Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 234 (2009) 37–54
www.elsevier.com/locate/entcs

Rewrite Based Specification of Access Control Policies[1]

Horatiu Cirstea,
Pierre-Etienne Moreau and Anderson Santana de Oliveira

Nancy Université & INRIA & UFRN & LORIA

Abstract

Data protection within information systems is one of the main concerns in computer systems security and different access control policies can be used to specify the access requests that should be granted or denied. These access control mechanisms should guarantee that information can be accessed only by authorized users and thus prevent all information leakage. We propose a methodology for specifying and implementing access control policies using the rewrite based framework Tom. This approach allows us to check that any reachable state obtained following a granted access in the implementation satisfies the policy specification. We show that when security levels are not totally ordered some information leakage can be detected.

Keywords: Access Control Policies, Term Rewriting, Model-Checking, Information Flow

1 Introduction

The main principle in information flow policies is to avoid information from higher security levels to be stored in resources with lower security levels. More generally, security-sensitive information should not be accessible to lower-privileged subjects. In this paper we will consider a "resource" (also called object) as an abstract concept, defined informally as a container of information. Subjects can be understood as human users, but it is more meaningful to think of them as programs. Typical examples of resources are files and directories in an operating system. For instance, an information flow policy states that a subject must be denied to copy a sensitive file, such as /etc/shadow in a UNIX file system (which contains all encrypted passwords), to a file with less restrictive rights, even if a program run by root is trying to perform such operation.

The mechanisms employed in order to enforce this kind of policy involve discretionary access control, and a system-wide mandatory policy, stipulating that

[1] This work has been partially funded by the ANR SSURF Project.

doi:10.1016/j.entcs.2009.02.071

resources can be owned by individual users, but that the system is able to override any user permissions, under certain circumstances and more precisely, when principals try to violate the information flow property. This is the approach followed by Bell and LaPadula (BLP) in their seminal work on mandatory access control [2]. Nevertheless, it is very hard to prove that a system starting on a valid configuration will reach only secure or valid states by analyzing the access control rules. This is in part due to the fact that typical information flow properties are expressed in terms of desirable read-write and write-read traces. Roughly speaking, since systems are abstracted as Turing machines, checking an information flow property corresponds to checking whether the machine enters a given state, which is in general undecidable.

In this paper, we show that it is not enough to assume that the security policy specifying the authorized accesses is correctly defined based on these traces, because some models can allow some sequences of requests that will indirectly violate the policy. We propose a method to make explicit such flows in a given state of the system. We use rewrite rules both to describe the access control policy and to explore information flow over the reachable states of a system. We propose an analysis technique that can automatically identify information leakages which may invalidate an implementation of an information flow policy. We use as support language the Tom framework, a language extension which adds new matching primitives to existing imperative languages. It allows users to write concise specifications and efficient validation methods possibly using features of the host language.

We apply our technique on two well-known models in the computer security literature. We consider the security model proposed by McLean in [8], whose intention is to generalize the Bell and LaPadula model for dealing with joint access. We show that confidentiality is compromised when some simplifying assumptions made in order to relate the general framework for joint access with BLP are taken.

We extend our previous work [9] by separating the verification procedure from the intrinsic characteristics of the policy model. Moreover, the verification technique used in the current approach is more generic in the sense that checking other policies can be done if the specification and implementation of the respective policy are provided (as a Tom class). In [9], the exploration of the search space was influenced by the analyzed policy and more precisely the access requests are performed following an order deduced from the policy specification. Consequently, other permutations of requests that would lead to different states are discarded and thus, the algorithm was not complete. The present approach provides a complete procedure which can make use of different optimizations to reduce the search space. Most of these optimizations are independent of the security model being checked (provided this model's properties depend on the set of current accesses), whereas they were hard coded in the former procedure. As we will see in Section 5.4, some natural assumptions on the policy model make it possible to implement more powerful optimizations of the search space.

The rest of the paper is organized as follows. Section 2 briefly presents the BLP and the McLean models for multilevel security. Section 3 introduces the Tom system

and Section 4 presents the encoding of the two policies in Tom. Section 5 shows how these models can be checked and proposes some optimizations used to cut the search space to speed-up the execution of the verification process. In Section 6 we discuss some related works and conclude the paper.

2 Lattice-based security models

We focus here on two lattice-based security models [14] which impose constraints on the way information can flow from one security level in the lattice to another.

2.1 The Bell and LaPadula Model

Bell and LaPadula have formalized the concept of mandatory access controls [2,3] originally targeted to the military domain. The essence of the BLP model is to augment discretionary access controls, represented in an access control matrix, with mandatory access controls to enforce information flow policies. Without loss of generality, we focus only on the mandatory part in this paper, which is really specific to the BLP model. Therefore we do not describe how the contents of the access control matrix can be modified by subjects, but we consider that the current state of the system is exactly the set of access tuples the matrix contains. Also, the notation and precise formulation of the rules of BLP used here are substantially different from those of the original Bell-LaPadula work, since we use a more up-to-date interpretation of these concepts in line with our formalism.

In BLP the relation among the security levels from the set L is a total ordering $\succeq$ whose most common classes are: *top secret* $\succeq$ *secret* $\succeq$ *confidential* $\succeq$ *unclassified*. In the following we consider a set of subjects S and a set of objects O together with two functions $f_s : S \to L$ and $f_o : O \to L$ which give the corresponding security clearances for subjects and classifications for objects.

If we consider a set A of actions containing $\{read, write\}$ and a list M of tuples of the form $m(S, O, A)$ representing the actions that have been performed so far in the system, then the mandatory rules for the BLP model are:

$$(1) \quad \forall s \in S \;\; \forall o \in O \quad m(s, o, read) \in M \Rightarrow f_s(s) \succeq f_o(o)$$

$$(2) \quad \forall s \in S \;\; \forall o_1, o_2 \in O \quad m(s, o_1, read) \in M \wedge m(s, o_2, write) \in M \Rightarrow f_o(o_2) \succeq f_o(o_1)$$

The properties (1) and (2) are known respectively as the *simple secure property* and the $\star$*-secure property*. The first property states that if a subject performs a read access on an object then the level of the subject dominates that of the object. The second one allows a user reading from an object o_1 to write only objects whose levels dominate that of o_1. Intuitively, this means that information read from one level cannot be disclosed towards lower level objects. We should point out that write access is interpreted here as write only.

From a practical point of view, the $\star$-property would not be applied to human users, but rather to programs. Human users are trusted not to leak information. A secret user can write an unclassified document because we assume that he or she will put only unclassified information in it. Programs, on the other hand, are not trusted because they can embed Trojan horses. The $\star$-property prohibits a program running at the secret level from writing to unclassified objects, even if it is permitted to do so by discretionary access controls.

2.2 The McLean Model

The algebra of security of McLean [8] is seen as a generalization of the BLP model. One of the new concepts introduced in this approach is *joint access*, which is the capability to express that a given task must be accomplished by a set of subjects concurrently. In the military field, this allows to capture the idea that two individuals need to command at the same time a missile launch, for instance. Privileges are then represented as tuples of the form (S, o, x), where S is a non-empty set of subjects, o is an object and x is an access mode.

In [8], the $\star$-secure property was modified to take joint access into account. If we consider the list M of current accesses, then this property is stated as follows:

> "*a state is $\star$-secure if for any (sets of) subjects S_1, S_2 and objects o_1, o_2, if $m(S_1, o_1, read) \in M$ and $m(S_2, o_2, write) \in M$ and the classification of o_1 dominates that of o_2, then $S_1 \cap S_2 = \emptyset$* "

A direct translation of this sentence leads to the following logical formula:

$$\forall S_1, S_2\ \forall o_1, o_2 \in O$$
$$(m(S_1, o_1, read) \in M \wedge m(S_2, o_2, write) \in M \wedge f_o(o_1) \succ f_o(o_2)) \Rightarrow S_1 \cap S_2 = \emptyset$$

This expression is logically equivalent (by contraposition) to:

$$\forall S_1, S_2\ \forall o_1, o_2 \in O$$
$$(m(S_1, o_1, read) \in M \wedge m(S_2, o_2, write) \in M \wedge S_1 \cap S_2 \neq \emptyset) \Rightarrow \neg(f_o(o_1) \succ f_o(o_2))$$

The BLP model can be obtained from the McLean one by considering it as a particular case of the joint access model where every set of subjects S is reduced to a singleton $\{s\}$ and thus the above property becomes:

$$\begin{aligned} &\forall s \in S\ \ \forall o_1, o_2 \in O, \\ &\qquad m(\{s\}, o_1, read) \in M \wedge m(\{s\}, o_2, write) \in M \Rightarrow \neg(f_o(o_1) \succ f_o(o_2)) \end{aligned} \tag{3}$$

This property is equivalent to the $\star$-property of BLP (2) only when the order among security levels is total since only in this situation we have [7]:

$$\neg(f_o(o_1) \succ f_o(o_2)) \Leftrightarrow (f_o(o_2) \succeq f_o(o_1))$$

In general, actual implementations of multilevel security models adopt partial orders to compare security levels, having non-linear lattice structures. Military systems often employ extra labels to annotate the security levels. This practice

introduces "compartments", for instance $secret_{Navy}$ and $secret_{Army}$. Thus, subjects with clearance $secret_{Navy}$ may access objects with the same classification, but not those from the $secret_{Army}$ class. Thus, the corresponding lattice of security levels has incomparable elements [14].

Therefore, in McLean's interpretation of BLP, a subject will be able not only to write objects with a superior security classification than his, but also those objects which cannot be compared to its security clearance. We show that this formulation is not only less restrictive than the original BLP model, but may also allow information leaks to happen.

3 Short introduction to Tom

Tom is a language extension which adds new matching primitives to existing imperative languages, and in particular Java. This is particularly well-suited for describing transformations on structured entities like trees/terms. The language provides support for matching modulo associativity and neutral element, which is particularly useful to manage lists of objects and to model the search space exploration.

The main originality of this system is its integration into existing languages, which makes it possible to use efficient and well-designed runtime libraries. For example, in the presented work, we make an intensive use of the `Collection` framework provided by the Java API. From an implementation point of view, Tom is a compiler which accepts several *native languages* as input, like C or Java, and whose compilation process consists in translating the matching constructs into the underlying native language. For an interested reader, design and implementation issues related to Tom are presented in [10,1].

In addition to matching constructs, Tom offers support (via Gom) for the definition of algebraic data-structures. Gom is a generator of abstract syntax tree implementations [16] which takes a many-sorted signature as input (composed of constructor symbols), and generates an efficient Java implementation. The generated code is characterized by strong typing combined with maximal sub-term sharing for memory efficiency and constant time equality checking.

For expository reasons, we assume that Tom only adds two new constructs: `%match` and *back-quote* (‘). The first construct is similar to the `match` primitive of ML and related languages: given a term (called subject) and a list of pairs pattern-action, the `match` primitive selects a pattern that matches the subject and performs the associated action. This construct may thus be seen as an extension of the classical `switch/case` construct. The second construct is a mechanism that allows one to easily build ground terms over a defined signature. This operator, called *back-quote*, is followed by a well-formed term built over constructors, variables and function calls. This term is written in prefix notation.

In order to give a better understanding of Tom's features, let us consider that `read` is a binary symbol used to represent the access of a given `subject` to a `resource` (both characterized by an identifier and a security level). For instance, the following statement assigns to the Java variable `a` the read access of the re-

source `abstract` by the subject `bob`:

```
Access a = `read(subject("bob","secret"),
                 resource("abstract","confidential"));
```

Such a Java variable can be matched for checking if an access is authorized or not:

```
%match(a) {
    read(subject(x,l1),resource(y,l1)) -> {
          print("same level");
    }
    read(subject(x,l1),resource(y,l2)) -> {
          if(l1<l2) print("denied");
    }
}
```

This example should be read as follows: given a term `a` (that represents an `Access`), the evaluation of `%match` prints `"same level"` if the involved subjects and resources have the same security level and `"denied"` when the access is not authorized. This is implemented by non-linear pattern matching: `l1` occurs twice in the first pattern. The reader should note that variables do not need to be declared, their type is automatically inferred from the definitions of the operators using them. The action associated to a pattern is a Java statement which can be guarded using the classical `if` construct. In the second pattern, we retrieve the two security levels `l1` and `l2`, and we assume that `<` is an order that makes their comparison possible. When `l1` is less than `l2`, the `"denied"` message is printed.

As mentioned previously, an important feature of Tom is to support list matching, also known as associative matching with neutral element. For instance, lists of accesses can be represented using the associative operator `accesses` and the verification of the $\star$-security property can be encoded as follows:

```
%match(listOfAccesses) {
  accesses(_*,read(s,resource(_,l1),_*,
              write(s,resource(_,l2),_*)) -> {
    if(l1>l2) {
      return false; // *-property not verified
    }
  }
}
```

In this example, one can remark the use of *list variables*, annotated by a '`*`': such a variable should be instantiated by a (possibly empty) list. In this example, in order to avoid giving a name to variables not reused in the action side, we used anonymous variables, denoted by '`_`'. Given a list of accesses, the matching algorithm tries to find two accesses (a `read` and a `write`) performed by the same subject `s` and such that the level `l1` is higher than `l2`. When such a match is found this code returns `false`.

4 Encoding the policies in Tom

In this section we give an encoding in Tom of the security properties of the BLP and McLean policies and propose implementations that will be checked against these properties. Each policy is encoded by a class that extends the abstract class `Policy` which declares two abstract functions: `valid` and `transition`.

The function `transition` defined in each concrete class encodes the implementation of the respective policy, and defines the state transition following an access request; the state does not change if the request is denied and the corresponding access is added to the list of current accesses only if the request is granted. Each of the obtained states can be checked against the function `valid`, which is defined for each policy by encoding their specific security properties.

4.1 Data structures

The datatypes corresponding to the main entities present in the definition of an access control policy — the *subjects*, the *objects* and the *accesses* performed by subjects on objects — are defined abstractly using many-sorted signatures and, more precisely, we use Gom to generate a Java implementation which ensures maximal sub-term sharing. We will also define the current *state* as the list of accesses performed so far in the system.

Subjects and objects (called in what follows resources in order to avoid the potential conflicts with the `Object` class of Java) are characterized by an integer `id` and a security level built on strings:

```
Subject = subject(id:int,sl:SecurityLevel)
Resource = resource(id:int,sl:SecurityLevel)
SecurityLevel = sl(l:String)
```

Subjects and objects can be easily grouped together in lists built using the (associative) operators `subjects` and `resources` respectively:

```
ListOfSubjects = subjects(Subject*)
ListOfResources = resources(Resource*)
```

The accesses performed by subjects on resources can be of type `read` or `write`. For extensibility reasons, the access `mode` can be also specified explicitly using the `access` operator:

```
Access = read(subject:Subject,resource:Resource)
       | write(subject:Subject,resource:Resource)
       | access(mode:int,subject:Subject,resource:Resource)
```

The access modes 0 and 1 are associated to `read` and `write` accesses; this correspondence is enforced using rewriting rules which are systematically applied:

```
rules() {
  access(0,s,r) -> read(s,r)
  access(1,s,r) -> write(s,r)
```

```
}
```

The global state of the system is defined as the list of already performed accesses

```
State = state(accesses:ListOfAccesses)
ListOfAccesses = accesses(Access*)
```

A request can be made for gaining access to a resource (`add`) or for releasing a (previously obtained) access (`delete`):

```
Request = add(access:Access)
        | delete(access:Access)
ListOfRequests = requests(Request*)
```

An access request can be granted or denied according to the accesses already performed in the system.

```
Decision = grant(state:State)
         | deny(state:State)
```

For the purpose of this paper we only consider `grant` and `deny` decisions but the definition can be easily extended to more elaborated decisions (like `not-applicable` for example). Similarly, the definition of a state can be easily extended to contain more information like the security levels of the subjects when they are dynamic or other environment specific data.
The security levels lattice is encoded using a list of lists of security levels:

```
SecurityLevelsLattice = slLattice(SecurityLevelsSet*)
SecurityLevelsSet = slSet(SecurityLevel*)
```

The levels from each list are comparable with the first element being the lowest and the last the biggest. When the security levels are totally ordered the lattice is specified by only one list containing all the security levels. For example, the order introduced in Section 2.1 can be encoded by `slLattice(slSet(sl("unclassified"), sl("confidential"), sl("secret"), sl("top secret")))`. We should mention that no check is done on the coherence between the orders specified by each list but this can be easily done by extending the construction functions of `slLattice` and `slSet` with explicit rewriting rules.

The comparison of two security levels with respect to the given lattice can be implemented as follows:

```
public int compare(SecurityLevel l1,SecurityLevel l2) {
  if(l1==l2) { return 0; }
  %match (this) { // a lattice of security levels
    slLattice(_*,slSet(_*,m,_*,n,_*),_*) -> {
      if(l1==m && l2==n) { return -1; }
      if(l2==m && l1==n) { return 1; }
    }
  }
  return 100; // l1 and l2 are incomparable
}
```

Note the use of associative matching to describe the search of an `slSet` which contains the two elements we want to compare. Auxiliary functions like `leq`, `ge`, etc. can be easily defined using the function `compare`.

```
public boolean leq(SecurityLevel l1,SecurityLevel l2) {
  int v = compare(l1,l2);
  return (v==0 || v==-1);
}
```

4.2 Encoding the security properties

The properties (1) and (2) given in Section 2.1 define in fact the acceptable states with respect to the BLP policy. This is encoded in Tom by a predicate `valid(State cs)` that evaluates to `true` if the given state `cs` is acceptable and to `false` otherwise.

A given state is not valid w.r.t. the BLP policy if one of its current accesses is a read access involving a resource with a security level which is not lower than that of the concerned subject (1) or if the state is not $\star$-secure (2). The validity of a state depends on the current state and on the security levels lattice of the given instance of the policy. The attribute `slL` defines this lattice of security levels for the corresponding policy and it is initialized when an instance of the Tom class encoding the given policy is created.

```
public boolean valid(State cs) {
  %match(cs) {
    state(accesses(_*,read(subject(_,ssl),resource(_,rsl)),_*)) -> {
      if(!slL.leq(rsl,ssl)) { return false; }
    }
    state(accesses(_*,read(s,resource(_,rsl1)),_*,
                      write(s,resource(_,rsl2)),_*)) -> {
      if(!slL.leq(rsl1,rsl2)) { return false; }
    }
  }
  return true;
}
```

As far as it concerns the McLean policy, the simple secure property is exactly the same, however the $\star$-secure property (3) is slightly different and corresponds to the following matching statement:

```
state(accesses(_*,read(s,resource(_,rsl1)),_*,
                  write(s,resource(_,rsl2)),_*)) -> {
  if(slL.le(rsl2,rsl1)) { return false; }
}
```

These functions are used in the checker for testing whether all reachable states are valid by using the the implementations of both policies.

4.3 BLP and McLean implementations

The implementation of a policy can be seen as a function `Decision transition(Request req, State cs)` that given an access request `req` and a state `cs` (defined by the list of previously granted accesses) returns a (grant or deny) decision. These decisions are taken based not only on the current state but depend also on the precise security levels lattice of the given instance of the policy.

The implementation of the BLP policy is realized starting from the properties (1) and (2) given in Section 2.1. We focus on the denied accesses and thus we consider the negation of the previously mentioned properties:

$$\neg(f_s(s) \succeq f_o(o)) \Rightarrow m(s, o, read) \notin M \tag{4}$$

$$\neg(f_o(o_2) \succeq f_o(o_1)) \Rightarrow m(s, o_1, read) \notin M \vee m(s, o_2, write) \notin M \tag{5}$$

A read request is denied if either the subject does not have enough privileges to read the corresponding object (4) or if there exist write accesses by the same subject and the security levels of the objects already written by the respective subject are not higher than the level of the object to read (5). This is achieved by matching the current request and state and check whether these conditions are satisfied:

```
%match(req,cs) {
  add(read(subject(_,ssl1),resource(_,rsl1))), _ -> {
    // not enough privileges to read
    if(!slL.leq(rsl1,ssl1)) { return ‘deny(cs); }
  }

  add(read(s,resource(_,rsl1))),
  state(accesses(_*,write(s,resource(_,rsl2)),_*)) -> {
    // existing write access with lower level
    if(!slL.leq(rsl1,rsl2)) { return ‘deny(cs); }
} }
```

Write access requests are handled similarly: the decision is in this case independent of the relationship between the levels of the subject and of the object but the write access of a subject over an object is denied if the subject has already performed a read access and the security level of the read object is not lower than that of the written object (5).

```
%match(req,cs) {
  add(write(s,resource(_,rsl1))),
  state(accesses(_*,read(s,resource(_,rsl2)),_*)) -> {
    // existing write access with lower level
    if(!slL.leq(rsl2,rsl1)) { return ‘deny(cs); }
} }
```

If none of these conditions is satisfied then the access is granted resulting in a new

state with containing this latter access.

```
%match(req,cs) {
  add(newAccess), state(accesses) -> {
    return ‘grant(state(accesses(newAccess,accesses*)));
} }
```

We proceed similarly for the implementation of the McLean policy by adopting this time properties (1) and (3) given in Section 2.1. The implementation is the same as before except for the $\star$-*property* whose negation is

(6) $f_o(o_1) \succ f_o(o_2) \Rightarrow m(\{s\}, o_1, read) \notin M \vee m(\{s\}, o_2, write) \notin M$

A read request is denied in this model if there exist write accesses by the same subject and the security levels of the objects already written by the respective subject are smaller than the level of the read object (6).

```
%match(req,cs) {
  add(write(s,resource(_,rsl1))),
  state(accesses(_*,read(s,resource(_,rsl2)),_*)) -> {
    if(slL.ge(rsl2,rsl1)) { return ‘deny(cs); }
} }
```

A similar rule is used for the read access requests.

4.4 Computing implicit accesses

In our approach, a state is represented by the list of authorized accesses. Of course, this list contains the accesses that have been explicitly requested and granted, but it should also contain the accesses that correspond to a possible information flow. These accesses can be deduced by transitive closure:

(7) $m(s_1, o_1, read) \wedge m(s_1, o_2, write) \wedge m(s_2, o_2, read) \Rightarrow m(s_2, o_1, read)$

To ensure at the implementation level that a state always contains both *explicit* and *implicit* accesses, we rely on the *hook* mechanism provided by Gom [13]. A hook is a specialization of a construction function. Instead of just allocating the memory to represent a constructor of the signature, a hook performs some additional treatments ensuring that the resulting term is in a particular canonical form. In our case we want to ensure that the implicit accesses are always represented in a state. This is implemented by specializing the insertion of an access `e` in a list `l` and by considering four cases:

```
accesses:make_insert(e,l) {
  // do not add an access already there
  if(l.contains(e)) { return l; }
  %match(e,l) {
    read(s1,o1),  accesses(_*,read(s2,o2),_*,write(s1,o2),_*)
  | read(s2,o2),  accesses(_*,read(s1,o1),_*,write(s1,o2),_*)
  | write(s1,o2), accesses(_*,read(s1,o1),_*,read(s2,o2),_*)
```

```
    | write(s1,o2), accesses(_*,read(s2,o2),_*,read(s1,o1),_*) -> {
        return `accesses(read(s2,o1),realMake(e,l));
      }
    }
  }
```

The '|' stands for a disjunction of patterns. The `realMake` function, which consists in the "inner" default allocation function. This function takes the same number of arguments as the hook. In any case, if the hooks code does not perform a return itself, this realMake function is called at the end of the hook execution, with the corresponding hooks arguments.

5 Search space exploration

5.1 Simple checker

A first naive version of the checker uses the lists of subjects $S = (s_1, \ldots, s_m)$ and resources $R = (r_1, \ldots, r_n)$ to generate a list of potential requests $L = (\texttt{read}(s_i, r_j), \texttt{write}(s_i, r_j), \ldots)$ with $i \in [1..m]$, $j \in [1..n]$. Then starting from an empty state, it performs a transition for each request $r \in L$. Each transition leads to a new state and the process is applied recursively on this new state and $L \setminus \{r\}$. When L is empty, the reached state is checked whether it is valid with respect to the security properties. This can be implemented as follows in Tom:

```
void checker(State s, Policy p,
             ListOfRequests previous,
             ListOfRequests lor) {
  %match(lor) {     // generate all possible permutations
    requests(R1*,r@(add|delete)((read|write)(subject,resource)),
             R2*) -> {
      Decision decision = p.transition(`r,s);
      State newState = decision.getstate();
      ListOfRequests nextLor, nextPrevious;
      if(decision.isgrant()) {
        nextPrevious = `requests();
        nextLor = `requests(previous*,R1*,R2*);
      } else {
        // opt: on deny, do not re-explore previous and R1
        nextPrevious = `requests(previous*,R1*);
        nextLor = `requests(R2*);
      }
      checker(newState,p,nextPrevious,nextLor,newTraces);
    }
    // do the validation  when the list of requests is empty
```

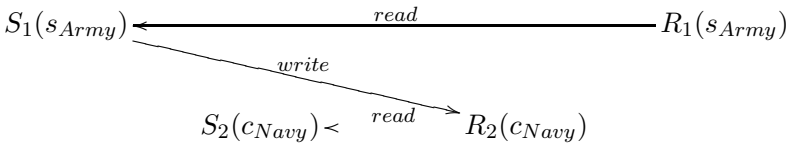
```
    requests() -> {
      if(p.valid(s) == false) {
        System.out.println("LEAKAGE DETECTED");
} } } }
```

It is important to remark in the code above that the action part does not contain `return` statements: all the solutions of the matching problem are computed. Combined with a recursive call to `checker`, this computes all possible permutations of the initial list of requests.

When, for a given permutation, all the transitions have been explored, the validity of the resulting state is checked and a message is printed when an information leakage is detected. This approach is sound and complete.

5.2 Information leakage

The execution of this checker for BLP policies with different combinations of comparable and incomparable security levels showed no information leakage. On the other hand, the execution of the checker for a McLean policy with a security lattice containing two incomparable levels $s(ecret)_{Army}$ and $c(onfidential)_{Navy}$ reveals the following information leakage:



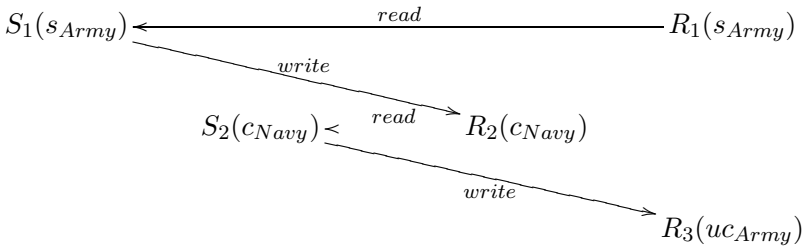
The depicted accesses correspond to valid transitions: since the two security levels are not comparable, the expression `ge(sArmy,cNavy)` tested when a write access is requested in the McLean policy (Section 4.3) is evaluated to false and thus the access is granted. On the other hand, the state containing these three accesses is not valid since the implicit read access of $R_1(s_{Army})$ by $S_2(c_{Navy})$ does not satisfy the simple secure property (since `!leq(sArmy,cNavy)` evaluates to true).

One can argue that the simple secure property of McLean should be modified to use a similar test as for the $\star$-secure property. The simple secure property (1) would become in this case:

$$\forall s \in S\ \forall o \in O \quad m(s, o, read) \in M \Rightarrow \neg(f_s(o) \succ f_o(s))$$

The previous scenario is no longer possible in this case but we obtain another

situation corresponding to an information leakage:



where *uc* stands for *unclassified* and $s_{Army} \succeq uc_{Army}$ but there is no relationship between these levels and c_{Navy}.

Once again all the (depicted) explicit accesses correspond to valid transitions but this time the implicit read access of $R_1(s_{Army})$ by $S_2(c_{Navy})$ is valid as well. On the other hand, this implicit access together with the write access of $R_3(uc_{Army})$ by $S_2(c_{Navy})$ leads to a violation of the $\star$-secure property since `le(ucArmy,sArmy)` evaluates to true.

5.3 A first optimization: caches and canonical forms

Checking if a state is valid is a frequently performed operation. To improve its efficiency, we store the valid states in a cache: a hash-map where the least recently used entry is dropped when needed.

Given a state and a list of requests that has to be explored, we also use a cache technique to store the already visited pairs (state, remaining requests), avoiding to explore twice the same search branch.

It is important to note that this approach is only effective if the equivalent states are represented in a unique way. For instance, consider the terms `‘state(accesses(a1,a2))` and `‘state(accesses(a2,a1))` of sort `State` (where `a1` and `a2` are of sort `Access`). They represent equivalent states since the order of granted accesses is not relevant. To reflect this equivalence class at the implementation level, we use the *hook* mechanism again to specialize the list construction function obtaining automatically sorted lists of accesses. We only present here the sorting algorithm but this code should be, of course, combined with the computation of implicit states presented previously:

```
accesses:make_insert(e,l) {
  // do not add an access already there
  if(l.contains(e)) { return l; }
  %match(l) {
    accesses() -> { return ‘realMake(e,accesses()); }
    accesses(head,tail*) -> {
      if(e.compareToLPO(‘head) > 0) {
        return ‘realMake(head,accesses(e,tail*));
      } else {
        return ‘realMake(e,l);
```

```
} } } }
```

Note the use of `realMake` which is the function that effectively concatenates `e` and `l` (`realMake` can only be used in a construction function). Note also that `compareToLPO` is an automatically generated total ordering function.

In practice, these optimizations are essential to reduce the search space and the memory footprint, as well as to improve the overall efficiency.

5.4 A more effective optimization: cut on deny

The optimizations presented in the previous section are independent of the security model being checked provided its properties depend on the set of current accesses. More effective optimizations can be obtained if some assumptions are made on the analyzed policy model. In particular, if we suppose that any granted access (and the corresponding modification of the set of current accesses) would lead to a new state where less requests will be granted, then the only states that should be checked are those obtained following a granted access; all the denied requests lead to states that will be eventually reached if only granted requests are considered. This assumption is obviously satisfied for the policies we considered but would not be reasonable for an access policy that requests, for example, two previously granted accesses of two subjects before granting the access for a third subject. The optimization proposed below works thus for the policies analyzed here but cannot be generalized to all access policies.

The new checker is simpler and more efficient:

```
private static void otherChecker(State s, Policy p,
                                 ListOfRequests lor) {
  %match(lor) {
    requests(R1*,r@(add|delete)((read|write)(subject,resource)),
             R2*) -> {
      ...
      if(decision1.isgrant()) {
        nextLor = `requests(R1*,R2*);
        otherChecker(newState,p,nextLor);
} } } }
```

We should point out that under the same assumption the set of final states (i.e. that are not modified by subsequent requests) obtained when considering `delete` requests is the same as if these requests were not attempted. This is why the set of initial requests we generate will contain no `delete` requests.

5.5 Experimental results

In this section we compare the efficiency of the simple checker with the one that successively integrates the presented optimizations. `+cache` stands for the integration of the least recently used cache mechanism. `+canonical form` corresponds to the unique representation of equivalent states (in addition to the cache mecha-

nism). Finally, `cut on deny` corresponds to the integration of the last presented optimization.

We considered scenarios which involve a finite number of subjects and a finite number of security levels.

The experiments have been performed on a Mac Pro 2x3GHz. For each version of the program we measured (in seconds) the time to check there is no information leakage. The columns ($m \times n$) correspond respectively to the number of *subjects* and the number of (objects with different) *security levels*:

Subjects × *Levels*	2×2	3×3	3×4	2×6	2×7	3×5	2×8	4×4
`Original`	0.5	> 1000	*na*	*na*	*na*	*na*	*na*	*na*
+`cache`	0.35	> 1000	*na*	*na*	*na*	*na*	*na*	*na*
+`canonical form`	0.18	131	*na*	*na*	*na*	*na*	*na*	*na*
+`cut on deny`	0.18	3.1	4.4	2.0	10.3	378	1309	63526

These experiments clearly show that without any optimization, checking the lack of information leakage can only be done in very simple cases: 2 subjects and 2 levels (*na* corresponds to memory explosion). The last row corresponds to the integration in the checker of all presented optimizations and shows that the verification of small, but realistic, configurations where more than 2 security levels are involved is possible. These results also show a clear need for new methods and new tools to tackle problems with unbounded numbers of subjects and security levels.

6 Conclusions

We summarize here the contributions of this paper and discuss some related works.

We have proposed an approach merging declarative and imperative paradigms for detecting information flow violations in (lattice-based) access control models. We have defined the state of the system as the list of current accesses and the policy implementations and the security properties they should verify as sets of rewrite rules (extended with operations from a host imperative language). The built-in list-matching available in Tom allows us to express easily the random selection of all possible access requests which can occur in a given system and thus to naturally describe the exploration of the search space.

The specification technique we have proposed in this paper allows us to explore a finite state space, hence it is well suited for detecting possible flaws in a given policy instance, but not for proving its correctness in an unbounded model. The advantage of this approach is that the specified policies can be not only checked but also used almost directly in applications written in Java (or another host language accepted by Tom).

We have only specified two simple access control policies but we claim that our results can be transferred to other security models as our framework can easily

accommodate new policy definitions.

Related Work Static analysis techniques have been applied to verify information flow in programs. This consists in finding out every data flow in the code of a program, and then to avoid paths which violate the security policy. This was done, for instance, on some extension of λ-calculus [6], or on extensions of programming languages as in [11]. Most of these extensions include a sophisticated type system to cope with how information can flow from protected variables with high security level to lower classified variables as surveyed in [15].

In [5], the authors verify the satisfiability of a temporal logic formula in a given policy model, but it is not clear how they could address information flow. In [4], the authors suggest to perform goal reachability over Datalog programs as a way to compare access control policies, but information flow is not addressed as well.

In [12] an approach close to ours is used for finding vulnerabilities by using rules to infer attacks from existing access control configurations in the Windows XP and SELinux operating systems. Several kinds of attacks are explored and the formalism used for specifying both the policy and the vulnerabilities is a variation of Datalog with negation. Our technique is more general, since we model check a security model instead of starting from concrete situations. We can model other kinds of vulnerabilities as well by integrating extra rules in the verification function.

References

[1] Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau, and Antoine Reilles. Tom: Piggybacking rewriting on java. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *Lecture Notes in Computer Science*, pages 36–47. Springer-Verlag, 2007.

[2] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report Mitre Report ESD-TR-73-278 (Vol. I-III), Mitre Corporation, 1974.

[3] D. Elliot Bell and Leonard J. LaPadula. Secure computer systems: A mathematical model, volume II. *Journal of Computer Security*, 4(2/3):229–263, 1996.

[4] Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. Specifying and reasoning about dynamic access-control policies. In Ulrich Furbach and Natarajan Shankar, editors, *IJCAR*, volume 4130 of *Lecture Notes in Computer Science*, pages 632–646. Springer, 2006.

[5] Dimitar P. Guelev, Mark Ryan, and Pierre-Yves Schobbens. Model-checking access control policies. In Kan Zhang and Yuliang Zheng, editors, *ISC*, volume 3225 of *Lecture Notes in Computer Science*, pages 219–230. Springer, 2004.

[6] Nevin Heintze and Jon G. Riecke. The SLam calculus: Programming with secrecy and integrity. In ACM, editor, *Conference record of POPL '98: the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California, 19–21 January 1998*, pages 365–377, New York, NY, USA, 1998. ACM Press.

[7] M. Jaume and C. Morisset. A formal approach to implement access control. *Journal of Information Assurance and Security*, 2:137–148, 2006.

[8] J. McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.

[9] Charles Morisset and Anderson Santana de Oliveira. Automated detection of information leakage in access control. In Monica Nesi and Ralf Treinen, editors, *Second International Workshop on Security and Rewriting Techniques - SecReT 2007*, Paris, France, 2007.

[10] Pierre-Etienne Moreau, Christophe Ringeissen, and Marian Vittek. A Pattern Matching Compiler for Multiple Target Languages. In G. Hedin, editor, *12th Conference on Compiler Construction*, volume 2622 of *LNCS*, pages 61–76, Warsaw, Poland, May 2003. Springer-Verlag.

[11] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, New York, NY, USA, 1999. ACM.

[12] Prasad Naldurg, Stefan Schwoon, Sriram K. Rajamani, and John Lambert. *NETRA*: seeing through access control. In Marianne Winslett, Andrew D. Gordon, and David Sands, editors, *FMSE*, pages 55–66. ACM, 2006.

[13] Antoine Reilles. Canonical abstract syntax trees. In Grit Denker and Carolyn Talcott, editors, *Proceedings of WRLA 2006*, volume 176, pages 165–179, Vienna, Austria, 2006. Electronic Notes in Theoretical Computer Science.

[14] Ravi S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11):9–19, 1993.

[15] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security*, 21(1):5–19, Jan 2003.

[16] Mark G. J. van den Brand, Pierre-Etienne Moreau, and Jurgen Vinju. Generator of efficient strongly typed abstract syntax trees in Java. *IEE Proceedings - Software Engineering*, 152(2):70–78, April 2005.