

**Polytechnic**  
UNIVERSITY

Brooklyn · Long Island · Westchester

# **A Conservative Algorithm for Computing the Flow of Permissions in Java Programs**

**Gleb Naumovich**



**Department of Computer and Information  
Science**

**Technical Report  
TR-CIS-2001-07  
12/21/2001**

# A Conservative Algorithm for Computing the Flow of Permissions in Java Programs

Gleb Naumovich

## Abstract

Software programs are increasingly distributed and open, which, unless designers and coders are careful, makes such programs vulnerable to attacks. Java offers a built-in security mechanism, enabling programmers to give permissions to distributed components and check these permissions at run-time. This security model is flexible, but using it is not straightforward. In this paper, we propose a data flow algorithm for automated analysis of the permissions flow in Java programs. Our algorithm produces, for a given point in the program, a set of all permissions that are checked on all possible executions to this point. These data can be used in program understanding tools or directly in checking properties that assert what permissions must always be checked before access to certain functionality is allowed. The worst-case complexity of our algorithm is low-order polynomial in the number of program statements and permission types, while comparable previous approaches have exponential costs.

## 1 Introduction

Software programs are increasingly distributed and open. Component-based architectures in which software components with a well-defined behavior can be dynamically linked to a running application gain popularity [27]. A drawback of this openness is the potential for attacks against such applications.

Chin [3] identifies three aspects of confidence in software security. *Confidence in algorithms* means that the underlying cryptographic and transfer algorithms are resistant to attacks. *Confidence in security* refers to ensuring authentication of communication, that is preventing an untrusted entity from posing as a trusted one. Finally, *confidence in implementation* means that implementations of security algorithms correctly capture all the salient features of these algorithms. In this paper our focus is on confidence in implementation.

Internet applications are increasingly developed in Java, a language whose design addresses the need for security. Java implements a security model where software components may be given permissions to carry out certain operations, based on where on the Internet these components reside and whether they are digitally signed by trusted authorities [6, 26]. This built-in security mechanism makes it much easier for designers and programmers to secure Java applications against attacks than applications implemented in other languages, but writing secure Java applications is far from straightforward. The Java security model is quite complex and therefore can easily be misused. Such misuse may result in bugs that open a door for attacks involving distributed Java applications.

In this paper, we propose a data flow algorithm for analyzing the flow of permissions in Java programs. Since checking of permissions in JVM is based on the program call stack, we use a call graph [8] based model of the program (which we call permissions call graph) in our analysis. Each node in this graph corresponds to either a method call or a statement relevant to permission checking. With each node in the permissions call graph we associate a set of permissions that are required on all paths to this node. The data flow algorithm propagates these sets through the graph, until a fixed point is reached. The permission set associated with a node at the end of the algorithm is conservative in the sense that if there is a path to this node on which a permission  $p$  is not checked, then  $p$  is not in the set associated with this node.

Our algorithm operates on a lattice of permission sets that is based on the `implies` function of the Java Permissions model. In addition, this lattice can be extended with additional relationships between permissions that are based on the application-specific property that the user of the technique checks. The worst-case complexity of our algorithm is cubic in the number of method call statements and statements relevant for checking permissions in Java programs and quadratic in the number of different permission types used in the program.

The next section discusses related work. We briefly overview the Java security model in Section 2. Section 4 introduces the program model used in our approach and Section 5 describes the permissions lattice. In Section 6 we discuss the types of permission-based properties that can be checked by using the results of our algorithm. Section 7 describes our algorithm in detail, presenting the data flow equations and a worklist version of the algorithm, as well as the proofs of its conservativeness and complexity. Finally, Section 8 describes conclusions and future work.

## 2 Related Work

Various aspects of the Java security model have been studied. Java bytecode verification in particular received a lot of attention [2, 21–23, 25]. The implications of strong typing for security have been investigated in detail [4, 14, 29, 30]. A number of approaches have concentrated on checking the flow of data through programs, with the goal of ensuring the absence of covert channels (i.e. ensuring that security-sensitive information does not “leak” to variables that are available to untrusted users) [10, 17, 24].

Jensen et al. [12] proposed an approach for specifying and analyzing general security properties for programs with procedural control flow. Properties are specified in temporal logic and the algorithm for verification of these properties is based on generating all possible stack configurations. It is assumed that an upper bound on the number of method frames on the call stack exists. The worst-case complexity of this algorithm is therefore exponential in the number of method call statements in the program. Nitta et al. [19] extend the approach of [12] in a way that enables it to check safety properties specified in a regular language, without a restriction on the number of method frames on the call stack. The worst-case complexity of the algorithm, however, is not improved. Neither [12] nor [19] present experimental data.

In contrast with [12, 19], our approach is specific to the Java security model. Furthermore, our approach concentrates on a subset of security properties, namely security properties that reason about the types of permissions that must be held in order for certain regions of the application code to be executed. On the plus side, the worst-case complexity of our technique is cubic in the number of method call statements and statements that check permissions and enter and exit regions of privileged code and quadratic in the number of different permission types in the program. Therefore, in theory, our approach is more scalable than previous approaches.

The program model that our technique constructs and uses is closely related to the call graph. Construction of call graphs for object-oriented programs has been explored in detail [8]. Lightweight approaches relying on type information alone [28] or more sophisticated approaches relying on a form of points-to analysis [11, 13, 15] have been developed. In this paper, we assume that a call graph is constructed using one of the existing techniques.

Data flow analysis [9] has been used extensively for statically computing information about programs, with applications in compiler optimization [1], code understanding and visualization [7], and program verification [5, 18, 20]. Our algorithm for computing permission flow is flow-sensitive, i.e. it takes into account the order of statements in a method, and context-insensitive, i.e. it does not preserve information about context from which a method is called when analyzing this method. The algorithm works by solving a monotone forward-flow all-paths problem [16]. As a result, the algorithm has a low-order polynomial worst-case behavior.

## 3 Permissions in the Java Security Model

The Java Security Model relies on stack-based permission checking for access control. A *permission* is a first-class object that has name and type. Optionally, a permission can have a set of targets and actions. For example, a local disk

access permission has files and/or directories as targets and an action of reading, writing, or both reading and writing. Both targets and actions are specified as strings.

To protect a region of code against unauthorized access, a programmer has to insert an operation that tells the JVM to check that the classes whose methods are on the call stack all have the required permission. This can be done in one of two ways. The programmer may call the static `checkPermission` method of class `AccessController` or the `checkPermission` method of instances of class `SecurityManager` or one of its subclasses<sup>1</sup>.

Figure 1 shows the example we use throughout this paper. This example models an online banking system, where a customer's account can be linked to other accounts. Class `Account` represents a basic account that is not linked to any other accounts and provides operations for creating an account, checking the balance, debiting, or crediting an account, as well as transferring a sum of money to another account<sup>2</sup>. Internally, those methods of class `Account` that can modify the balance call on the protected method `write` of this class that is responsible for saving the account information persistently (in our case, information about each account is written to a file on the local disk, the name of which is given to the account at the time of the constructor call). Class `AccountWithProtection` represents an account that is linked to a *protection* account (e.g. overdraft protection). If the `debit` method of this class would result in an overdraft, the amount of overdraft is withdrawn from the protection account. The main method of class `CustomerInterface` gives an example where four accounts (checking, overdraft protection, credit card, and savings) are linked and initialized. The `debit` method that is called on the checking account exceeds the amount in the account, and therefore will result in a recursive call to the `debit` method of the overdraft protection account, then a subsequent call to the `debit` method of the credit card, and finally in a call to the `debit` method of the savings account. At the end of this main method, the checking, overdraft protection, and credit card accounts have the balance of \$0 and the savings account has the balance of \$1000.

The online banking example defines several application-specific permissions, `NewAccountPermission`, `BalancePermission`, `DebitPermission`, `CreditPermission`, and `CustomerPermission`. For example, the first statement of method `credit` of the `Account` class checks that permission `CreditPermission` is held by all classes on the call stack at the time of execution of that method.

Permissions are granted to Java bytecodes statically, using a *security policy* file. This file specifies the location of the classes that are granted one or more permissions, as well as entities whose signatures these classes must have. In our online banking example, we assume that code from any URL should be allowed to check balance and perform credit operation, as long as it is digitally signed by the bank. We assume that code from a very specific URL (e.g. representing a customer's home machine) is granted `CustomerPermission`. The security policy file for this example appears in Figure 2<sup>3</sup>.

Note that, while permissions have to be assigned statically using a policy file, checking permissions in the code is dynamic. In Java, instances of permissions, rather than statically defined permission types, are passed to the `checkPermission` methods. To enable static analysis of permissions, we assume that all targets and actions of permissions are specified statically in the code and that one instances of a permission class always implies any other instance of this class. We address this issue in detail in Section 4.

The Java security model allows application programmers to relax the permission requirements for a region of code. Such regions of code are called *privileged*. Any permission check operations that are encountered after a privileged region is entered and before it is exited are ignored by the JVM. This gives the programmer additional flexibility in configuring permissions for the application, but, if used injudiciously, is a threat to the program security.

A privileged region is entered when one of methods `doPrivileged` of class `AccessController` is called and exited when this method terminates. Methods `write` of class `Account` and `debit` of class `AccountWithProtection`

---

<sup>1</sup>No more than one object of class `SecurityManager` or its subclass can be created for any program. The default implementation of method `checkPermission` in `SecurityManager` simply calls method `checkPermission` of `AccessController`.

<sup>2</sup>To save space, we do not show the implementation of the `Money` class.

<sup>3</sup>The quoted strings that follow permission types in this file are matched with string parameters passed to constructors of the corresponding permissions in the code (see `checkPermission` calls in Figure 1).

```

public class Account {
    private Money balance;
    private String persistentLocation;

    public Account(Money initialAmount,
                  String persistentLocation) {
        AccessController.checkPermission(
            new NewAccountPermission("NewAccountPermission"));
        this.balance = (Money) initialAmount.clone();
        this.persistentLocation = persistentLocation;
    }

    public Money getBalance() {
        AccessController.checkPermission(
            new BalancePermission("BalancePermission"));
        return (Money) this.balance.clone();
    }

    public void credit(Money amount) {
        AccessController.checkPermission(
            new CreditPermission("CreditPermission"));
        this.balance.add(amount);
        this.write();
    }

    public void debit(Money amount) {
        AccessController.checkPermission(
            new DebitPermission("DebitPermission"));
        this.balance.subtract(amount);
        this.write();
    }

    protected void write() {
        AccessController.doPrivileged() (
            new PrivilegedAction() {
                public Object run() {
                    FileWriter writer = new FileWriter
                        (this.persistentLocation);
                    writer.write(balance);
                    writer.close();
                }
            }
        );
    }

    public void transfer(Money amount, Account toAccount) {
        this.debit(amount);
        toAccount.credit(amount);
    }
}

public class AccountWithProtection extends Account {
    private Account protection;

    public AccountWithProtection(Money initialAmount,
                                String persistentLocation,
                                Account protection) {
        super(initialAmount, persistentLocation);
        this.protection = protection;
    }

    public void debit(Money amount) {
        AccessController.checkPermission(
            new CustomerPermission("CustomerPermission"));
        AccessController.doPrivileged() (
            new PrivilegedAction() {
                public Object run() {
                    Money currentBalance = this.getBalance();
                    if (currentBalance.compare(amount) ==
                        Money.LESS_THAN) {
                        Money toTransfer = amount.clone();
                        toTransfer.subtract(currentBalance);
                        this.protection.transfer(toTransfer, this);
                    }
                    this.debit(amount);
                }
            }
        );
    }
}

public class CustomerInterface {
    public static void main(String[] args) {
        Account savings = new Account(
            new Money(3000, 0), "savings");
        AccountWithProtection credit =
            new AccountWithProtection(
                new Money(5000, 0), "credit", savings);
        AccountWithProtection overdraft =
            new AccountWithProtection(
                new Money(1000, 0), "overdraft", credit);
        AccountWithProtection checking =
            new AccountWithProtection(
                new Money(2000, 0), "checking", overdraft);

        checking.debit(new Money(10000, 0));
    }
}

```

Figure 1: An online banking example

in Figure 1 use this method<sup>4</sup>. The write method of class FileWriter<sup>5</sup> checks that its callers have a FilePermission permission. Because this method is called from a privileged region, the JVM will check that the Account class has a

<sup>4</sup>We construct an object that conforms to the PrivilegedAction interface in-line and pass it to doPrivileged.

<sup>5</sup>Defined in the standard java.io package.

```

grant signed "bank" {
  permission NewAccountPermission "NewAccountPermission";
  permission BalancePermission "BalancePermission";
};

grant codeBase "http://customer.machine.com/banking/classes/" {
  permission CustomerPermission "CustomerPermission";
};

```

Figure 2: The security policy file for the online banking example in Figure 1

FilePermission permission, but will not run this check for other classes whose methods directly or indirectly call method write of class Account.

## 4 Permissions Call Graph

A *Permissions Call Graph (PCG)* is a graph that statically captures the relationship between permissions checked in the program and the flow of control among program methods. To obtain this model, we modify the program model used in [12, 19]. Each node in PCG corresponds to one of the following actions in the program:

- method calls,
- permission checking (calls to method checkPermission of an object of class SecurityManager or one of its subclasses or static method checkPermission of class AccessManager),
- entering a privileged region, and
- exiting a privileged region.

A PCG can be constructed from the set of control flow graphs (CFGs) for all methods in the program by first reducing each of the CFGs so that only nodes corresponding to method calls, including permission checking methods and entering and exiting privileged regions, remain and then connecting method call nodes to the head nodes in the methods that may be called by these statements. Figure 3 shows an algorithm for constructing a PCG from the set of CFGs for a program.

Figure 4 shows the PCG for the example in Figure 1. To improve readability, we group PCG nodes into shaded boxes that represent methods and classes in the program (the main method appears in Figure 1 only for illustration and is not present in the PCG). In addition, to reduce the size of the PCG for this example, we omit calls to methods of class Money<sup>6</sup>. Note that because AccountWithProtection is a subclass of Account, the call to method debit from method transfer of class Account is polymorphic, with methods debit of both classes Account and AccountWithProtection as its targets. The same situation occurs at the place of call to debit inside method debit of class AccountWithProtection.

In the rest of this section we introduce some terminology for PCG that we then use in the description of our data flow algorithm. Formally a PCG is a tuple  $(N, H, CE, LE, kind)$ , where

- $N$  is the set of PCG nodes,
- $H$  is the set of *head nodes* of the PCG. A head node for a method marks the start of this method. Note that, according to the algorithm of PCG construction in Figure 3, a method can have more than one head node,

<sup>6</sup>This optimization can be generalized, if the called method is not important for checking permission-related properties. We assume that methods of class Money check no permissions and call (directly or indirectly) no methods that do.

**Algorithm 1 (PCG construction algorithm).**

```

Input: CFGs for all methods in the program.
Output: A PCG  $(N, H, CE, LE, kind)$ .
Initialization:  $N = H = CE = LE = \emptyset$ .
(1)  Inline all CFGs for privileged actions and create statements that represent the start and end of each privileged region
(2)  For each CFG  $c$ :
(3)    For each statement  $s$  in  $c$ :
(4)      if  $s$  represents entering a privileged region
(5)        Find statement  $s'$  that represents exiting this privileged region
(6)        Create  $n_1 \in N : kind(n_1) = \text{enterPriv}$ 
(7)        Create  $n_2 \in N : kind(n_2) = \text{exitPriv}$ 
(8)        Set  $Partner(n_2) = n_1$ 
(9)        Set  $\mu(s) = n_1$  and  $\mu(s') = n_2$ 
(10)     else if  $s$  represents a call to checkPermission:
(11)       Create  $n \in N : kind(n) = \text{check}$ 
(12)       Set  $\mu(s) = n$ 
(13)     else if  $s$  represents a method call
(14)       Create  $n \in N : kind(n) = \text{call}$ 
(15)       Set  $\mu(s) = n$ 
(16)     else if  $s$  is the method start instruction
(17)       Create  $h \in N, H : kind(h) = \text{head}$ 
(18)       Set  $\mu(s) = h$ 
      end if
    end for
  end for
(19) For each  $n \in N$ :
(20)   if  $kind(n) = \text{call}$ 
(21)     for each method  $m$  that can be called at  $n$ :
(22)       create  $(n, h) \in CE$ , where  $h$  is the head node of  $m$ 
    end if
(23)   For each  $n' \in N$ :
(24)     Let  $s = \mu^{-1}(n), s' = \mu^{-1}(n')$ 
(25)     if  $s$  and  $s'$  are in the same CFG and there is a path from  $s$  to  $s'$  such that for all statements  $s''$  on this path
      other than  $s$  and  $s'$ ,  $\mu(s'') = \text{null}$ 
(26)       create  $(n, n') \in LE$ 
    end if
  end for
end for

```

Figure 3: An algorithm for constructing a PCG from the set of CFGs for a program.

- $CE$  is the set of *call edges* that represent control flow across methods (these edges go from method call nodes to head nodes of the corresponding method),
- $LE$  is the set of edges that represent control flow within methods, and
- $kind$  is a labeling function  $N \rightarrow \{\text{call}, \text{check}, \text{enterPriv}, \text{exitPriv}, \text{head}\}$  that marks each node as a method call, permission checking, enter privileged region, exit privileged region node, or a method head node.

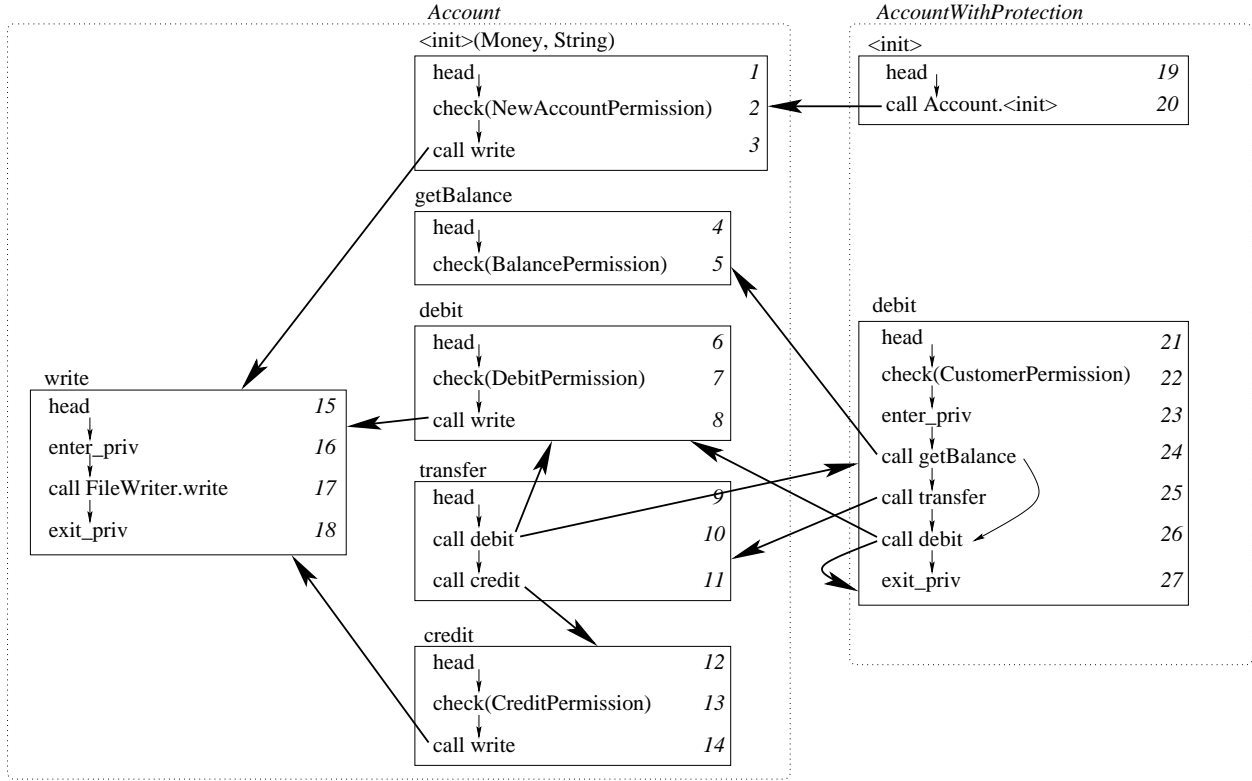


Figure 4: PCG for the online banking example in Figure 1

We write *Permissions* to denote the set of all permissions used in the program, as well as all permissions directly or indirectly implied by those. For example, for the online banking example in Figure 1,  $Permissions = \{NewAccountPermission, BalancePermission, CreditPermission, DebitPermission, CustomerPermission, WritePermission\}$ . In our algorithm for computing permissions, we overload term *Permissions* to denote the set of permissions that are checked on all paths to the given node.

For each node  $n$  in the PCG, we define functions *Preds* and *Succs*, returning respectively the set of all predecessors and successors for  $n$  in the PCG.

For the purposes of our technique, we represent privileged action command as a pair of instructions, one to represent entering and another to representing exiting a privileged region. The Java semantics imply that both of these instructions have to be in the same method. Because privileged actions can be nested, we match an instruction of exiting a privileged region with the instruction of entering this region. We write  $Partner(n_2) = n_1$  if  $n_1$  is a node that corresponds to the instruction of entering some privileged region and  $n_2$  is the node that corresponds to the instruction of exiting this privileged region.

We need to capture the flow of permissions through the graph statically. To be able to do that, we place the following restrictions on the use of permissions in the program under analysis:

- The values for targets and actions have to be statically defined strings. Our approach effectively replaces a permission with targets and actions with a number of permissions, one for each target-action combination. In the rest of the paper we assume that this transformation has been performed and therefore none of the permissions used in the program have targets or actions.

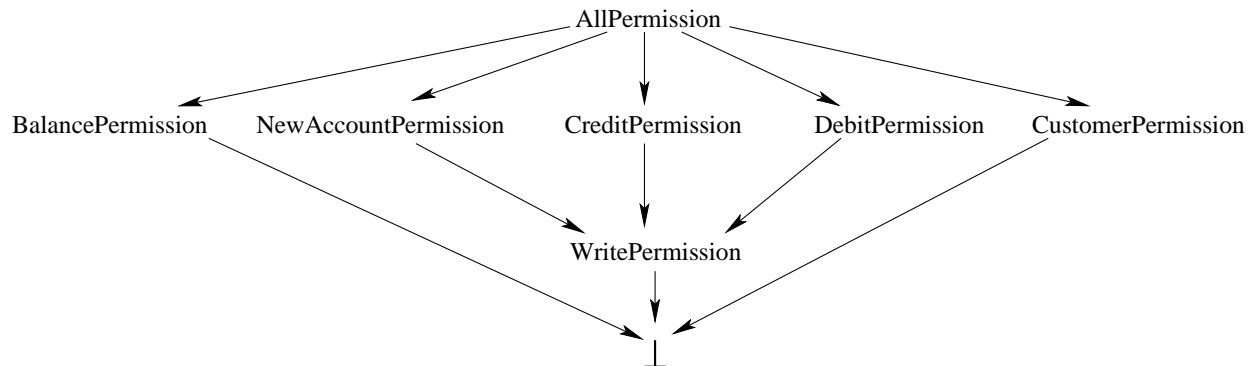


Figure 5: Permissions lattice  $L_{Perm}$  for the online banking example in Figure 1

- The `implies` function for each permission class is defined in such a way that a permission object implies all other permission objects of the same class.
- Permission objects are created only as in-line parameters in the calls to `checkPermission` methods of `SecurityManager` and `AccessController`.

Effectively, these restrictions allows us to use statically defined permission types as opposed to dynamically defined permission objects. Some of these restrictions can be relaxed by using alias resolution techniques [13], but we believe that in practice the restricted permission model is sufficient in most situations.

## 5 Permissions Lattice

Every permission class  $p_1$  has to specify an `implies` function. Given a permission object  $p_2$ , this function determines if any code that holds permission of class  $p_1$  is automatically granted permission  $p_2$ . We can define *permission graph* that contains a node for each permission class in the program and an edge  $(p_1, p_2)$  if any permission of class  $p_1$  implies all permissions of class  $p_2$  (restrictions on the use of permissions in Java that we introduced at the end of Section 4 allow us to reason in terms of permission classes instead of permission objects).

The permission graph for a program is in itself a useful security checking tool, since it visualizes the implication relationship among permission classes. For example, in general, if permission  $p_1$  directly or indirectly implies permission  $p_2$  and  $p_2$  directly or indirectly implies  $p_1$ , it means that all permissions involved in these implication relationships are equivalent. Therefore, strongly connected regions in the permission graph should be carefully investigated.

We use the permission graph only as an intermediate step. We collapse all permissions in each strongly connected component in this graph into a single permission, modifying the PCG accordingly. The resulting graph does not have cycles and therefore defines a lattice of permissions  $L_{Perm}$ , where the top element  $\top$  is `AllPermission`<sup>7</sup>, the bottom element  $\perp$  represents the absence of permissions, and the partial order relation is derived directly from the `implies` function (i.e., for two given permissions  $p_1$  and  $p_2$ , if  $p_2.\text{implies}(p_1) = \text{true}$ , then  $p_1 \sqsubseteq p_2$ ). The permission lattice for the permissions used in our example is shown in Figure 5.

We use  $L_{Perm}$  to define a lattice  $L_{2Perm}$  of sets of permissions. We would like to define a partial order on permission sets in this lattice from the implication-based partial order on individual permissions. At the first glance, it seems that this partial order on permission sets can be given simply as  $P_1 \sqsubseteq P_2$  if  $\forall p_1 \in P_1 \exists p_2 \in P_2 : p_1 \sqsubseteq p_2$ , where  $p_1$  and  $p_2$  are individual permissions and  $P_1$  and  $P_2$  are permission sets. However, defining a lattice that includes all permission

<sup>7</sup>A predefined Java permission that implies any other permission.

sets is not possible. For example, let  $p_1$  and  $p_2$  be two permissions used in a program. If  $p_1 \sqsubseteq p_2$ , then, according for the definition above,  $\{p_1\} \sqsubseteq \{p_1, p_2\}$  and  $\{p_1, p_2\} \sqsubseteq \{p_1\}$ .

We say that a set of permissions  $P$  is *canonical* if no implication relationship exists between any two permissions in this set:  $\forall p_1, p_2 \in P, p_1 \not\sqsubseteq p_2 \wedge p_2 \not\sqsubseteq p_1$ . We define the *canonical reduction* operation that removes from a permission set all permissions that are implied by some other permission in this set:

$$\forall P \in \text{Permissions}, \text{CR}(P) = \{p \mid p \in P \wedge (\forall p' \in P, p \not\sqsubseteq p')\}$$

**Theorem 1.** *For any set of permissions  $P$ , there exists only one corresponding canonical set, i.e. the canonical reduction operation is deterministic.*

*Proof.* Suppose that the statement of the theorem is wrong, i.e. there exists a set of permissions  $P_0$  that has two different canonical sets  $P_1$  and  $P_2$ . Take any permission  $p \in (P_0 \setminus P_1)$ , such that  $p \in P_2$  (if no such  $p$  exists,  $P_1 = P_2$ ). Since the canonical reduction that reduces  $P_0$  to  $P_1$  removes  $p$  from  $P_0$ , there is some other permission  $p'$  such that  $p' \in P_0 \wedge p' \in P_1$  and  $p \sqsubseteq p'$ .

If  $p' \in P_2$ , then  $P_2$  would not be canonical, since it would contain two permissions  $p$  and  $p'$  such that  $p \sqsubseteq p'$ . So,  $p' \notin P_2$ . By the definition of canonical sets, there must be a permission  $p'' \in P_2$  such that  $p' \sqsubseteq p''$ . Here we arrive at a contradiction, since  $p, p'' \in P_2$  and  $p \sqsubseteq p''$  by the transitivity of implications. Therefore, the statement of the theorem holds.  $\square$

We introduce a special *null* set that we will use to represent the fact that no information about permissions is available. Note that *null* is different from an empty set. We define merge ( $\sqcap$ ) and join ( $\sqcup$ ) operations on permissions sets as follows:

$$\begin{aligned} & \forall P_1, P_2 \in 2^{\text{Permissions}} \cup \{\text{null}\} : \\ & P_1 \sqcap P_2 = \begin{cases} P_1 \cap P_2 & \text{if } P_1 \neq \text{null} \wedge P_2 \neq \text{null} \\ P_1 & \text{if } P_2 = \text{null} \\ P_2 & \text{if } P_1 = \text{null} \end{cases} \\ & P_1 \sqcup P_2 = \begin{cases} \text{CR}(P_1 \cup P_2) & \text{if } P_1 \neq \text{null} \wedge P_2 \neq \text{null} \\ P_1 & \text{if } P_2 = \text{null} \\ P_2 & \text{if } P_1 = \text{null} \end{cases} \end{aligned} \quad (1)$$

Now we can define the lattice of permission sets  $L_{2^{\text{Perm}}}$  as follows. The top element  $\top$  of  $L_{2^{\text{Perm}}}$  is a set that consists of a single permission `AllPermission`. The bottom element  $\perp$  of  $L_{2^{\text{Perm}}}$  is the empty set. Any two sets of permissions  $P_1$  and  $P_2$  are elements of  $L_{2^{\text{Perm}}}$ .  $P_1 \sqsubseteq P_2$  iff  $\forall p_1 \in P_1, \exists p_2 \in P_2 : p_1 \sqsubseteq p_2$ . Figure 6 illustrates the  $L_{2^{\text{Perm}}}$  for the permissions used in the example in Figure 1.

## 6 Permission-based Properties

Since permissions are intended for protection of certain code regions from unauthorized access, it seems that the most straightforward way to specify permission-based property is of the general kind “in order to access this region of code, the following permissions must be held...”. In general, we specify properties on the PCG in the following way:  $n \Rightarrow P_1 \vee \dots \vee P_k$ . This means that the code corresponding to node  $n$  should be accessible only if all permissions in one of the permission sets  $P_1, \dots, P_k$  are obtained by the time execution reaches this node.

For the online banking example, we can specify a property that when the `write` method of class `FileWriter` is called, permissions `NewAccountPermission`, `CreditPermission`, or `DebitPermission` must be held:

$$17 \Rightarrow \{\text{NewAccountPermission}\} \vee \{\text{CreditPermission}\} \vee \{\text{DebitPermission}\}$$

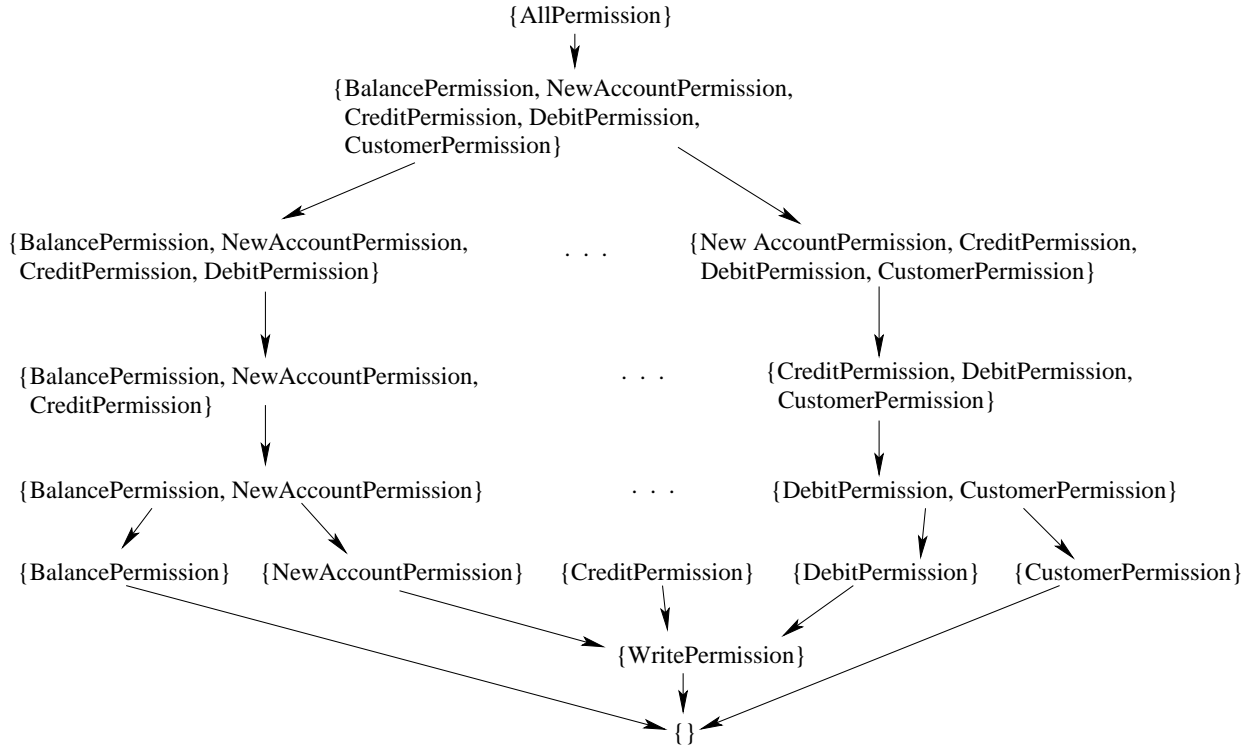


Figure 6: Permissions set lattice  $L_{2^{perm}}$  for the online banking example in Figure 1

In general, our approach requires a modification of the permission lattice  $L_{2^{perm}}$  to enable checking of such properties. The reason for this is that we use the merge operation on the lattice to compute the flow of permissions into PCG nodes. For each property  $R$  of the form  $n \Rightarrow P_1 \vee \dots \vee P_k$ , we introduce a new permission  $p_R$ . A set containing only this permission is added to  $L_{2^{perm}}$  in such a way that  $\forall i, 1 \leq i \leq k, P_i \implies \{p_R\}$ .

## 7 Data Flow Algorithm for Propagating Permissions through the Call Graph

In this section we describe a forward-flow all-paths data flow analysis for checking permission-based properties, called *Permission Flow Analysis (PFA)*. The intuition behind the PFA algorithm is as follows. JVM checks permissions for all classes whose methods are present on the call stack, except when privileged regions are used. We propagate information about permissions along method call chains in the PCG and obtain sets of permissions that must hold on all paths leading to a specific method. We can then compare the set of permissions thus computed for a node  $n$  with sets  $P_1, \dots, P_k$  in the permission-based property for this node.

In an application that can load classes dynamically, if a malicious class is loaded, this class can attempt to access any variables and objects that are accessible in the context from which methods of this class are called. This context is defined by the static public members of all public classes in the application, parameters passed to the methods of the malicious class, and any objects and fields that are reachable from those through method calls and references. While in the future we plan to use points-to analysis for computing a conservative estimate of the objects and values that are available to each method of a dynamically loaded class, in this paper we make a simple conservative assumption that all public methods of all public classes can potentially be called by the methods of a malicious class. Therefore, we

assume that there may be executions on which each public method of each public class is called with no permissions owned by the calling class.

## 7.1 Data flow equations

First, we introduce operations on sets of permissions that we need for the PFA algorithm. With each node  $n$  in the PCG we associate two sets of permissions,  $priv(n)$  and  $unpriv(n)$ . The former contains all permissions that hold on executions where a privileged region is entered but not exited by the time  $n$  is executed. The latter contains all permissions that hold on executions where node  $n$  executes outside privileged regions. Initially, both sets are set to *null* for all nodes in  $N$ , except the head nodes in public methods, for which  $unpriv$  sets are set to  $\emptyset$ .

The PFA algorithm propagates the  $priv$  and  $unpriv$  sets along the edges of the PCG. If several edges enter a node  $n$ , the permissions information flowing into  $n$  is *merged* before being propagated through  $n$ . The merge function  $merge : 2^{Permissions} \times 2^{Permissions} \rightarrow 2^{Permissions}$  uses the  $\sqcap$  operation to merge two sets. The intuition behind this is that only permissions that are present in all reachable predecessors of  $n$  have to be propagated into  $n$ . (If a predecessor of  $n$  is unreachable, its  $priv$  and  $unpriv$  sets are *null* and, according to equation (1), do not affect the flow of permission information into  $n$ .) We define sets  $IN_{priv}(n)$  and  $IN_{unpriv}(n)$  of separately merged privileged and unprivileged permissions coming to node  $n$  from its predecessors in the PCG:

$$\begin{aligned} IN_{priv}(n) &= \sqcap_{p \in Preds(n)} priv(p) \\ IN_{unpriv}(n) &= \sqcap_{p \in Preds(n)} unpriv(p) \end{aligned} \quad (2)$$

Propagation function  $prop_n : (2^{Permissions}, 2^{Permissions}) \rightarrow (2^{Permissions}, 2^{Permissions})$  defines the way in which the node  $n$  changes sets of permissions that are propagated into it:

$$prop_n(IN_{priv}(n), IN_{unpriv}(n)) = \begin{cases} (IN_{priv}(n), IN_{unpriv}(n)) & \text{if } kind(n) = \text{call} \\ (IN_{priv}(n), IN_{unpriv}(n) \sqcup \{Permissions(n)\}) & \text{if } kind(n) = \text{check} \\ (IN_{priv}(n) \sqcap IN_{unpriv}(n), null) & \text{if } kind(n) = \text{enterPriv} \\ (IN_{priv}(Partner(n)), IN_{unpriv}(Partner(n))) & \text{if } kind(n) = \text{exitPriv} \end{cases} \quad (3)$$

Privileged and unprivileged sets of permissions are propagated through the PCG by using merge and propagation functions of (2) and (3) until  $priv(n)$  and  $unpriv(n)$  stop changing for all nodes  $n$  in the PCG. At this point, the set of permissions that are checked on all paths to a given node  $n$  is computed simply as  $Permissions(n) = priv(n) \sqcap unpriv(n)$ .

## 7.2 Example

We illustrate the PFA algorithm on the online banking example in Figure 1. For each non-head node  $n$  in the PCG for this example (shown in Figure 4), we set initially  $priv(n) = unpriv(n) = null$ . This signifies that at this point, no path to the node exists, whether or not the node lies in a privileged region. For each head node  $n$  of a public method (nodes 1, 4, 6, 9, 12, 19, 21), we set  $unpriv(n) = \{\}$ .

Figure 7 shows the results of running our algorithm on the PCG for the online banking example from Figure 4. For each node  $n$ , the pair of permission sets  $(priv(n), unpriv(n))$  is shown next to it.

Consider the path  $21 \rightarrow 22 \rightarrow 23 \rightarrow 24 \rightarrow 9 \rightarrow 21 \rightarrow 22$ . Initially,  $IN_{priv}(21) = null$  and  $IN_{unpriv}(21) = \{\}$ . When this information is propagated through 21, according to the propagation equation (3), we obtain  $priv(21) = null$  and  $unpriv(21) = \{CustomerPermission\}$ . These sets become  $IN_{priv}(22)$  and  $IN_{unpriv}(22)$  respectively. Again, according to equation (3),  $priv(22) = \{CustomerPermission\}$  and  $unpriv(22) = null$ . These sets are subsequently propagated to nodes 23 and 24. When these sets are propagated in node 9, they are merged with  $(null, \{\})$  initially assigned to this node, which results in  $IN_{priv}(9) = priv(9) = \{CustomerPermission\}$  and  $IN_{unpriv}(9) = unpriv(9) = \{\}$ . When these sets are merged with  $IN_{priv}(21) = null$  and  $IN_{unpriv}(21) = \{\}$ , we obtain  $IN_{priv}(21) =$

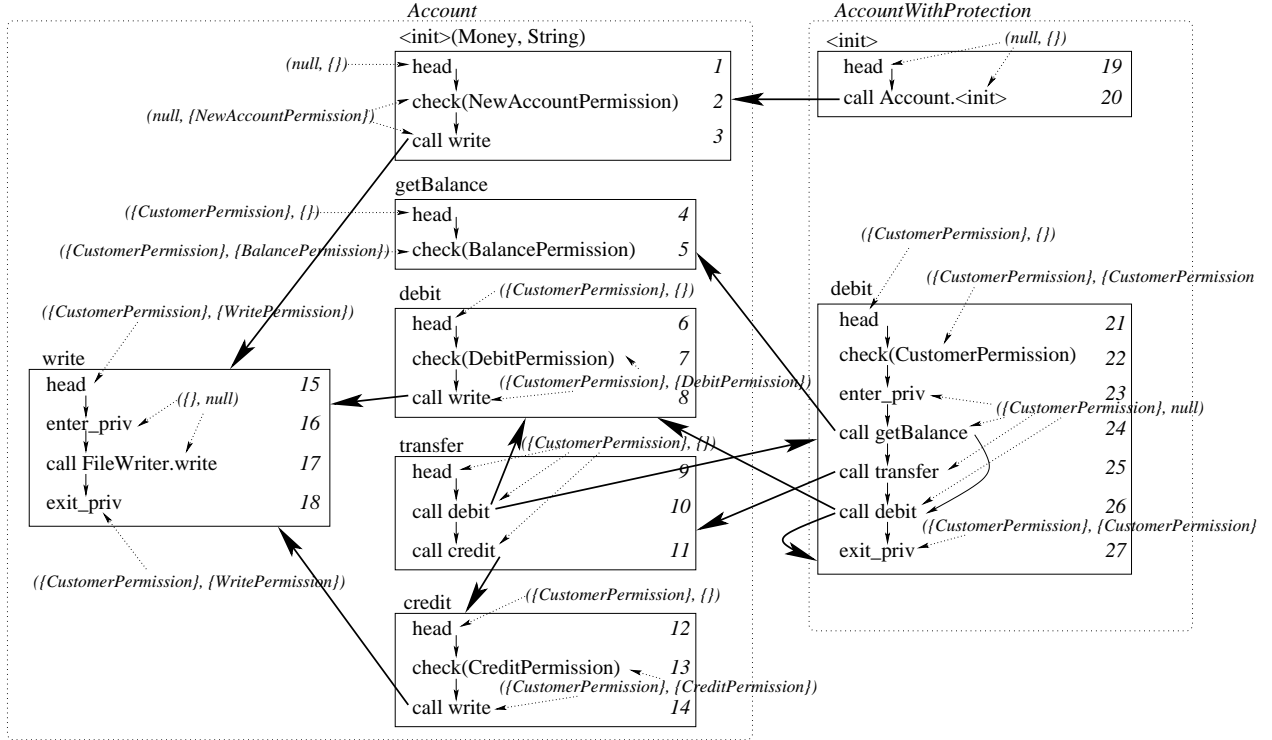


Figure 7: The PCG for the online banking example with  $(priv, unpriv)$  permission set pairs shown for each node.

$\{CustomerPermission\}$  and  $IN_{unpriv}(21) = \{\}$ . Once these sets are propagated through 14, we obtain  $IN_{priv}(21) = \{CustomerPermission\}$  and  $IN_{unpriv}(21) = \{CustomerPermission\}$ , which become  $IN_{priv}(22)$  and  $IN_{unpriv}(22)$  respectively. Finally, once these sets are propagated through 22, we obtain  $IN_{priv}(22) = \{CustomerPermission\}$  and  $IN_{unpriv}(22) = \{\}$ .

Observe that the PFA algorithm determines that  $priv(17) = null$  and  $unpriv(17) = \{\}$ , and therefore  $Permissions(17) = \emptyset$ . The reason for this is that there are two categories of paths through the PCG that lead to node 17. First, there are paths where a call to write is made while method debit of AccountWithPermission class is on the stack and therefore write executes in a privileged region, where only CustomerPermission is required from the code. Second, there are paths where a call to write is made from the constructor or methods debit and credit of Account class. The join of permissions CustomerPermission, NewAccountPermission, DebitPermission, and CreditPermission results in  $\perp$ .

In general, there may be several explanations for a violation of a permission-based property detected by the PFA algorithm. First, it is possible that permission checking statements are omitted or used incorrectly (e.g., a wrong permission may be checked or the check itself may happen in the wrong place). Second, definitions of some permissions used in the program, especially their *implies* methods, may contain errors. Third, the use of privileged regions may result in the required permissions not being checked. Finally, it is possible that the result is *spurious*, i.e. the property being checked holds for the actual program. Spurious results are possible, because all possible executions of the program are not modeled directly in the call graph. The developer of the application has to examine this result and decide whether it is a result of an error or a spurious result. Our algorithm can be extended to preserve information about propagation paths of permissions, so that when a property violation is detected, the corresponding path(s) can be examined.

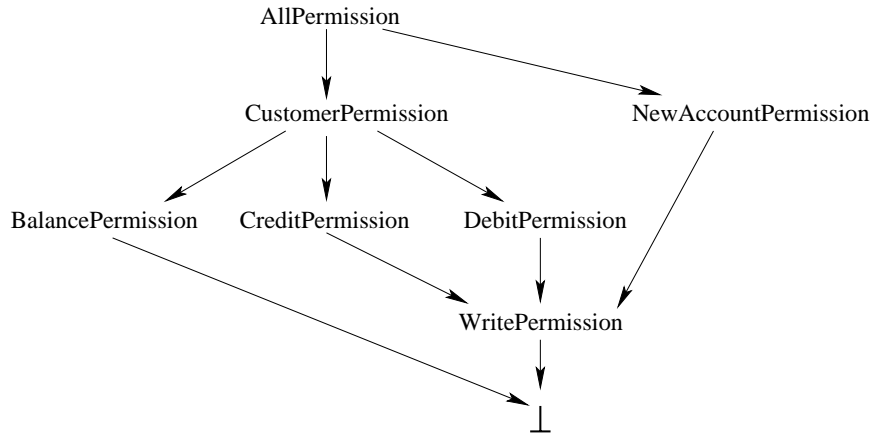


Figure 8: Permissions lattice  $L_{Perm}$  for the online banking example in Figure 1 after `CustomerPermission` permission has been changed to imply permissions `BalancePermission`, `CreditPermission`, and `DebitPermission`.

```
grant signed "bank" {
  permission NewAccountPermission "NewAccountPermission";
  permission BalancePermission "BalancePermission";
};

grant codeBase "http://customer.machine.com/banking/classes/" {
  permission CustomerPermission "CustomerPermission";
  permission BalancePermission "BalancePermission";
  permission DebitPermission "DebitPermission";
  permission CreditPermission "CreditPermission";
};
```

Figure 9: The modified security policy file for the online banking example in Figure 1

In the case of the property violation for the online banking example we described above, the problem is in the implementation of the `implies` method for `CustomerPermission` that, according to the permissions lattice in Figure 5, does not imply any other permissions in this application. Logically, a customer should be allowed access to operations that modify her account. Therefore, we can modify the `implies` method of `CustomerPermission` in such a way that it implies permissions `BalancePermission`, `CreditPermission`, and `DebitPermission`, which results in the new permissions lattice, shown in Figure 8.

Another way in which the detected problem could be fixed is by modifying the security policy and removing the privileged region in the `debit` method of class `AccountWithPermission`. The modified security policy assigns all permissions necessary to use an account to the classes on the client machine. This new security policy appears in Figure 9. This would also involve changing the property of interest to become

$$17 \Rightarrow \{NewAccountPermission\} \vee \{CreditPermission\} \vee \{DebitPermission\} \vee \{CustomerPermission\}$$

### 7.3 Worklist formulation of the PFA algorithm

In this section we give a worklist formulation of the PFA algorithm. Figure 10 gives pseudo-code for this algorithm.

Initialization of the worklist algorithm proceeds according to the initialization of the general PFA algorithm. The worklist  $W$  stores nodes whose  $IN_{priv}$ ,  $priv$ ,  $IN_{unpriv}$ , and  $unpriv$  sets have to be re-computed. Initially, all successors of head nodes of all public methods are placed on the worklist.

In the main loop of the worklist algorithm, a node  $n$  is taken from the worklist<sup>8</sup>. The  $IN_{priv}$  and  $IN_{unpriv}$  sets of  $n$  are computed by merging permission sets for all predecessors of  $n$ , according to equation (2). Equation (3) is then used to obtain  $priv$  and  $unpriv$  sets for  $n$ . If these sets change on this iteration of the algorithm (compared to their values before this iteration started), then all successors of  $n$  are added to the worklist.

### 7.4 Conservativeness

In this section, we prove that the PFA algorithm is conservative. First, we prove that the PCG conservatively models all possible executions of the program under analysis. We extend the mapping  $\mu$  between statements  $Instr$  in the program and sets of PCG nodes  $N$ , introduced in the PCG construction algorithm in Figure 3. The mapping algorithm is shown in Figure 11.

**Lemma 2.** *For each actual execution of the program, represented by a sequence of instructions  $s_0, \dots, s_k$ , there exists a path  $n_0, \dots, n_r$  through the PCG, such that*

- *instruction  $s_0$  is represented by node  $n_0$ ,*
- *instruction  $s_k$  is represented by node  $n_r$ , and*
- *$\forall i, 0 \leq i \leq k - 1$ , if  $s_i$  is represented by  $n_j$ , then  $s_{i+1}$  is represented either by  $n_j$  or  $n_{j+1}$ .*

*Proof.* The proof trivially follows from the mapping algorithm in Figure 11. □

The following theorem states and proves the conservativeness of the information about permissions computed by the PFA algorithm.

**Theorem 3 (Conservativeness).** *If there exists an actual execution of the program under analysis, such that some permission  $p$  is not checked on this path up to instruction  $s$ , then the PFA algorithm does not associate  $p$  with at least one node corresponding to  $s$ .*

*Proof.* Let  $s_0, \dots, s_k$  be an actual execution of the program, such that permission  $p$  is not checked on this execution. Let  $n_0, \dots, n_r$  be the corresponding path through the PCG from Lemma 2. The PFA algorithm has to propagate permission information through this path. We prove the following statement by induction on the number of iterations of the main loop in the worklist version of the algorithm in Figure 10.

$$\forall m \in n_0, \dots, n_r, p \notin priv(m) \vee p \notin unpriv(m) \text{ on iteration number } k \text{ of the algorithm} \quad (4)$$

Statement (4) trivially holds before any iterations of the main loop, since the  $priv$  and  $unpriv$  sets of all nodes in the PCG are initialized to either  $\{\}$  or  $null$ .

As an inductive hypothesis, assume that statement (4) holds on iteration number  $k - 1$  and consider iteration number  $k$ . Let  $q$  be node taken off the worklist on this iteration. If  $q \notin n_0, \dots, n_r$ , then statement (4) holds by inductive hypothesis.

Now let  $q = n_0$ . According to the initialization phase of the algorithm, initially  $unpriv(n_0) = \{\}$ . According to the merge equation (2), this set will not change throughout the algorithm, therefore statement (4) holds for  $q$ .

Finally, let  $q = n_i$  for some  $1 \leq i \leq r$ . By the induction hypothesis,  $p \notin priv(n_{i-1}) \vee p \notin unpriv(n_{i-1})$ , the merge function from equation (2) ensures that statement (4) holds for  $q$ . □

<sup>8</sup>We do not restrict the order in which nodes can be taken from the worklist. Different scheduling algorithms may be appropriate in different situations.

**Algorithm 2 (Worklist PFA algorithm).**

```

Input: A PCG  $(N, H, CE, LE, kind)$  and a lattice  $L_{2^{perm}}$  of permission sets.
Output: A set  $Permissions(n), \forall n \in N$ , containing all permissions that must have been checked by the time the execution gets to  $n$ .
(1) Let  $W$  be an empty worklist with properties of a set
    // initialize the priv and unpriv sets of the head nodes and put successors of the head nodes
    // on the worklist
(2)  $\forall n \in H,$ 
(3)   Set  $priv(n) = null$ 
(4)   Set  $unpriv(n) = \emptyset$ 
(5)    $\forall s \in Succs(n),$ 
(6)     add  $s$  to  $W$ 
    initialize the priv and unpriv sets of all other nodes to be null
    end for
  end for
(7)  $\forall n \in N \setminus H,$ 
(8)   Set  $priv(n) = null$ 
(9)   Set  $unpriv(n) = null$ 
  end for
  // the main loop of the algorithm
(10) while  $(W \neq \emptyset)$ 
    // remove a node from the worklist
(11)   select an arbitrary  $n \in W$ 
(12)    $W = W \setminus \{n\}$ 
    // merge the flow of permission information from the predecessors of  $n$ 
(13)    $IN_{priv}(n) = \prod_{p \in Preds(n)} priv(p)$ 
(14)    $IN_{unpriv}(n) = \prod_{p \in Preds(n)} unpriv(p)$ 
(15)   compute the new priv and unpriv sets for  $n$  according to formula (3)
    // only update the worklist if the sets for  $n$  changed
(16)   if this iteration changed either  $priv(n)$  or  $unpriv(n)$ 
(17)      $W = W \cup Succs(n)$ 
  // compute the final permissions information for all nodes by combining their priv and unpriv sets
  end if
(18)    $\forall n \in N$ 
(19)      $Permissions(n) = priv(n) \cap unpriv(n)$ 
  end while

```

Figure 10: A worklist version of the PFA algorithm

## 7.5 Complexity

In this section, we prove that the worst-case complexity of the PFA algorithm is cubic in the size of the PCG and quadratic in the number of different types of permissions used in the program<sup>9</sup>.

**Theorem 4 (Complexity).** *The worst-case complexity of the PFA algorithm is  $\mathcal{O}(|N|^3 |Permissions|^2)$ .*

*Proof.* In the worst case, each of the two sets of permissions *priv* and *unpriv* associated with each node in the PCG has size  $|Permissions|$ . The successors of node  $n$  are put on the worklist only if either  $priv(n)$  or  $unpriv(n)$  change.

<sup>9</sup>We believe that we can improve the worklist PFA algorithm in Figure 10 in a way that makes the worst-case complexity quadratic in the size of the PCG and linear in the number of different types of permissions.

**Algorithm 3 (Mapping between the PCG and program instructions).**

<p>Input: <i>The PCG</i> <math>(N, H, CE, LE, kind)</math>, <i>program instruction graph</i>, and a <i>partial mapping</i> <math>Instr \rightarrow N</math> from the algorithm in Figure 3.</p> <p>Output: A <i>mapping</i> <math>\mu : Instr \rightarrow 2^N</math>.</p> <p>// initialize the sets of instructions for all PCG nodes</p> <ol style="list-style-type: none"> <li>(1) For each unmapped <math>s \in Instr</math>:</li> <li>(2) For each mapped instruction <math>s'</math>, such that there is a path from <math>s</math> to <math>s'</math> in the program instruction graph, where all instructions on this path, except for <math>s'</math>, are unmapped</li> <li>(3) For each node <math>n \in \mu(s')</math></li> <li>(4) Add <math>n</math> to <math>\mu(s)</math></li> </ol>
---

Figure 11: An algorithm for creating a mapping between PCG nodes and program instructions

The maximal number of times either of these sets can change is  $|Permissions| + 1$  (if first the set changes from *null* to the set containing all permissions in *Permissions* and subsequently permissions are removed from this set one by one on different iterations of the algorithm). Thus, because of changes of sets of  $n$ , each of the successors of  $n$  can be put on the worklist  $\mathcal{O}(|Permissions|)$  times. Since a node in the PCG can have  $\mathcal{O}(|N|)$  predecessors, each node in the PCG can be put on the worklist  $\mathcal{O}(|Permissions||N|)$  times.

On each iteration of the algorithm, a node  $n$  is removed from the worklist and the  $IN_{priv}$ ,  $IN_{unpriv}$ ,  $priv$ , and  $unpriv$  sets of  $n$  are re-computed. Since  $\mathcal{O}(|N|)$  predecessors may be involved in this re-computation, each with  $\mathcal{O}(|Permissions|)$  information, the operations of merging and propagation (2) and (3) take  $\mathcal{O}(|Permissions||N|)$  on each iteration<sup>10</sup>.

So, for each of the  $|N|$  nodes in the PCG, the permissions information can be re-computed  $\mathcal{O}(|Permissions||N|)$  times, each re-computation of complexity  $\mathcal{O}(|Permissions||N|)$ . The statement of the theorem follows.  $\square$

## 8 Conclusions

In this paper we propose a data flow algorithm for computing information about permissions checked in Java programs. For each statement in the program, the algorithm computes a conservative approximation of all permissions that are checked on *all* executions of the program leading to this statement. We assume the presence of simple permissions-based security properties that specify which groups of permissions must be checked before certain security sensitive regions of the program can be executed. Unlike the previously proposed approaches for this problem, our algorithm is low-order polynomial in the size of the program and the number of different types of permissions used in this program.

Our approach relies on the call graph for the program under analysis and therefore is sensitive to the precision of this call graph. We do not believe that very precise call graphs are necessary in our approach in order to obtain sufficiently precise information about permission flow, but this hypothesis remains to be tested experimentally.

The algorithm presented in this paper is context-insensitive. Adding context-sensitivity would theoretically improve the precision of this analysis, while making it less tractable. Our hypothesis is that context-sensitivity would add little in terms of precision, while significantly impacting the performance of the algorithm. We plan to evaluate the need for added precision experimentally. If it turns out that in practice the context-insensitive algorithm often produces imprecise results that could be improved by a context-sensitive algorithm, we will implement such a context-sensitive algorithm.

<sup>10</sup>We assume that all sets used in the algorithm are implemented as look-up tables, so that operations of insertion, deletion, and look-up take constant time.

In our future work, we plan to implement our algorithm and use it in case studies involving checking permission-related security properties of realistic Java programs. We believe that the algorithm will scale well, given its polynomial complexity and the fact that the call graph based model on which the algorithm operates can be reduced. (If no permission check operations are performed in a method and in all method that it calls directly or indirectly, then statements from this method can be omitted from the program model.) In addition, we plan to investigate whether permission-related properties other than of the simple kind we use in this paper are needed.

## References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1988.
- [2] P. Bertelsen. Dynamic semantics of Java bytecode. In *Workshop on Principles of Abstract Machines*, Sept. 1998.
- [3] S.-K. Chin. High-confidence design for security. *Communications of the ACM*, 42(7):33–37, July 1999.
- [4] D. Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, Zurich, Switzerland, Apr. 1997.
- [5] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.
- [6] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.
- [7] W. G. Griswold, M. I. Chen, R. W. Bowdidge, J. L. Cabaniss, V. B. Nguyen, and J. D. Morgenthaler. Tool support for planning the restructuring of data abstractions in large systems. *IEEE Transactions on Software Engineering*, 24(7):534–558, July 1998.
- [8] D. Grove and C. Chambers. A framework for call graph construction algorithms. Research Report 21699 (97756), IBM, Mar. 2000. To appear in *ACM Transactions on Programming Languages and Systems*.
- [9] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.
- [10] N. Heintze and J. G. Riecke. The SLam calculus: Programming with security and integrity. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 365–377, San Diego, California, 19–21 Jan. 1998.
- [11] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages ??–??, June 2001.
- [12] T. Jensen, D. L. Métayer, and T. Thorn. Verification of control flow based security properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (SSP '99)*, pages 89–105. IEEE, May 1999.
- [13] W. A. Landi and B. G. Ryder. Pointer-induced aliasing: A problem taxonomy. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 93–103, Jan. 1991.
- [14] X. Leroy and F. Rouaix. Security properties of typed applets. In *25th Symposium on Principles of Programming Languages (POPL)*. ACM Press, 1998.

- [15] D. Liang, M. Pennings, and M. J. Harrold. Extending and evaluating flow-insensitive and context-insensitive points-to analyses for java. In *Proceedings of the 2001 ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE-01)*, pages 73–79, New York, June 18–19 2001. ACM Press.
- [16] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, 1990.
- [17] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *1998 IEEE Symposium on Security and Privacy (SSP '98)*, pages 186–197, Washington - Brussels - Tokyo, May 1998. IEEE.
- [18] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.
- [19] N. Nitta, Y. Takata, and H. Seki. Security verification of programs with stack inspection. In *Proceedings of the Sixth ACM Symposium on Access Control Models and Technologies*, pages 31–40, 2001.
- [20] K. M. Olender and L. J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan. 1992.
- [21] C. Pusch. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)*.
- [22] Z. Qian, A. Goldberg, and A. Coglio. A formal specification of Java<sup>TM</sup> class loading. *ACM SIGPLAN Notices*, 35(10):325–336, Oct. 2000.
- [23] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, Oct. 1998.
- [24] G. Smith and D. Volpano. Secure information flow in a multi-threaded imperative language. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 355–364, New York, NY, Jan. 1998. ACM.
- [25] R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *25th Annual ACM Symposium on Principles of Programming Languages*, pages 149–160, Jan. 1998.
- [26] I. Sun Microsystems. Java security architecture. <http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-%specT0C.fm.html>, 1998.
- [27] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, Boston, 1998.
- [28] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented*, pages 281–293, 2000.
- [29] D. Volpano, G. Smith, and C. Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(3):167–187, Dec. 1996.
- [30] D. M. Volpano and G. Smith. A type-based approach to program security. In M. Bidoit and M. Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 607–621. Springer-Verlag, 1997.