma_{N(m)}[k]$[6]. 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 $\mathsf{Policies}_\Gamma(m) = \{\Gamma_{N(m)}[k] \mid k \in \mathcal{S}\}$. The security signature has the shape $\boldsymbol{k_p} \xrightarrow{k_h} \boldsymbol{k_r}$:
  - · The vector $\boldsymbol{k_p}$ 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 $\boldsymbol{k_r}$ 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}.

Non-interference The non-interference property is articulated by a safety definition in [4,5]. Informally, a program is non-interferent if all its methods are safe; a method is safe if two terminating runs of the method with inputs that cannot be distinguished by an attacker, and equivalent heaps, yield results that cannot be distinguished by the attacker and if the method cannot modify the heap in a way that is observable by an attacker.

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

[6] In [5] a less precise notation $\Gamma_m[k]$ is used.

in an information flow system [5], we need the same descriptions of the bytecode program structure.

We use a binary successor relation $\mapsto^{\tau} \subseteq PP \times PP$ defined on the program points. This relation is parametrised by a tag τ since a instructions may have several successors as they may execute normally (the tag is $\emptyset$) or may trigger exceptions (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 $i \mapsto$ when $\mapsto$ is undefined for i i.e. if i corresponds to a return instruction (or $i \mapsto^{\tau}$ if i corresponds to an instruction that may throw an exception that is not handled locally). Note that an instruction may have more than one successor.

We assume that a bytecode program P comes equipped with a *control dependence regions* structure cdr which consists of a pair of partial functions $(\mathtt{region}, \mathtt{jun})$. 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 $\mathtt{region}$ function describes the internal parts of these regions while $\mathtt{jun}$ the connections between them. The types of the functions are the following:

$$\mathtt{region}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \to P(\mathcal{PP}) \qquad \mathtt{jun}_m : \mathcal{PP} \times (\{\emptyset\} + \mathcal{C}) \rightharpoonup \mathcal{PP}$$

The functions can be axiomatised by the SOAP (Safe Over APproximation) properties [5, Sect. 4] ensuring that the control dependence regions structure correctly describe information flow in a program P.

Typable programs To check that a program is non-interferent one may use a type system presented in [5]. In this type system, every method is checked against its signatures separately. The type system is parametrised by:

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

We assume also that the functions below are given and that they are correct:

(i) $\mathsf{classAnalysis}$ which for a program point returns the set of exception classes of exceptions that may be thrown at the program point.

(ii) $\mathsf{excAnalysis}$ which for a method name $N(m)$ returns the set of exception classes that are possibly thrown by m.

(iii) nbLocs which for a method name $N(m)$ returns the number of its local variables.

(iv) nbArgs which for a method name $N(m)$ returns the number of its arguments.

(v) $\mathsf{Handler}_m$ 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 2.1 *(typable programs and methods)* Method m is *typable* with respect to ft, Γ, $\mathtt{region}_m$ and a signature sgn if there exists a security environment $\mathsf{se} : PP \to S$ and a function $\mathsf{st} : PP \to S^*$ such that $st(0) = \varepsilon$ and for all $i, j \in PP$, $e \in \{\emptyset\} \cup \mathcal{C}$:

$$\frac{P_m[i] = \texttt{ifeq}\ j \qquad \forall j' \in \texttt{region}(i, \emptyset),\ k \leq se(j')}{\Gamma, \mathsf{ft}, \texttt{region}, \mathsf{se}, \boldsymbol{k_p} \xrightarrow{k_h} \boldsymbol{k_r}, i \vdash^{\emptyset} k :: st \implies \texttt{lift}_k(st)}$$

Figure 1. The typing rule for `ifeq`

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

(2) if $i \mapsto^e$ then $\Gamma, \mathsf{ft}, \texttt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies$

where $\sqsubseteq$ denotes the point-wise partial order on the type stack with respect to the partial order taken on security levels. An example of a rule to derive $\ldots \vdash^e \ldots \implies \ldots$, for `ifeq`, is given in Figure 1 [7]. The set of all typing rules is presented in [4].

A program P is *typable* with the policy $(k_{\text{obs}}, \mathsf{ft}, \Gamma)$ and cdr satisfying SOAP if every method m from P is typable with respect to ft, Γ, $\texttt{region}_m$ and all signatures in $\mathsf{Policies}_\Gamma(m)$.

Intuitively, the function 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 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 sgn is the currently used security signature for the analysed method.

Let us analyse the rule on Figure 1. The assertion under the line assumes that given are: a table of method signatures Γ, security levels of fields ft, a `region` function, a security environment function se, a signature of the current method m $\boldsymbol{k_p} \xrightarrow{k_h} \boldsymbol{k_r}$ and a program point i. It asserts safety for the cases when the normal (non-exceptional) step is taken by the programme. A safe step, here, must transform in the abstract world the stack $h :: \mathsf{st}$ to a stack $\texttt{lift}_k(st)$. This rule allows to assert it provided that two requirements are fulfilled. The first one $P_m[i] = \texttt{ifeq}\ j$ requires that the instruction at point i is `ifeq`. The second one, $\forall j' \in \texttt{region}(i, \emptyset)$, $k \leq se(j')$, describes the requirement on the code executed after the branch. The level k is the security level of the value read by `ifeq`. All program points in $\texttt{region}(i)$ are points executing under the guard of i. Since security environment se is meant to be the upper bound of all the guards under which the program point execute, it is natural that $k \leq \mathsf{se}(j')$ for all $j' \in \texttt{region}(i)$. The stack on the right-hand side of $\implies$ is lifted; $\texttt{lift}_k$ is the point-wise extension to stack types of $\lambda l.k \sqcup l$. The lifting operation prevents illicit flows through the operand stack; the stack shows what happened before, for example what was the value at `ifeq`, and lifting prevents leaking of this information.

Note that st and se are chosen for particular signature sgn. This signature, in turn, comes from $\mathsf{Policies}_\Gamma(m)$ 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 [5] states that typable programs are non-interferent.

[7] We limit our exposition to the `ifeq` instruction due to the space restrictions.

3 Translation from the information flow system

In order to express in the bytecode program logic that there is no unwanted 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 formulae which encode the conditions from Def. 2.1. Both can be done separately for each method. The translation we present here uses extensively BML and in particular ghost fields of the formalism.

3.1 Summary of BML

The translation of the information flow system into the bytecode program logic is done with the use of the BML syntax. The BML formulae can be translated to the actual bytecode program logic with the use of the translation in [12]. Here is a brief summary of the relevant BML syntax.

We use the modifier **ghost** to indicate that a particular variable is not a program variable, but a specification variable. Other Java type modifiers such as **public**, **private**, **final**, **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 ==>. 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 variable declaration; bound on the quantification; actual formula)

where the *variable declaration* has the same form as a variable declaration in Java and introduces the quantified variables. The *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 expression. At last, the *actual formula* is the formula we are interested in. A *bound on the quantification* B and an *actual formula* A are understood as the implication B ==> A.

3.2 Data to translate

The annotations we need are of two kinds. First of all, a reliable description of the security requirements and the program structure must be provided at the side of BML i.e. the security levels of fields, method signatures, cdr structure etc. must be represented in the form of ghost variables. Therefore, the first group of the translated data consists of:

- a table Γ of method signatures,
- a global policy ft that provides security levels of fields,
- a cdr structure $(\mathtt{region}_m, \mathtt{jun}_m)$ [8],

[8] We actually do not provide the definition for $\mathtt{jun}_m$ as the function does not occur in the typing rules in [4].

- functions classAnalysis, excAnalysis, nbLocs, nbArgs, Handler.

The idea of our translation is that we check by means of the BML formulae that a derivation in the information-flow type system is correct. Therefore, we must translate the data on which the type system operates. To this end we need to transform the functions mentioned in Def. 2.1 i.e. for each policy signature of a method: a security environment $\mathsf{se} : PP \to S$, and a function $\mathsf{st} : PP \to S^*$.

Additionally, we use certain static data which is not defined explicitly in terms of ghost variables, but is inlined in the definitions below. The values of this kind are:

- maxEx the number of all exception types in P.
- maxS the maximal security level used to type-check P; the level $0 \notin S$ will be used to mark the undefined value. Let $S0 = S \cup \{0\}$.
- maxNbArg the maximal number of arguments of all methods in P.
- lm the maximal label of the method m.
- maxStack the maximal height of the stack in the execution of method m.

These values are computed during the translation process.

Security requirements and program structure 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:

```
public final ghost S gft_f = ft(f);
```

this enables access to the security levels of `f`. The domain of Γ, excAnalysis, nbLocs, nbArgs is the set of 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 are stored as local ghost variables of the actual methods.

In the definitions below, we use a fixed correspondence between the exception types and the natural numbers $0, \ldots, \mathsf{maxEx} - 1$. 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.

```
public static final ghost int gnbArgs_N(m) = nbArgs(N(m));
public static final ghost int gnbLocs_N(m) = nbLocs(N(m));
public static final ghost boolean [maxEx] gexcAnalysis_N(m) =
                                 { e_0, ..., e_{maxEx-1} };
public static final ghost S0
  [maxS][gnbLocs_N(m)+3+maxEx] gsgn_N(m) =
    { { s_{0,0}, ..., s_{0,gnbLocs_N(m)+3+maxEx-1} }, ...
      { s_{maxS-1,0}, ..., s_{maxS-1,gnbLocs_N(m)+3+maxEx-1} } };
```

The last two definitions make use of additional values defined below. The information contained in **gexcAnalysis**_$N(m)$ is defined with the use of:

$$e_i = \begin{cases} \textbf{true} & \text{when } \mathsf{excAnalysis}(N(m)) \text{ says that the exception } i \text{ is thrown in } m, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

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

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

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

$$\mathsf{gsgn}[k][gnbLocs\_N(m) + 3 \ldots gnbLocs\_N(m) + 3 + \mathsf{maxEx} - 1]$$

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

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

```
ghost boolean [lm][maxEx] gclassAnalysis =
  { { c_{0,0},...,c_{0,maxEx-1} },...,{ c_{lm-1,0},...,c_{lm-1,maxEx-1} } };
```

where

$$c_{i,j} = \begin{cases} \textbf{true} & \text{when } \mathsf{classAnalysis}(i) \text{ says that the exception } j \text{ can be thrown in } m \text{ at } i, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

```
ghost boolean [lm][maxEx+1][lm] gregion = {
   { { r_{0,0,0},...,r_{0,0,lm-1} },..., { r_{0,maxEx,0},...,r_{0,maxEx,lm-1} } }, ...,
   { { r_{lm-1,0,0},...,r_{lm-1,0,lm-1} },..., { r_{lm-1,maxEx,0},...,r_{lm-1,maxEx,lm-1} } }
 };
```

where

$$r_{i,j,k} = \begin{cases} \textbf{true} & \text{when } k \in \texttt{region}_m(i,e) \text{ and the exception corresponding to } e \text{ is } j, \\ \textbf{true} & \text{when } k \in \texttt{region}_m(i,\emptyset) \text{ and } j = \mathsf{maxEx}, \\ \textbf{false} & \text{otherwise.} \end{cases}$$

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

```
ghost int [lm][maxEx] gHandler =
  { { h_{0,0},...,h_{0,maxEx-1} },..., { h_{lm-1,0},...,h_{lm-1,maxEx-1} } };
```

where $h_{i,j} = \mathsf{Handler}_m(i,e)$ with e corresponding to the exception number j.

Type system data As noted below Def. 2.1, we may assume that se and st are indexed with security levels from S. We use the notation se_i and st_i for $i \in S$ to refer to the elements of the indexed families.

```
ghost S [maxS][lm] gse =
  { { v_{0,0},...,v_{0,lm-1} },..., { v_{maxS-1,0},...,v_{maxS-1,lm-1} } };
```

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

```
ghost S0 [maxS][lm][maxStack] gst = {
      { { t_{0,0,0},...,t_{0,0,maxStack−1} },...,{ t_{0,lm−1,0},...,t_{0,lm−1,maxStack−1} } },
      ...,
      { { t_{maxS−1,0,0},...,t_{maxS−1,0,maxStack−1} },...,
        { t_{maxS−1,lm−1,0},...,t_{maxS−1,lm−1,maxStack−1} } }
};
```

where $t_{i,j,k} = n$ whenever $(k+1)$-st element of the sequence $\mathsf{st}_i(j)$ is n. Note that the function st_i gives security levels for the stack positions so that the length of each $\mathsf{st}_i(j)$ is less that the maximal stack length $\mathsf{maxStack}$. 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 the rules Once we have all the annotations above, we may encode the typability property from Def. 2.1. We do it for each method m separately and we decide to use the local annotation table Q_m (as in [12, Chapter 3]).

Q_m is a finite partial map which for a program label i in m gives an assertion $Q_i(s0, s)$. If the point i in m is annotated with Q_m then $Q_i(s0, s)$ is supposed to hold in every state s at i during any execution of m with the initial state $s0$ satisfying $R_m(s0)$ (i.e. the precondition of the method). Intuitively, $Q_i(s0, s)$ provides the content of the `assert` statement right before the instruction with the label i.

Let us describe how to extend a given specification Q_m so that it ensures the non-interference property. According to Def. 2.1, we need to state that for every security signature of the method (recall that for each security level s there is a separate security signature $\Gamma_{N(m)}[s]$ applicable in the situation when the object on which m is called has the 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 formulae $N(i)(s)$. We define QNI_m, the local annotation table extended with the non-interference checking, as

$$\begin{aligned}\mathsf{QNI}_m(i) = \lambda c0 \in \mathsf{State}\ \ \lambda c \in \mathsf{State}.\\ Q_i(c0, c)\ \&\&\ N(i)(1)\ \&\&\ \ldots\ \&\&\ N(i)(\mathsf{maxS}).\end{aligned} \tag{1}$$

This formula expresses a new assert before the instruction at i which states that the old assert must hold together with all the security guarding formulae which ensure (together with all the formulae for other instructions) that the security signatures of m are obeyed.

The security guarding formulae $N(i)(s)$ have similar form; it is

$$\begin{aligned}&(\backslash\mathsf{forall\ int}\ e,\ j;\ 0 \le e\ \&\&\ e \le \mathsf{maxEx}\ \&\&\ 0 \le j\ \&\&\ j < \mathsf{lm};\\ &\qquad (i \mapsto^e j \texttt{==>} (\mathsf{Reg}_1^{\mathtt{inst}(i),s}(\boldsymbol{p_1})\ ||\ \ldots\ ||\ \mathsf{Reg}_k^{\mathtt{inst}(i),s}(\boldsymbol{p_k})))\ \&\&\\ &\qquad (i \mapsto^e\ \ \texttt{==>} (\mathsf{Reg}_{k+1}^{\mathtt{inst}(i),s}(\boldsymbol{p_1})\ ||\ \ldots\ ||\ \mathsf{Reg}_{k'}^{\mathtt{inst}(i),s}(\boldsymbol{p_{k'}}))))\end{aligned} \tag{2}$$ [9]

where $\mathtt{inst}(i)$ 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 static information which can be defined directly as

[9] We add this subformula only in case the instruction may throw an exception.

a subformula. For example, in the most typical case when the control flow moves to the next instruction, the formula is of the form $j == i + 1$ && $e ==$ maxEx[10]. Here, the condition that maxEx equals e enforces that we consider a normal step. In the case when the method has an exception handler of e_0 at the point j' we define $i \mapsto^e j$ to be $j == j'$ && $e == e_0$. As the `ifeq` j_0 has two successors, but one rule handles the instruction in the type system, the premise of the implication is $(j == i + 1 \mid\mid j == j_0)$ && $e ==$ maxEx in this case.

Every $\mathsf{Reg}_i^{\mathtt{inst}(i),s}(\boldsymbol{p_i})$ for $i = 1, \dots, k$ or for $i = k+1, \dots, k'$ corresponds to one of possibly applicable typing rules for instruction $\mathtt{inst}(i)$ in case $i \mapsto^e j$ (or $i \mapsto^e$). The type system considers an instruction to be correct when at least one of the rules can be successfully satisfied. Therefore, the formulae $\mathsf{Reg}_i^{\mathtt{inst}(i),s}$ are combined as an alternative. Let us point out that the vectors $\boldsymbol{p_i}$ are parameters of the instruction `inst`. For instance, there is one rule for `ifeq` and $\mathsf{Reg}_1^{\mathtt{ifeq},s}(j)$, corresponding to `ifeq` j, equals to

$$
\begin{aligned}
&(\backslash\mathsf{forall\ int}\ j';\ 0 \le j'\ \&\&\ j' < \mathsf{lm};\\
&\qquad gregion[i][\mathsf{maxEx}][j'] ==> gst[s][i][\mathsf{cntr}] \le gse[s][j'])\quad \&\&\\
&(\backslash\mathsf{forall\ int}\ p;\ 0 \le p\ \&\&\ p < \mathsf{cntr};\\
&\qquad gst[s][i][p] \sqcup gst[s][i][\mathsf{cntr}] \le gst[s][j][p])\quad \&\&\\
&(\backslash\mathsf{forall\ int}\ p;\ \mathsf{cntr} \le p\ \&\&\ p \le \mathsf{maxStack};\\
&\qquad gst[s][j][p] = 0)
\end{aligned}
\tag{3}
$$

We can now relate the formula to the rule in Figure 1. First, observe, that the index cntr is a counter of the operand stack; hence $gst[s][i][\mathsf{cntr}]$ points to the top of the stack and it corresponds to k from the rule. The first precondition of the rule in Figure 1 holds as the formula is generated only for `ifeq` instruction. The formula above consists of three $\forall$ subformulae. The first subformula expresses the condition $\forall j' \in \mathtt{region}(i, \emptyset),\ k \le se(j')$ from the precondition in the rule in Figure 1. Recall that maxEx value in second parameter of $gregion$ means a normal (non-exceptional) behaviour. The second and the third subformulae state that $\mathsf{lift}_k(st) \sqsubseteq st(j)$, where st is $st(i)$ without its top element; in particular, the last formula checks that $st(j)$ is one element shorter than $st(i)$ and that the unused part of the stack contains the default value 0.

4 Proof of non-interference

The following theorem relates typability and the fact that the program verifies correctly in the bytecode logic. It says that whenever a program with annotations proposed in Sect. 3 successfully verifies it also successfully typechecks. This property and the main theorem of [4] (see Sect.6) imply the non-interference.

[10] Note that the addition can be performed 'on-the-fly' in the course of the translation and therefore is not a part of the formula syntax.

Please recall that like [4], we assume that functions $\mathsf{classAnalysis}$, $\mathsf{excAnalysis}$, nbLocs, nbArgs, $\mathsf{Handler}$ are correct.

Theorem 4.1 (typechecking and verifiability) *Let P be a Java bytecode program, $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ a desired security policy, and cdr a control dependence regions structure satisfying SOAP. Let TR be the translation defined in Sect. 3 that adds base logic annotations to P. For every security environment family $\{\mathsf{se}_i : PP \to S\}_{i \in S}$ and a family of functions $\{\mathsf{st}_i : PP \to S^*\}_{i \in S}$ such that $st_i(0) = \varepsilon$ for all i,*

- $\Rightarrow$ *if the annotated program $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly then P with the policy $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable,*
- $\Leftarrow$ *if the program P with the policy $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable with se, st, and all $Q_i(c0, c)$ in QNI_m in (1) on page 10 are true then the annotated program $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ verifies correctly.*

Proof We present here a sketch of the proof only.

($\Rightarrow$) Suppose that the annotated program $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ 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 $\mathsf{Policies}_\Gamma(m)$. We need to verify that the condition in Def. 2.1 is fulfilled. Let sgn be a signature corresponding to security level s. We take se and st as above. $\mathsf{st}_s(0) = \varepsilon$ is guaranteed by the way gst is initialised. Since P verifies correctly the formula $N(i)(s)$ holds for every i. The conditions (1)–(2) from Def. 2.1 are guaranteed by the fact that all the typing rules are faithfully modelled in the logic. Let us see it for $\mathtt{inst}(i) = \mathtt{ifeq}\ j$. In other cases the proof is similar.

ifeq As the instruction at i is $\mathtt{ifeq}\ j_0$ we must only ensure condition (1) of Def. 2.1. In case of the $\mathtt{ifeq}$ instruction, the body of the formula $N(i)(s)$ is $(i \mapsto^e j_0 ==> (\mathsf{Reg}^{\mathtt{ifeq},s}(j_0)))$. This formula ensures that in case $j = j_0$ or $j = i+1$ the formula $\mathsf{Reg}^{\mathtt{ifeq},s}(j_0)$ holds. This ensures that the check of the premises of the rule from Figure 1 takes place indeed for the instruction $\mathtt{ifeq}$. Then, as the first \forall subformula of $\mathsf{Reg}^{\mathtt{ifeq},s}(j_0)$ holds, we obtain $\forall j' \in \mathtt{region}(i, \emptyset),\ k \le se(j')$ as k is identified with $gst[s][i][\mathsf{cntr}]$. The second and the third \forall subformula of $\mathsf{Reg}^{\mathtt{ifeq},s}(j_0)$ ensure that $\mathsf{lift}_k(st) \sqsubseteq st(j)$ (where $j = i+1$ or j_0).

($\Leftarrow$) Suppose that the program P with the policy $(k_{\mathrm{obs}}, \mathsf{ft}, \Gamma)$ and cdr is typable with se and st. We have to ensure that each $\mathsf{QNI}_m(i)$, for i being a label in the method m, holds. As $Q_i(c0, c)$ is true, it is enough to check that each $N(i)(j)$ holds for $j \in \mathcal{S}$. Each of the $N(i)(s)$ has similar structure presented in (2). It is enough to show that one of the corresponding $\mathsf{Reg}_l^{\mathtt{inst}(i),s}(\boldsymbol{p_l})$ holds in case $i \mapsto^e j$ (or in case $i \mapsto^e$). As the method is typable, we know that $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies s$ (or $\Gamma, \mathsf{ft}, \mathtt{region}_m, \mathsf{se}, \mathsf{sgn}, i \vdash^e \mathsf{st}(i) \implies$) 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 $\mathsf{Reg}_l^{\mathtt{inst}(i),s}(\boldsymbol{p_l})$ holds. We show this in case $\mathtt{inst}(i)$ is $\mathtt{ifeq}\ j$.

- the first subformula of (3) holds as the typing rule guarantees that the property $\forall j' \in \mathtt{region}(i, \emptyset), k \le \mathsf{se}(j')$ holds,

- the second subformula of (3) holds as the typability requires that $\mathsf{lift}_k(\mathsf{st}) \sqsubseteq \mathsf{st}(j)$, where st is $st(i)$ without its top element,
- the third subformula of (3) holds as the $\mathsf{st}(j)$ 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. □

4.1 Proof of stability

Theorem 4.3 below states that we can safely extend the specifications so that the non-interference property is preserved. More precisely, it allows to mix the specifications that result from our translation with specifications that come from other sources (e.g. are written by hand).

Definition 4.2 *(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 Sect. 3 is set.

Theorem 4.3 (stability) *Let P' be a specificational extension of* $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ *that does not conflict with* $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$*. 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 $\mathsf{TR}(P, k_{\mathrm{obs}}, \mathsf{ft}, \Gamma, \mathsf{cdr}, \mathsf{se}, \mathsf{st})$ then the values of all the variables used in the translation are the same. In this light, the logical values of the formulae are the same as in case there are no additional specifications. Hence we obtain the non-interference for P. □

5 Discussion of the solution

Declassification Our translation modifies the local assertion table Q_m so that each typing rule is checked right before the instruction instance it concerns. Note that the logic, as presented in Sect. 2, allows to change the state of the ghost variables by means of the local instruction table Ins_m. 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 entries of $gsgn$ and gst arrays that correspond to i. We can exploit a `set` instruction in $\mathsf{Ins}_m(i)$ to change $gsgn$ and gst right before the instruction that requires the declassification and revert it back right before the next one. In this way we obtain a clear declassification management mechanism — declassification is present when the `set` instructions manipulate the mentioned above arrays.

In fact, the presented method can also be applied to many other type systems which are information flow sensitive — the flow of the information is simply traced by the ghost variables. In essence, the practice of using the ghost variables in programs specified in JML is in many cases such that they serve as a method to provide an ad hoc information flow typing system.

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.

Design choices The primary goal of the design choices we took here was to find a way to express a system which ensures the non-interference property in terms of the BML formulae. The main challenge here was to connect the flow of the data with the first order formulae available in the language. We decided to simulate in ghost variables the operation of the type checking.

Another possibility would be to use ghost variables to simulate an alternative operation of the program in a flavour similar to the approaches [3,6,13]. However, the operation on the variables would use the control flow of the original program and it is not clear if it is possible to express the non-interference in this way.

The formulae we generate in our approach fall easily within the class of formulae with three predicates $\leq, <$ and $=$ and addition. This kind of arithmetic is decidable according to the classical result by Presburger (see e.g. [9]). This class is of course more powerful and the formulae could take up, roughly, the form of $\exists \mathsf{se}, \mathsf{st} \forall \ldots$ where se is the security environment and st is the abstract stack. In this way, we would not need to rely on some external source to supply the arrays and the decision procedure for the first-order logic would infer the typing for the program. However, one cannot quantify in BML so that the quantification ranges over several different `assert` formulae. Therefore, this approach is not available directly.

This obstacle could be overcome with the help of the observation that the satisfiability of the formulae does not depend on the values of the source code variables and the control flow of the programme. This lets us to store the conjunction of all the formulae in the method precondition R_m. That, however, would make the implementation of the declassification more involved. In this framework, the declassification must be implemented by a modification of the formulae themselves instead of the modification of the data they operate on.

Future work Currently, it is rather difficult to present a single succinct example of how the translation works as the result of the translation is rather complicated. At the momenta tool to transform the inferences in the information flow type system to BML using the translation is under the development.

Acknowledgement

The authors thank the anonymous referees for all the comments which helped to improve the paper.

References

[1] Amtoft, T., S. Bandhakavi and A. Banerjee, *A logic for information flow in object-oriented programs*, SIGPLAN Not. **41** (2006), pp. 91–102, an extended version in the report KSU CIS-TR-2005-1.

[2] Appel, A. W. and A. P. Felty, *A semantic model of types and machine instructions for proof-carrying code*, in: *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2000), pp. 243–253.

[3] Barthe, G., P. R. D'Argenio and T. Rezk, *Secure information flow by self-composition*, in: *CSFW '04: Proceedings of the 17th IEEE workshop on Computer Security Foundations* (2004), p. 100.

[4] Barthe, G., D. Pichardie and T. Rezk, *A certified lightweight non-interference Java bytecode verifier*, Technical report, INRIA (2006).

[5] Barthe, G., D. Pichardie and T. Rezk, *A Certified Lightweight Non-Interference Java Bytecode Verifier*, in: *Proc. of 16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science **4421** (2007), pp. 125–140.

[6] Benton, N., *Simple relational correctness proofs for static analyses and program transformations*, in: *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (2004), pp. 14–25.

[7] Beringer, L. and M. Hofmann, *A bytecode logic for JML and types*, in: *Asian Programming Languages and Systems Symposium*, LNCS (2006), pp. 389–405.
URL http://www.tcs.informatik.uni-muenchen.de/~beringer/

[8] Beringer, L. and M. Hofmann, *Secure information flow and program logics*, in: *CSF '07: Proceedings of the 20th IEEE Computer Security Foundations Symposium* (2007), pp. 233–248.

[9] Boudet, A. and H. Comon, *Diophantine equations, Presburger arithmetic and finite automata*, in: *CAAP '96: Proceedings of the 21st International Colloquium on Trees in Algebra and Programming* (1996), pp. 30–43.

[10] Burdy, L., Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino and E. Poll, *An overview of JML tools and applications*, in: T. Arts and W. Fokkink, editors, *Workshop on Formal Methods for Industrial Critical Systems*, ENTCS **80** (2003), pp. 73–89.

[11] Burdy, L., M. Huisman and M. Pavlova, *Preliminary design of BML: A behavioral interface specification language for Java bytecode*, in: *Fundamental Approaches to Software Engineering (FASE 2007)*, Lecture Notes in Computer Science **4422** (2007), pp. 215–229.

[12] Consortium, M., *Deliverable 3.1: Bytecode specification language and program logic* (2006), available online from http://mobius.inria.fr.

[13] Hahnle, R., J. Pan, P. Rummer and D. Walter, *Integration of a security type system into a program logic*, Theoretical Computer Science **402** (2008), pp. 172–189.

[14] Hennessy, M., *The security pi-calculus and non-interference*, Journal of Logic and Algebraic Programming **63** (2004), pp. 3–34.

[15] Honda, K. and N. Yoshida, *Noninterference through flow analysis*, Journal of Functional Programming **15** (2005), pp. 293–349.

[16] Hristova, K., T. Rothamel, Y. A. Liu and S. D. Stoller, *Efficient type inference for secure information flow*, in: *PLAS '06: Proceedings of the 2006 workshop on Programming languages and analysis for security* (2006), pp. 85–94.

[17] Hunt, S. and D. Sands, *On flow-sensitive security types*, in: *Principles of Programming Languages* (2006).
URL http://mobius.inria.fr/twiki/pub/Publications/WebHome/Hunt-Sands-POPL06.pdf

[18] Jacobs, B. and E. Poll, *A logic for the Java Modeling Language JML*, in: H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, LNCS **2029** (2001), pp. 284–299.

[19] Leavens, G., A. Baker and C. Ruby, *Preliminary design of JML: A behavioral interface specification language for Java*, Technical Report TR 98-06y, Iowa State University (1998), (revised since then 2004).

[20] Leavens, G., E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok and J. Kiniry, "JML Reference Manual," (2005), in Progress. Department of Computer Science, Iowa State Univer sity. Available from http://www.jmlspecs.org.

[21] Sabelfeld, A. and A. Myers, *Language-Based Information-Flow Security*, IEEE Journal on Selected Areas in Communication **21** (2003), pp. 5–19.