Privacy-Sensitive Information Flow with JML

Guillaume Dufay[1], Amy Felty[1], and Stan Matwin[1,2]

[1] SITE, University of Ottawa. Ottawa, Ontario K1N 6N5, Canada
[2] Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland
{gdufay,afelty,stan}@site.uottawa.ca

Abstract. In today's society, people have very little control over what kinds of personal data are collected and stored by various agencies in both the private and public sectors. We describe an approach to addressing this problem that allows individuals to specify constraints on the way their own data is used. Our solution uses formal methods to allow developers of software that processes personal data to provide assurances that the software meets the specified privacy constraints. In the domain of privacy, it is often not sufficient to express properties of interest as a relation between the input and output of a program as is done for general program correctness. Here we consider a stronger class of properties that allows us to express constraints on information flow. In particular, we can express that an algorithm does not leak any information from particular "sensitive" values. We describe a general methodology for expressing this kind of information flow property as Hoare-style program verification judgments. We begin with the Java Modelling Language (JML), which is a behavioral interface specification language designed for Java, and we extend the language to include new concepts and keywords for expressing such properties. We use the Krakatoa tool which starts from JML-annotated Java programs, generates proof obligations in the Coq Proof Assistant, and helps to automate their proofs. We extend the Krakatoa tool to understand our extensions to JML and to generate the new form of required proof obligations. We illustrate our method on several data mining algorithms implemented in Java.

1 Introduction

Privacy is one of the main concerns expressed about modern computing, especially in the Internet context. People and groups are concerned by the practice of gathering information without explicitly informing the individuals that data about them is being collected. Oftentimes, even when people are aware that their information is being collected, it is used for purposes other than the ones stated at collection time. The last concern is further aggravated by the power of modern database and data mining operations which allow inferring, from combined data sets, knowledge of which the person is not aware, and would have never consented to generating and disseminating. People have no ownership of their own data: it is not easy for someone to exclude themselves from, e.g. direct marketing campaigns, where the targeted individuals are selected by data mining models.

One of the main concepts that has emerged from research on societal and legal aspects of privacy is the idea of Use Limitation Principle (ULP). That principle states that the data should be used only for the explicit purpose for which it has been collected [15]. Our work addresses this question from the technical standpoint. We provide tool support for verifying that this principle is indeed upheld by organizations that perform data mining operations on personal data.

In our setting, users can express individual preferences about what can and cannot be done with their data. We have not yet addressed the question of how users express preferences, though our approach allows any data properties that can be expressed syntactically in formal logic. Certainly, a user-friendly language, easy to handle by an average person, needs to be designed. It could initially have the form of a set of options from which an individual would make choices. We assume that an organization that writes data mining software must provide guarantees that individuals' constraints are met, and that these guarantees come in the form of formal proofs about the source code. Organizations who use the data mining software are given an executable binary. An independent agency, whose purpose is to verify that privacy constraints are met, obtains the binary from the software user, obtains the source code and proof from the software developer, checks the proof, and verifies that the binary is a compiled version of the source code. The details of the architecture just described can be found in another paper [13]. The scenario just described involves using our techniques to guard against malicious code. Our approach can also apply to a setting in which trust is not an issue, for example, within a company that wants to insure that its software release is free of privacy flaws. In this paper, we concentrate on extending the class of privacy constraints which can be handled, and providing tool support for proving these properties.

In our previous work [3, 13], we considered privacy properties that could be expressed as requirements on the input-output relation. Additionally, we showed how to incorporate constraints on operations that could potentially violate privacy by overloading the output so that a trace of such operations was evident in the result. Here we consider a stronger class of properties that allows us to express constraints on information flow. In particular, we are interested in properties that express that an algorithm does not leak any information from particular sensitive values, or that a program never writes such sensitive data to a file. Many such properties can be handled through the framework of *non-interference*. Non-interference [7] is a high-level property of programs that guarantees the absence of illegal information flow at runtime. More precisely, non-interference requires distinguishing between public input/output and sensitive input/output. A program that satisfies non-interference will be such that sensitive inputs of that program have no influence on public outputs. Thus, with the non-interference framework, it is straightforward to express the expected properties that an algorithm does not leak any sensitive information or that a value is never written into a file (considering this value as a sensitive input and the file as a public output). The examples we present in this paper include properties constraining information flow both with and without the use of non-interference.

In addition to increasing the class of privacy constraints we can express, we present a new approach to proving such properties. We start from the Weka repository of Java code which implements a variety of data mining algorithms [22]. We simplify the code somewhat, both for illustration purposes and so that we can work within the limitations of current tools that support our approach. Also, we must modify the code to include checks that the privacy constraints that we allow users to specify are met. We then annotate the code with JML (Java Modeling Language) [11] assertions, which express Hoare-style [9] preconditions, postconditions, loop invariants, etc. We use the Krakatoa [12] tool to generate proof obligations in Coq [14] and to partially automate their proofs. Successful completion of these proof obligations ensures that the Java code satisfies its JML specifications. The new approach has two advantages over our old approach [3, 13]. First, we work directly with Java programs. Previously, we started with Java code, and translated it to the ML-like language used in Coq. Second, we hope that using tools engineered for program verification will improve the ability to automate proofs in the privacy domain.

To handle the kinds of information-flow properties we are interested in, we extend the expressive power of JML. We also extend Krakatoa to generate the proof obligations required for the extra expressive power, and to help automate their proofs.

Contents of the paper. In Section 2, we present the tools used in our approach to building proofs of privacy constraints of data mining algorithms. In Section 3, we present our first example; it is a simple one which serves to illustrate our approach using JML and Krakatoa. In Section 4, we discuss a nearest neighbor classification algorithm, and present JML annotations which guarantee that the result value of a data-mining program does not reveal any sensitive information by constraining it to a set of public values. In Section 5, we discuss how non-interference can be used to handle a larger class of privacy-sensitive properties, and how we extend JML and Krakatoa to provide support for non-interference. In Section 6, we apply these results to a Naive Bayes classification algorithm. Finally, in Section 7, we conclude and discuss future work.

2 Tools

First, we will introduce Weka, a Java library of data-mining algorithms, and the notion of classifiers; then, the JML assertion language in which privacy properties are expressed; and finally, the Krakatoa tool, that we use to verify JML-annotated Java programs.

2.1 Weka and Classifiers

Since we target data mining software, we have decided to apply our approach to selected classification modules of Weka (Waikato Environment for Knowledge Analysis) [22]. Weka is an open-source library of data mining algorithms, written

in Java, providing a rich set of mining functions, data preprocessing operations, evaluation procedures, and GUIs. Weka has become a tool of choice, commonly used in the data mining community. Classification is one of the basic data mining tasks, and Weka provides Java implementations of all the main classification algorithms. In this paper, we illustrate our approach to privacy-sensitive information flow with two commonly used classification tools, the so called Nearest Neighbor and Naive Bayes classifiers.

The classification task can be defined as follows: given a finite training set $T = \{\langle x_i, y_i \rangle | x_i \in D, y_i \in C\}$, where D denotes the data (set of instances), and C denotes the set of classes, find a function $c : D \mapsto C$ such that for each pair $\langle x', y' \rangle \in T, y' = c(x')$, and furthermore, c will correctly classify unseen examples (i.e. examples that will only arrive in the future, and as such cannot be included in the training set T.) c is referred to as a classifier, and the task of finding c is known as learning c from T, or —alternatively— as the classifier induction task. Usually, $D = A_1 \times \ldots \times A_n$, where $A_i, i = 1, \ldots, n$, is an attribute domain. A_i's are either sets of discrete (nominal) values, or subsets of $\mathbb{R}$. Each $x_i \in D$ can therefore be seen as $x_i = a_{i,1}, \ldots, a_{i,n}$, where $a_{i,j} \in A_j$.

2.2 The Java Modeling Language

The Java Modeling Language [11] (JML) is a behavioral interface specification language designed for Java. It relies on the design by contract approach [16] to guarantee that a program satisfies its specification during runtime. These specifications are given as annotations of the Java source file. More precisely, they are included as special Java comments, either after the symbols //@ or enclosed between /*@ and @*/. For example, the general schema for the annotation of a method is the following:

```
/*@ behavior
  @   requires <precondition>;
  @   modifiable <modified fields and variables>;
  @   ensures <postcondition if no exception raised>;
  @   signals(E) <postcondition when exception E raised>; @*/
```

The underlying model is a an extension of Hoare-Floyd logic [9]: if the precondition holds at the beginning of the method call, then postconditions (with and without exceptions) will hold after the call.

Preconditions and postconditions express first-order logic statements, with a syntax following the Java syntax. Thus, they can easily be written by a programmer. The Java syntax is enriched with special keywords: `\result` and `\old(<expr>)` to denote respectively the return value of a method, and the value of a given expression before the execution of a method; and `\forall`, `\exists`, `==>` to denote respectively universal quantification, existential quantification and logical implication. If the `modifiable` clause is omitted, it means by convention that the method is side-effect free.

Apart from methods specification, it is also possible to annotate a program with class invariants (predicates on the fields of a class that hold at any time in the class) using the keyword `invariant`, loop invariants (inside the code of

a method with loops) using the keyword `loop_invariant`, and assertions (that must hold at the given point of the program) using the keyword `assert`.

Finally, when annotating a program, it might be useful to introduce new variables to keep track of certain aspects or computations. Instead of adding them to the program itself, thus adding new code, it is possible to define variables that will only be used for specification. These variables, called *ghost* variables, are defined in a JML annotation with the keyword `ghost` and assigned to a Java expression with the keyword `set`.

2.3 Krakatoa

Once a program has been annotated with JML, these annotations can be verified either during runtime (an exception will be raised if they do not hold) or statically, with a static checker or a theorem prover, given the semantics of the program. A wide range of tools can be used to achieve this goal. Among these, the LOOP Tool [21] will work with the PVS theorem prover, Jack [10] with Atelier B, or Krakatoa [12], that we chose, with Coq [14].

The Krakatoa tool provides a generic model in Coq of the Java runtime environment. Annotated Java programs are not translated directly to Coq, but to the Why [5] input language (an annotated ML-like language), without any loss of precision. Then, Krakatoa relies on static analysis and weakest preconditions calculus of the stand-alone Why tool to generate Coq proof obligations, corresponding to requirements the Java program must respect to meet its specifications. Some of these proof obligations can be discharged automatically through Coq built-in or Krakatoa provided tactics. The remaining proof obligations have to be completed manually, through the interactive proof mechanism of Coq. In some cases, preconditions or loop invariants of the annotated Java program might not be strong enough to prove the postcondition of a method and need to be modified. Proof obligations are then regenerated, but completed proofs not affected by these modifications are kept.

The successful completion of all proof obligations is sufficient to ensure that a program satisfies its specifications. However, the Why tool can also perform a final step, called validation, to embed each functional translation of the methods of Java program with its specification into a Coq term whose type corresponds to the JML specification of that method. This term can be given as a certificate of the soundness of the whole process.

3 A First Example: Joining Two Database Tables

This sections presents an example of a JML-annotated Java program. We redo the example described in [3], where the program was written and proved within Coq. This example serves to illustrate our new approach as well as compare it to the old one. The program performs a database join operation. The data from two sets (`Payroll` and `Employee`) is joined into a single set (`Combined`), ignoring the data from individuals that do not want their data to be used in a join

operation. For example, individuals with an exceptionally high salary may not want their `Payroll` information in the same record as their address and phone number. Such detailed records may contain enough information to identify them or to make them the target of certain kinds of direct-marketing campaigns. In this example, such individuals can express that they want to opt out of this operation.

The data structures for this example are standard Java classes. For instance, the payroll notion is captured by the following class that contains the employee ID `PID` it refers to, the salary, and a boolean `JoinInd` which indicates if the person who owns the data has given the permission to use the data in a join operation.

```
class Payroll {
   public int PID;
   public int Salary;
   public boolean JoinInd;
};
```

The result of the join is stored in a `Combined` class, that gathers the data from the classes `Payroll` and `Employee` (which contains, among other fields, name and `EID` which records the employee ID). We can notice at this point that the constructor for the class `Combined` is annotated with a JML specification. It prevents the creation of a `Combined` class for the users that do not allow it (the field `JoinInd` has to be true), and ensures that the field `JoinInd` is unchanged in the created class.

```
class Combined {
  public Payroll m_payroll;
  public Employee m_employee;

  /*@ public normal_behavior
    @ requires p != null && p.JoinInd == true && e.EID == p.PID;
    @ ensures m_payroll.JoinInd == p.JoinInd; @*/
  public Combined(Employee e, Payroll p) {
    m_employee = e;
    m_payroll = p;
} };
```

Note that the assertion above includes a statement that the employee ID fields of `e` and `p` are the same. We did not need this in the version in [3] since only one copy of the ID was kept in the new `Combined` record. This is a minor difference which has little effect on the proofs.

The algorithm that iterates though a set of `Payroll` records to perform a join operation is given below:

```
/*@ public normal_behavior
  @ requires Ps != null && Es != null;
  @ ensures (\forall int i; 0 <= i && i < \result.length; \result != null &&
  @    (\result[i] != null ==> \result[i].m_payroll.JoinInd == true)); @*/
public Combined[] join(Payroll[] Ps, Employee[] Es) {
  Combined tab[] = new Combined[Ps.length];

  /*@ loop_invariant
    @ 0 <= i && i <= Ps.length &&
    @ (\forall int j; 0 <= j && j < i && j < tab.length;
    @     (tab[j] != null ==> tab[j].m_payroll.JoinInd == true));
    @ decreases Ps.length-i; @*/
  for (int i=0; i < Ps.length; i++)
```

```
      if (Ps[i] != null)
        tab[i] = checkJoinIndAndfindEmployee(Ps[i], Es);
      else
        tab[i] = null;

  return tab;
}
```

The specification of this algorithm expresses the same property as in [3], but here it is expressed in JML, which uses Java-like syntax and refers directly to variables occurring in the program. The particular property that is expressed is that all data that took part in the join was in fact permitted to do so by the owners of the data. In [3], this property was expressed directly as a formula in Coq. Here, the requirements on individual methods taken together express this property. The method `checkJoinIndAndfindEmployee`, whose code is not given here, takes a single `Payroll` record `Ps[i]` and the entire list of `Employee` records `Es` as arguments. If (1) a record is found such that `Ps[i].EID` matches the employee ID value in one of the records in `Es`, and (2) `Ps[i].JoinInd` has value **`true`**, then a new `Combined` record is created and returned. Otherwise **`null`** is returned.

Note that the loop inside the `join` method had to be annotated, like any loop in the Hoare-logic formalism. Also, the method `checkJoinIndAndfindEmployee` had to be annotated with precondition, postcondition, and loop invariant since it is called by `join` and extends the results of the call to the constructor of the class `Combined`.

After going through Krakatoa, most of the generated proof obligations are automatically solved by Coq. In the JML annotations from the code above, we have omitted some dynamic type information (such as `Ps` is an instance of `Payroll[]`) that was needed to complete the proof. The fact that we had to manually insert this information is due to current limitations of Krakatoa that will be fixed in the near future. Around 100 lines of proof were entered to discharge the remaining proof obligations, which is slightly less than the length of the proofs in [3]. Although the difference is not really significant due to the limited size of the example, we believe that this approach leads to smaller proofs and to an increased confidence in the whole engineering process.

4 A Simple Data Privacy Preserving Classifier

In this section, we will describe how to enforce the value of a data-mining algorithm not to reveal any sensitive information, by constraining the output to a set of public values.

The following algorithm, the nearest neighbor classifier algorithm, has been extracted from the Weka library but, to keep proofs simpler, unwanted features for the purpose of this example have been removed (such as method calls related to the Weka graphic interface, or checks for an incorrect or incomplete data set) and data are accessed directly, not through objects. For this particular classifier, the returned value of the *class attribute* (in a data-mining context, the attribute that the classifier aims at determining) for the given instance `instance`, is the

value of the class attribute for the instance from the training set `m_Train` determined as the nearest of the given instance. The corresponding distance is calculated from the non-class attributes of both.

The specifications for this algorithm constrain the result value to be one of the class attributes (in the row `classIndex` of instances), thus preventing leaks of any other value of the dataset. This kind of specification can be used for other classifiers on a finite set of class attributes and that return an element of this set. It would be possible to also prevent a particular instance to be used in the algorithm based on the owner requirements, as done in Section 3, but it is supposed in this example that sensitive information resides in non-class attributes and that the class attribute can be public.

```
/*@ public normal_behavior
  @ requires instance != null && m_Train != null && numInstances > 0 &&
  @   instance.length == numAttributes && m_Train.length == numInstances &&
  @   (\forall int i; 0 <= i && i < numInstances;
  @      m_Train[i] != null && m_Train[i].length == numAttributes);
  @ ensures (\exists int i; 0 <= i && i < m_Train.length;
  @   \result == m_Train[i][classIndex]); @*/
public double classifyInstance(double [] instance) throws Exception {
  double dist, minDistance, classValue = 0.0;
  boolean first = true;

  buildClassifier();
  updateMinMax(instance);

  /*@ loop_invariant
    @ 0 <= i && i <= numInstances &&
    @ ((first == true && i == 0) ||
    @ (\exists int j; 0 <= j && j < i; classValue == m_Train[j][classIndex]));
    @ decreases numInstances - i; @*/
  for (int i = 0; i < numInstances; i++)  {
      dist = distance(instance, m_Train[i]);
      if (first || dist < minDistance) {
        minDistance = dist;
        classValue = m_Train[i][classIndex];
        first = false;
      }
  }

  return classValue;
}
```

Proof obligations for this algorithm do not lead to particular problems, they just follow the structure of the code and the annotations. A total of 180 lines of manually entered proof scripts is needed for `buildClassifier`, `classifyInstance` and the auxiliary functions involved such as `updateMinMax` and `distance`.

This approach leads to simple specifications and proofs for which the result is constrained to a known set. However, in cases where the set result is infinite, a stronger framework is needed, such as the one provided by non-interference.

5 Privacy Through Non-Interference

As explained in the introduction, non-interference distinguishes public inputs (resp. outputs) and sensitive inputs (resp. outputs) and prevents leaks from sensitive inputs to public outputs. For example, if we consider the input/output

variables `x` as public and `y` as sensitive, the program `x = y*2` is interferent (direct flow from `y` to `x`), whereas the program `x = y; x = 0` is not (it is impossible for an attacker to guess the value of `y` by observing `x` at the end of the execution). It is also possible to get interference through indirect information flow, for instance the following program is interferent (it is possible to guess the nullity of `y`):

```
if (y != 0) then x = 1; else x = 2;
```

Finally, interference can be observable through termination of programs (termination-sensitive) or timing leaks. For the sake of this paper, we will not consider these possibilities.

5.1 The General Framework

Non-interference can be enforced through type systems [20, 1]. However, in practice these type systems turn out to be laborious to use and they can reject obvious non-interferent programs. Instead, we prefer to follow the approach described in [2] that proceeds by self-composition of the program and that can be described using the Hoare-style logic of JML (and then integrated in the tools we have used so far to study privacy).

Self-composition proceeds by duplicating the code of a program, with two sets of inputs. Thus imperative pointer-free program $P(\boldsymbol{x}, \boldsymbol{y})$ with public input/output variables $\boldsymbol{x}$ and sensitive input/output $\boldsymbol{y}$ will be non-interferent if forall $\boldsymbol{x}_1$, $\boldsymbol{x}_2$, $\boldsymbol{y}_1$, $\boldsymbol{y}_2$ we have the following Hoare formula:

$$\{\boldsymbol{x}_1 = \boldsymbol{x}_2\}\ P(\boldsymbol{x}_1, \boldsymbol{y}_1); P(\boldsymbol{x}_2, \boldsymbol{y}_2)\ \{\boldsymbol{x}_1 = \boldsymbol{x}_2\}$$

where `;` is usual sequential composition. This formula expresses the fact that the output values of the public variables are independent from the values of the sensitive variables.

More generally, dependencies between the parameters of the program, before and after the execution, are characterized by a relation called *L-equivalence*. In the above Hoare formula, this relation is simple equality. By allowing more general L-equivalence relations between public and sensitive variables, it becomes possible to capture the notion of *declassification* [4, 18] within the same framework. Declassification allows leaks, in a controlled way, of sensitive global information to public variables. Indeed declassification would allow a data-mining algorithm to compute over sensitive variables and yield public results that do not give any specific information about any of these sensitive variables. An example of such a use is given in the example of Section 6 in the context of data-mining.

5.2 Extension of Krakatoa

In order to be able to handle non-interference with Krakatoa following the previous framework, some modifications have to be made to both JML and Krakatoa. First, we wish to distinguish pre and post-conditions related to the normal execution of the program and those related to non-interference. For this purpose, we

have introduced two optional keywords for method specifications: `requires_ni` and `ensures_ni`. Then, to define L-equivalence relations in the pre and post-conditions of the self-composed program, we need to distinguish variables for each of the two runs of the program. Therefore, we have introduced two keywords `\ni1(<var>)` and `\ni2(<var>)`. Finally, annotations inside the code are also required to exist in three variants, one for normal execution and one for each run of the program for non-interference (these annotations are not necessarily the same for each run). For example, the keywords `loop_invariant_ni1` and `loop_invariant_ni2` are available to distinguish loop invariants to be used for each run of the program.

Krakatoa is modified to recognize these new keywords. When a method is annotated with non-interference specifications, it generates the code for the self-composed method with the corresponding specifications and the appropriate variable names. A particular case appears for method invocations inside a method body. Indeed, the non-interference results from the invoked method will only be available in the second copy of the self-composed code (the two runs must have occurred). In addition, it is necessary to modify the program, with ghost variables, to keep track of the invoked method parameters and result values of the first run (they can be modified later on by assignment and thus would not be available anymore). These values will be used as values of the variables of the first run, in the non-interference results of the invoked method of the second run.

6 Non-Interference for the Naive Bayes Classifier

In this section, we will illustrate how the idea of non-interference can be applied on a data-mining algorithm, the naive Bayes classifier, to express a privacy property.

The Naive Bayes classifier predicts the class of an instance $x = a_1, \ldots, a_n$ (`mTrain[i]` in the code below) as

$$c(x) = argmax_{c_j \in C} P(c_j) \prod P(a_i|c_j)$$

i.e. the class of x is obtained by estimating the probabilities of all classes for given attribute values of x. These estimates, known as *priors*, are known from the training set. Probability estimates are approximated by counting frequencies of different classes for given attribute values. The training data needs therefore to be summarized in a table (the variable `probs` in the code below), which keeps the count of the number of instances with specific attribute values for each class.

From a data privacy point of view, we will assume that some people do not want their data (more precisely the corresponding instances in the training set T) to be used by the classifier for this particular class. Having one's data used by a classifier means that this particular individual stands out in T, and can be targeted (by the use of so called data drilling operations) in marketing campaigns, sampling routines, etc. It might be reasonable to object to this.

Thus, the output class value of the algorithm, that is a public result, should not depend on these sensitive data. The entire set of training instances can not be considered as fully sensitive data since some information has to be gained from it to classify the instance; it can not be considered as public data either due to the restrictions given above. Rather, the training set should be considered as sensitive data with part of these data (instances that are allowed by their owner to be used in a classifier algorithm) being declassified for non-interference. Then, the L-equivalence relation for this algorithm will expect:

- from the input `m_Train`, the training set of instances, an array of `m_NumInstances` instances, to be such that the corresponding duplicated variables for self-composition `\ni1(m_Train)` and `\ni2(m_Train)` agree on the values that can be used in the classifier algorithm;
- from the public inputs `instance`, the instance to classify, to be such that the duplicated variables are equal;
- from the public input `m_inst_Allow`, an array of `m_NumInstances` booleans that express for a given index whether the instance at the corresponding index in `m_Train` can be used in the classifier, to be such that the duplicated variables are equal;
- from the public output `\result`, an array of `m_NumClassValues` probabilities, to be such that the duplicated variables are equal.

More formally, the JML specification for the naive Bayes algorithm is:

```
/*@ public normal_behavior
  @ requires_ni (\forall int i; i <= 0 && i < m_NumInstances;
  @   ((\ni1(m_inst_Allow)[i] == true) ==>
  @     (\forall int j; j <= 0 && j < m_numAttributes;
  @       \ni1(m_Train)[i][j] == \ni2(m_Train)[i][j]))) &&
  @   (\ni1(instance)[i] == \ni2(instance)[i]) &&
  @   (\ni1(m_inst_Allow)[i] == \ni2(m_inst_Allow)[i]);
  @ ensures_ni (\forall int i; 0 <= i && i < m_NumClassValues;
  @   \ni1(\result[i]) == \ni2(\result[i])); @*/
```

The structure of the code for this classifier relies on two main methods: `buildClassifier` that initializes the classifier with the training set data, and `distributionForInstance` that uses the previously built classifier to classify a given instance. The `buildClassifier` method must ensure that two training sets that verify the conditions given above will generate equal probability estimators:

```
/*@ public normal_behavior
  @ requires_ni (\forall int i; i <= 0 && i < m_NumInstances;
  @   (\ni1(m_inst_Allow)[i] == \ni2(m_inst_Allow)[i]) &&
  @   ((\ni1(m_inst_Allow)[i] == true) ==>
  @     (\forall int j; j <= 0 && j < m_numAttributes;
  @       \ni1(m_Train)[i][j] == \ni2(m_Train)[i][j])));
  @ ensures_ni instance != null && instance.length == m_NumAttributes &&
  @   (\forall int i; 0 <= i && i < m_NumClassValues;
  @     \ni1(m_ClassDistribution).getProbability(i) ==
  @     \ni2(m_ClassDistribution).getProbability(i)) &&
  @   (\forall int attIndex; 0 <= attIndex && attIndex < m_NumAttributes;
  @     (\forall int i; 0 <= i && i < m_NumClassValues;
  @       (\forall int j; 0 <= j && j < m_NumValues[attIndex];
  @         \ni1(m_Distributions)[attIndex][i].getProbability(j) ==
  @         \ni2(m_Distributions)[attIndex][i].getProbability(j)))); @*/
```

```
public void buildClassifier() {
  m_ClassDistribution = new DiscreteEstimator(m_NumClassValues, true);

  for (int attIndex = 0; attIndex < m_NumAttributes; attIndex++)
    for (int j = 0; j < m_NumClassValues; j++) {
      m_Distributions[attIndex][j] =
        new DiscreteEstimator(m_NumValues[attIndex], true);
    }
  for (int i = 0; i < m_NumInstances; i++) {
    if (m_inst_Allow[i] ==  true) {
      for (int attIndex = 0; attIndex < m_NumAttributes; attIndex++) {
        int distr_idx = m_Train[i][m_ClassIndex];
        m_Distributions[attIndex][distr_idx].
                  addValue(m_Train[i][attIndex]);
      }
    m_ClassDistribution.addValue(inst[m_ClassIndex]);
  }
}
```

The `distributionForInstance` method will now compute the probability distribution `probs` (an array of `m_NumClassValues` values), such that `probs[i]` is equal to the probability for the instance `instance` to be classified as the i^{th} class value. Based on the post-conditions of the previous method, the following specifications will ensure for non-interference that two equal probability distributions will be generated at the end of self-composition.

```
/*@ public normal_behavior
  @ requires_ni (\forall int i; i <= 0 && i < m_NumInstances;
  @   (\ni1(instance)[i] == \ni2(instance)[i])) &&
  @   <ensures_ni of buildClassifier>;
  @ ensures_ni (\forall int i; 0 <= i && i < m_NumClassValues;
  @   \ni1(\result[i]) == \ni2(\result[i])); @*/
public void distributionForInstance(double[] instance) {

  /*@ loop_invariant_ni1
    @   0 <= \ni1(j) && \ni1(j) <= \ni1(m_NumClassValues) &&
    @   (\forall int i; 0 <= i && i < \ni1(j);
    @     \ni1(probs[i]) == \ni1(m_ClassDistribution).getProbability(i) &&
    @     \ni1(probs_save[0][i]) == \ni1(probs[i]));
    @ decreases \ni1(m_NumClassValues) - \ni1(j);
    @ loop_invariant_ni2
    @   0 <= \ni2(j) && \ni2(j) <= \ni2(m_NumClassValues) &&
    @   (\forall int i; 0 <= i && i < \ni2(j);
    @     \ni2(probs[i]) == \ni1(probs_save[0][i]) &&
    @     \ni2(probs_save[0][i]) == \ni1(probs_save[0][i]));
    @ decreases \ni2(m_NumClassValues) - \ni2(j); @*/
  for (int j = 0; j < m_NumClassValues; j++) {
    probs[j] = m_ClassDistribution.getProbability(j);
    //@ set prob_save[0][j] = probs[j];
  }

  /*@ loop_invariant_ni1 <...> ;
    @ loop_invariant_ni2 <...> ; @*/
  for (int attIndex = 0; attIndex < m_NumAttributes; attIndex++) {

        /*@ loop_invariant_ni1
          @   0 <= \ni1(j) && \ni1(j) <= \ni1(m_NumClassValues) &&
          @   (\forall int i; 0 <= i && i < \ni1(j);
          @     \ni1(probs[i]) == \ni1(probs_save[attIndex][i]) *
          @       \ni1(m_Distributions)[0][i].getProbability(\ni1(instance[attIndex])) &&
          @       \ni1(probs_save[attIndex+1][i]) == \ni1(probs[i])) &&
          @   (\forall int i; \ni1(j) <= i && i < \ni1(m_NumClassValues) ;
          @     \ni1(probs[i]) == \ni1(probs_save[attIndex][i]));
          @ loop_invariant_ni2
          @   0 <= \ni2(j) && \ni2(j) <= \ni2(m_NumClassValues) &&
          @   (\forall int i; 0 <= i && i < \ni2(j);
```

```
          @    \ni2(probs[i]) == \ni1(probs[i]) &&
          @    \ni2(probs_save[attIndex+1][i]) == \ni1(probs_save[attIndex+1][i])) &&
          @   (\forall int i; \ni2(j) <= i && i < \ni2(m_NumClassValues) ;
          @    \ni2(probs[i]) == \ni1(probs_save[attIndex][i])); @*/
    for (int j = 0; j < m_NumClassValues; j++) {
      probs[j] *= m_Distributions[attIndex][j].getProbability(instance[attIndex]);
      //@ set prob_save[attIndex+1][j] = probs[j];
    }
  }
  return(probs);
}
```

Non-interference specifications have been given for all methods involved in this example (including the three methods from the class `DiscreteEstimator` devoted to representing probability estimators). Note however that the specifications given above are not complete due to the lack space in the sense that some loop invariants are not shown (they are similar to the ones given) and that Krakatoa requires some additional information, also not shown, about bound limits for arrays, types and non-nullity of objects, loop variants (the index used in the loop) and modified objects.

The resolution of generated proof obligations proceeds by matching values of the second run to the corresponding values of the first run. To do so, it is necessary to keep track of the successive values assigned, which is the role in the specifications of extra variables (for example the array `probs_save` to keep track of values of the variable `probs` inside the loop), that can be declared as JML ghost variables. Proofs are not yet completed due to the presence of method invocations involving arrays. Although we are able to use non-interference results in those cases, we are currently working on automatically generating assertions related to the extra ghost variables (arrays) needed to store parameters and result values of the invoked method. The **`if`** condition inside one loop does not cause any particular problems. The specifications just require the use of logical implication to reason about the value of the test, as was done in Section 3.

Although statements of generated proof obligations can be very long due to the various loops involved (one statement of a proof obligation is over 700 lines long), individual subgoal statements are very concise and the required total length of proof script that had to be given manually for the `distributionForInstance` method and methods from the class `DiscreteEstimator` is about 200 lines.

7 Conclusion

We have presented several ways to enforce privacy-sensitive information flow with JML, which we have illustrated on data-mining algorithms. We first extended results from a previous paper to integrate them into the JML framework. We then proposed a way to prevent leaks from sensitive variables when the set of possible results is finite. Finally, we applied the framework of non-interference to provide a stronger means to express and enforce privacy properties. To do so, we extended JML specifications with new specific keywords, but we kept the underlying Hoare-Floyd style verification mechanism. We have completed all proofs

in Coq of the generated proof obligations for the first two examples. For the more complex example which uses non-interference, we provided specifications for all methods, but proof obligations for one method could not be completed due to current limitations of the tools. However, we completed proofs for all others methods, thus providing a proof of concept of our methodology.

Related work. One of the most comprehensive tools related to information flow for Java is JFlow [17]. This tool acts as a compiler to statically and dynamically check programs. It relies on a concept of security levels for variables, which is not sufficient for our purpose, i.e. to catch the kind of declassification we are dealing with. Concerning non-interference, although research in this domain is very active (see [19] for a survey), most of the work done remains theoretical. On the practical side, applications of non-interferent programs are currently limited to security issues in smart cards. [6] is one of the more advanced contributions in this area, using JFlow and Esc/Java. [8] is another example of work exploiting JFlow information-flow policy to address privacy. Although this work does not address data mining in particular, it may be possible to integrate this kind of approach with ours, when dealing with simpler declassification properties, to improve the scope of privacy concerns that can be enforced. Our use of non-interference for JML to express privacy of data-mining algorithms is a novel, promising application area.

Further work. Future development of our work will aim at first to address the limitations explained in Section 6 concerning the inclusion of non-interference results from called methods inside the proofs, and scaling the approach to more complex Java features and algorithms of the Weka library. Another interesting development would be to automatically generate loop invariants, which can be tedious to write, but are needed for proofs of non-interference. Indeed, the invariant for the two copies of the code of a self-composed program follow the same pattern, and it can be determined statically which variable of the second run corresponds to which variable of the first.

Acknowledgments

The authors acknowledge the support of the Natural Sciences and Engineering Research Council of Canada, and of the Ontario Centres of Excellence, Inc (CITO Division).

References

1. Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, page 253. IEEE Computer Society, 2002.
2. Gilles Barthe, Pedro D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 100–114. IEEE Computer Society, 2004.

3. Amy Felty and Stan Matwin. Privacy-oriented data mining by proof checking. In *Proceedings of the Sixth European Conference on Principles of Data Mining and Knowledge Discovery*, volume 2431 of *LNCS*. Springer-Verlag, August 2002.
4. E. Ferrari, P. Samarati, E. Bertino, and S. Jajodia. Providing flexibility in information flow control for object-oriented systems. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, May 1997.
5. Jean-Christophe Filliâtre. Why: A multi-language multi-prover verification tool. Research Report 1366, LRI, Université Paris Sud, March 2003.
6. Pablo Giambiagi and Mads Dam. Verification of confidentiality properties for Java Card applets. Manuscript, 2003.
7. Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proceedings of 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982.
8. Katia Hayati and Martín Abadi. Language-based enforcement of privacy policies. In *Proceedings of Privacy Enhancing Technologies Workshop (PET 2004)*, LNCS. Springer-Verlag, 2004.
9. C. A. R. Hoare. An axiomatic basis for computer programming. *Commununications of ACM*, 12(10):576–580, 1969.
10. JACK: Java Applet Correctness Kit. `http://www-sop.inria.fr/everest/soft/Jack/jack.html`.
11. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999. `http://www.jmlspecs.org`.
12. Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The Krakatoa Tool for Certification of Java/JavaCard Programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):86–106, 2004.
13. Stan Matwin, Amy Felty, István Hernádvölgyi, and Venanzio Capretta. Privacy in data mining using formal methods. In *Proceedings of the Seventh International Conference on Typed Lambda Calculi and Applications*, volume 2841 of *LNCS*. Springer-Verlag, 2005.
14. Coq Development Team. The Coq Proof Assistant Reference Manual – Version 8.0. `http://coq.inria.fr/doc/`, 2004.
15. Information and Privacy Commissioner/Ontario. Data mining: Staking a claim on your privacy. `http://www.ipc.on.ca/scripts/index.asp?action=31&P_ID=11387`, January 1998.
16. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2 edition, 1997.
17. Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Symposium on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.
18. Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 172–186. IEEE Computer Society, 2004.
19. Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, January 2003.
20. Geoffrey Smith. A new type system for secure information flow. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop*, page 115. IEEE Computer Society, 2001.
21. The LOOP Project. `http://www.cs.kun.nl/sos/research/loop/`.
22. I.H. Witten and E. Frank. *Data Mining*. Morgan Kaufmann, 2000.