

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

Gleb Naumovich  
Department of Computer and Information Science  
Polytechnic University  
5 MetroTech Center Brooklyn, NY 11201  
gleb@poly.edu

## ABSTRACT

Open distributed systems are becoming increasingly popular. Such systems include components that may be obtained from a number of different sources. For example, Java allows run-time loading of software components residing on remote machines. One unfortunate side-effect of this openness is the possibility that “hostile” software components may compromise the security of both the program and the system on which it runs. Java offers a built-in security mechanism, using which programmers can give permissions to distributed components and check these permissions at run-time. This security model is flexible, but using it is not straightforward, which may lead to insufficiently tight permission checking and therefore breaches of security.

In this paper, we propose a data flow algorithm for automated analysis of the flow of permissions in Java programs. Our algorithm produces, for a given instruction in the program, a set of permissions that are checked on all possible executions up to this instruction. This information can be used in program understanding tools or directly for 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.

## Keywords

Data flow analysis, static analysis, security, Java, verification.

## 1. INTRODUCTION

Distributed software applications are increasingly open, in the sense that external components can be linked dynamically to a running application. While open applications offer many advantages, they also present the danger of potential vulnerability to attacks by “hostile” external components.

\*This research was partially supported by the National Science Foundation under Grant CCR-0093174.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSTA 2002, Rome, Italy.

Copyright 2002 ACM 1-58113-562-9/02/0007 .....\$5.00.

To date, most work on software security has concentrated in the area of devising mechanisms that can be used by software developers and users to ensure security of applications and systems. While a lot of effort has been devoted to validation of these mechanisms, little work is available on validation of the *use* of these mechanisms by software practitioners. In this paper, we concentrate on validating the use of the Java security mechanism.

Internet applications are increasingly developed in Java, a language whose design addresses the need for security. Java implements a security model under which 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 [9, 37]. 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 easily misused. Its misuse may result in bugs that open a door for attacks involving distributed Java applications. In this paper, we concentrate on statically modeling and analyzing the flow of permissions in Java programs and discovering insufficiently tight checking of permissions.

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 [11] based model of the program, *permission call graph*, in our analysis. Nodes in this graph correspond to method calls and statements relevant to permission checking. With each node in the permission call graph we associate a set of permissions that are checked on all paths terminating in this node. The data flow algorithm propagates these sets through the graph, until a fixed point is reached. After the algorithm terminates, the permission set associated with a node is conservative in the sense that if there is an execution terminating in this node, on which permission  $p$  is not checked, then our algorithm does not put  $p$  in the set associated with this node.

Our algorithm operates on a lattice of permission sets that is based on the `implies` method of the `Permission` class from the Java security model. 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 also cubic in the number of different permission types used in the program.

This paper makes several important contributions in the field of static analysis of Java security. First, we present a graph representation of the permission hierarchy in the program, which can be used directly for discovering certain types of problems in relationships between different permission classes. Second, we introduce a new efficient algorithm for computing the flow of permissions. Finally,

we propose a simple and intuitive specification for properties that describe what permissions have to hold at specific points in the program. These properties can be checked by analyzing the results of running our algorithm on this program.

The next section discusses related work. We briefly overview the Java security model in Section 3. Section 4 describes the permission lattice and Section 5 introduces the program model used in our approach. Section 6 describes our algorithm in detail, presenting the data flow equations and a worklist version of the algorithm, as well as results about its conservativeness and complexity. In Section 7 we discuss the types of permission-based properties that can be checked by analyzing the results of our algorithm. Finally, Section 8 describes conclusions and future work.

## 2. RELATED WORK

Various aspects of security on the programming language level have been studied. Java bytecode verification in particular received a lot of attention [2, 31–33, 36]. The implications of strong typing for security have been investigated in detail [6, 19, 39, 40]. 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” into variables that are available to untrusted users) [13, 25, 35].

Model checking approaches have been successfully applied to verification of security properties of numerous protocols and applications. For example, Mitchell et al. used Mur $\phi$  [7] to verify a variety of security properties for several protocols [23, 24, 34]. Using SPIN [15] for verification of security properties has also been proposed [17]. In general, model-checking has the benefit of being able to check for arbitrary user-specified properties, including security properties. The drawback of model-checking approaches is their prohibitive worst-case complexity bounds. Although studies have shown that model checking techniques scale well for some types of systems [3, 5], in general model checking remains expensive. In contrast, the approach proposed in this paper has low-order polynomial worst-case complexity bounds.

Two approaches that are closely related to this paper have been proposed for analyzing security properties for languages with stack-based security (as in Java). Jensen et al. [16] proposed an approach for specifying and analyzing general security properties for programs with procedural control flow and a stack-based security model. Properties are specified in temporal logic [21] and the algorithm for verification of these properties is based on generating all possible stack configurations. Java-specific permission-based properties similar to those we describe in Section 7 can be specified and checked using the approach of [16]. This approach makes an assumption that there exists an upper bound on the number of method frames on the call stack. Since this approach is based on enumeration of possible stack configurations, the worst-case complexity of its analysis is exponential in the number of method call statements in the program. Nitta et al. [28] extend the approach of [16] in a way that enables checking safety properties specified in a regular language and removes the restriction on the number of method frames on the call stack. The worst-case complexity of their algorithm, however, is still exponential in the number of method call statements.

Our approach is different from [16,28] in several important ways. First, our approach is specific to the Java security model. Second, 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. Third, the approaches of [16,28] are the-

oretically more precise than our approach. Fourth, our technique is based on data flow analysis. Finally, the worst-case complexity of our technique is low-order polynomial in the number of method call and permission checking related statements and the number of different permission types in the program. Therefore, in theory, our approach is more scalable than the previous approaches.

The program model used by our technique is closely related to the call graph. Construction of call graphs for object-oriented programs has been explored in detail [11]. Both lightweight approaches relying on type information alone [38] and more sophisticated approaches relying on a form of points-to analysis [14, 18, 20] have been developed. In this paper, we assume that the program call graph is constructed using one of the existing techniques.

Data flow analysis [12] has been used extensively for statically computing information about programs, with applications in compiler optimization [1], code understanding and visualization (e.g. [10]), and program verification [8, 27, 29]. 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 forward-flow all-paths problem [22]. As a result, the algorithm has a low-order polynomial worst-case behavior.

## 3. PERMISSIONS IN THE JAVA SECURITY MODEL

In this section, we first introduce a Java program that we use throughout the paper for illustrations. After that, we describe the aspects of the Java security model [30] that are relevant to the topic of this paper.

### Example

Figure 1 shows the example we use throughout this paper. This example models a part of an online banking system, where a customer’s account can be linked to other accounts<sup>1</sup>. 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<sup>2</sup> to another account. Internally, those methods of class `Account` that can modify the balance call the private method `write` of this class. This `write` method is responsible for saving the account information persistently (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). This class overrides the `debit` method of class `Account`, so that if debiting an account for the specified amount 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 together. The `debit` method that is called on the checking account exceeds the amount in this 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

<sup>1</sup>This system is based on an example from [28], but has been made more complex and concrete.

<sup>2</sup>To save space, the implementation of the `Money` class is not shown.

```

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();
    }

    private 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);
                    }
                    super.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

protection, and credit card accounts have the balance of \$0 and the savings account has the balance of \$1000.

## Permission Classes

The Java security model relies on stack-based permission checking for access control. A *permission* is a first-class object that has a name and type. Optionally, a permission can have a set of *targets* and *actions*. For example, a file access permission has files and/or directories as targets and actions of reading, writing, executing, or deleting. Both targets and actions are specified as strings and passed as parameters to constructors of permissions. A permission that has no actions and no target is called *named* (its name is the only attribute that identifies this permission; a class either has or does not have a named permission).

A program may use many different classes of permissions, since each permission is created to address only a very specific aspect of security. In many cases, it is convenient to define relationships between different types of permissions. The Java security model supports implication between permissions. Permission  $p_1$  *implies* permission  $p_2$  if any code that is granted permission  $p_1$  is automatically granted permission  $p_2$ . In Java, the implication relation-

```

public class CreditPermission extends BasicPermission {
    ...

    public boolean implies(Permission p) {
        return (p instanceof CreditPermission) ||
            (p instanceof FilePermission);
    }
}

```

Figure 2: Definition of permission class `CreditPermission` for the example in Figure 1.

ship between permissions is defined by method `implies`. Figure 2 shows the definition of the `implies` method of a named permission class `CreditPermission` from our banking example that relies on the standard class `BasicPermission` from the Java security model. According to this definition, a permission of type `CreditPermission` implies any other permission of this type and any permission of type `FilePermission`. The Java security model defines a permission class `AllPermission`, instances of which imply any other permission.

```

grant codeBase "http://bank.machine.com/banking/classes/" {
  permission NewAccountPermission "NewAccountPermission";
  permission BalancePermission "BalancePermission";
};

grant signed "TheBank" {
  permission CustomerPermission "CustomerPermission";
};

```

**Figure 3: The security policy file for the online banking example in Figure 1**

## Granting permissions to code

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 allow code from a very specific URL (e.g. representing a machine in the bank branch office) to check balance and create new accounts. Only code digitally signed by the bank is granted `CustomerPermission`. The security policy file for this example appears in Figure 3<sup>3</sup>. All classes in the standard Java libraries are given an `AllPermission` permission automatically.

## Run-time checking of permissions

To protect a region of code against unauthorized access, a programmer has to insert in the beginning of this region an operation that tells the JVM to check that classes whose methods are on the call stack all have the required permission. This is done by calling either the static `checkPermission` method of class `AccessController` or the `checkPermission` method of instances of class `SecurityManager` or one of its subclasses<sup>4</sup>.

JVM checks permissions at run-time as follows. When executing a call to method `checkPermission` of `AccessController`, the JVM checks all classes that define methods that are currently on the program call stack. If at least one such class is not given a permission that equals to or implies the permission passed to `checkPermission` as a parameter, the JVM throws an exception of type `SecurityException`.

The online banking example defines several application-specific permissions: `NewAccountPermission`, `BalancePermission`, `DebitPermission`, `CreditPermission`, and `CustomerPermission`<sup>5</sup>. 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. Suppose that this method is called for some object of class `Account` from method `m` of class `C1` that is, in turn, called from the `main` method of class `C2`. At the time of the call to `checkPermission` in `credit`, the stack will contain (from bottom up) methods `main`, `m`, and `credit`. During this call to `checkPermission`, the JVM will check whether each of the classes `C2`, `C1`, and `Account` have permission `CreditPermission`. The JVM throws a `SecurityException` if at least one of these three classes does

<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).

<sup>4</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>5</sup>To save space, we do not show the definitions of these permission classes. Each of them simply extends the standard Java `Permission` class, similar to `CreditPermission` in Figure 2.

not have this permission.

Note that, while permissions have to be assigned statically using a policy file, checking permissions in the code is done at run time. 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 instance of a permission class always implies any other instance of this class. We address this issue in detail in Section 4.1.

## Privileged regions

The Java security model allows application programmers to relax the permission requirements for a region of code. Such regions of code are called *privileged*. Let  $m_p$  be a method that contains a privileged region. During execution of the code in the privileged region of  $m_p$ , the JVM marks all frames that were placed on the stack before that of  $m_p$  as privileged. After  $m_p$  is entered and before it is exited, if a `checkPermission` method is called (either from  $m_p$  or one of the methods directly or indirectly called by  $m_p$ ), the JVM does not check the specified permission for the classes that define methods from stack frames marked as privileged. Privileged regions give the programmer additional flexibility in configuring permissions for the application, but, if not used judiciously, may become 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` in Figure 1 contain privileged regions<sup>6</sup>. The `write` method of class `java.io.FileWriter` 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 `FilePermission` permission, but will not run this check for other classes whose methods directly or indirectly call method `write` of class `Account`.

Formally, we say that a Java statement may be inside a privileged region if there exists an execution of the program on which this statement is executed during a call to a `doPrivileged` method. For example, statements inside method `transfer` of class `Account` in the banking example in Figure 1 may be inside a privileged region, because method `transfer` is called from the `run` method that is in turn called when `doPrivileged` method is called in method `debit` of class `AccountWithProtection`.

## 4. PERMISSION LATTICE

In this section, first we discuss several restrictions on the use of permissions, necessary for our algorithm. Then we introduce a lattice of permissions, based on the implication relationships defined in the Java security model. Finally, we use this lattice to define a lattice of permission sets, which is used by our algorithm.

### 4.1 Restrictions on permissions

The goal of our technique is to capture the flow of permissions in the program statically. Since in Java permissions are dynamically created objects, we introduce the following restrictions on permission types:

1. Only named permissions must be used. Permissions that have targets and/or actions can be automatically converted to named permissions if the values of targets and actions are statically defined strings. Our approach effectively replaces

<sup>6</sup>We construct an object that conforms to the `PrivilegedAction` interface in-line and pass it to `doPrivileged`.

a permission class with targets and actions with a number of named permission classes, one for each target-action combination. For example, in our online banking system, we replace permission of type `FilePermission` with target identified by the value in field `persistentLocation` and action “write” with a permission of type `WritePermission` that does not have any targets or actions. In the rest of the paper we assume that this transformation has been performed and therefore all permissions are named.

2. The `implies` method for each permission class must be defined in such a way that a permission object implies all other permission objects of the same class. For example, `CreditPermission` in Figure 2 uses the `instanceof` operation to return `true` if the type of permission passed to the `implies` method is `CreditPermission`.
3. Permission objects must be immutable. This restriction ensures that a permission object is not modified after it is created and before it is passed to a `checkPermission` method.

Effectively, these restrictions allow us to reason about statically defined permission types instead of dynamically defined permission objects. Some of these restrictions can be relaxed by using alias resolution techniques [18], but we believe that in practice the restricted permission model is sufficient in most situations. In the remainder of this paper, we use the term *permission* to refer to all permissions of a particular class, since all such permissions are equivalent. In general, if one of the above restrictions on permission types is violated, the results of our analysis may be incorrect.

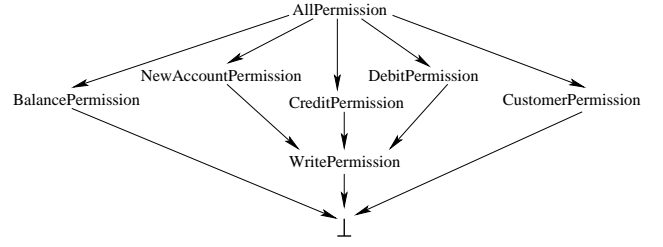
We write  $Permissions$  to denote the set of all permissions used in the program, all permissions directly or indirectly implied by those, and `AllPermission`. For example, for the online banking example in Figure 1,  $Permissions = \{AllPermission, NewAccountPermission, BalancePermission, CreditPermission, DebitPermission, CustomerPermission, WritePermission\}$ .

## 4.2 Lattice of individual permissions

As described in Section 3, every permission class  $p$  has to define an `implies` method that determines what permissions are implied by objects of this class. Given our restrictions on permissions from Section 4.1, we define implication relationships based on permission classes. We say that permission  $p_1$  *directly implies* permission  $p_2$  if  $p_1$  implies  $p_2$  and there does not exist another permission  $p : p \neq p_1, p \neq p_2$ , such that  $p_1$  implies  $p$  and  $p$  implies  $p_2$ . Based on this definition of indirect implication, we introduce a *permission graph* that contains a node for each permission in the program and an edge  $(p_1, p_2)$  if permission  $p_1$  directly implies permission  $p_2$ . By the definition of direct implication,  $p_1$  implies  $p_2$  if and only if there is a path from node  $p_1$  to node  $p_2$  in the graph<sup>7</sup>.

The permission graph for a program is in itself a useful analysis tool, since it visualizes the implication relationship among permissions in this program. This visualization is particularly helpful for large programs that use many related permission types. In addition, the permission graph can be used for detecting problems in definitions of `implies` methods in permission classes. In general, if permission  $p_1$  implies permission  $p_2$ ,  $p_2$  implies  $p_1$ , and  $p_1 \neq p_2$ , it means that  $p_1$  and  $p_2$  are equivalent. Therefore, strongly connected regions in the permission graph may be a symptom of faulty logic in the definition of `implies` methods and should be carefully investigated by software developers and validators.

In this paper, we use the permission graph only as an intermediate step. We collapse all permissions in each strongly connected



**Figure 4: Permission lattice  $L_{Perm}$  for the online banking example in Figure 1**

component in this graph into a single permission. Let  $P$  be a set of all permissions in a strongly connected region in this graph,  $Preds_{PG}$  be the set of all permissions that are not in  $P$  and directly imply at least one permission in  $P$ , and  $Succs_{PG}$  be the set of all permissions that are not in  $P$  and are directly implied by at least one permission in  $P$ . We modify the permission graph by first removing all permissions in  $P$  and their incident edges and then adding a new permission  $p$  and an edge from each permission in  $Preds_{PG}$  to  $p$  and an edge from  $p$  to each permission in  $Succs_{PG}$ . The resulting graph does not have cycles. Set  $Permissions$  is modified accordingly:  $Permissions = (Permissions \setminus P) \cup \{p\}$ .

We obtain a lattice of permissions  $L_{Perm}$  from the cycle-free permission graph by adding a top and bottom elements. The top element  $\top$  for this lattice is `AllPermission`. The node for `AllPermission` is connected with all nodes that do not already have predecessors in the graph. The bottom element  $\perp$  for this lattice represents the absence of any permissions. We connect all nodes that do not have successors in the graph to  $\perp$ . The partial order relation is derived directly from the `implies` method: for two given permissions  $p_1$  and  $p_2$ , if  $p_2.implies(p_1) = true$ , then  $p_1 \sqsubseteq p_2$ . The merge  $\sqcap$  and join  $\sqcup$  operations on  $L_{Perm}$  are defined in the usual way:

$$\begin{aligned}
 p_1 \sqcap p_2 &= p, \text{ where } p \sqsubseteq p_1, p \sqsubseteq p_2, \text{ and} \\
 \forall p' : p' \sqsubseteq p_1, p' \sqsubseteq p_2 &\Rightarrow p' \sqsubseteq p \\
 p_1 \sqcup p_2 &= p, \text{ where } p_1 \sqsubseteq p, p_2 \sqsubseteq p, \text{ and} \\
 \forall p' : p_1 \sqsubseteq p', p_2 \sqsubseteq p' &\Rightarrow p \sqsubseteq p'
 \end{aligned} \tag{1}$$

The permission lattice for the permissions used in our banking example is shown in Figure 4. For example, `WritePermission`  $\sqsubseteq$  `CreditPermission` and `CreditPermission`  $\sqcap$  `DebitPermission` = `WritePermission`.

## 4.3 Lattice of permission sets

We use  $L_{Perm}$  to define a lattice  $L_{2Perm}$  of sets of permissions. We derive a partial order on permission sets in this lattice from the implication-based partial order on individual permissions from Equation (1). At the first glance, it seems that this partial order on permission sets  $P_1, P_2$  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 has a node for each subset of  $Permissions$  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 to the definition above,  $\{p_1\} \sqsubseteq \{p_1, p_2\}$  and  $\{p_1, p_2\} \sqsubseteq \{p_1\}$ . Therefore, nodes  $\{p_1\}$  and  $\{p_1, p_2\}$  cannot be in the same lattice.

We say that a set of permissions  $P \subseteq Permissions$  is *canonical* if no permission in  $P$  implies any other permission in  $P$  except itself:  $\forall p_1, p_2 \in P, p_1 \neq p_2 \Rightarrow p_1 \not\sqsubseteq p_2 \wedge p_2 \not\sqsubseteq p_1$ . We define a *canonical reduction* operation  $CR$  that removes from a permission

<sup>7</sup>A permission always implies itself, which corresponds to a path of length 0.

set all permissions that are implied by some other permission in this set:

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

The proof that for any set of permissions  $P \subseteq \text{Permissions}$  there exists only one corresponding canonical set appears in the technical report version of this paper [26].

We define merge ( $\sqcap$ ) and join ( $\sqcup$ ) operations on permissions sets as follows:

$$\begin{aligned} \forall P_1, P_2 \subseteq \text{Permissions} : \\ P_1 \sqcap P_2 &= CR(\{p_1 \sqcap p_2 | p_1 \in P_1, p_2 \in P_2\}) \\ P_1 \sqcup P_2 &= CR(P_1 \cup P_2) \end{aligned} \quad (2)$$

Now we can define the lattice of permission sets  $L_{2^{perm}}$  as follows. The top element of  $L_{2^{perm}}$  is a set that consists of a single permission `AllPermission`. The bottom element of  $L_{2^{perm}}$  is the empty set. Any two canonical sets of permissions  $P_1$  and  $P_2$  are elements of  $L_{2^{perm}}$ .  $P_1 \sqsubseteq P_2$  iff  $\forall p_1 \in P_1, \exists p_2 \in P_2 : p_1 \sqsubseteq p_2$ . The height of the permission set lattice is at most  $|\text{Permissions}|$  (the proof of this result appears in [26]).

## 5. PERMISSION CALL GRAPH

A *Permission 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 [16, 28]. Each node in PCG corresponds to one of the following actions in the program:

- starting execution of a method,
- 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 is constructed from the set of control flow graphs (CFGs) for all methods in the program or a part of the program under analysis. Each of the CFGs is reduced, so that only nodes corresponding to method calls, including permission checking methods and entering and exiting privileged regions, remain. Then, method call nodes are connected to the head nodes in the methods that may be called by these statements. We can use the standard call graph construction techniques [11] for determining targets of method calls<sup>8</sup>. Thus, edges in a PCG represent both intra- and inter-procedural flow of control.

For the purposes of our technique, we model privileged regions with a pair of nodes, one to represent entering and another to represent 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 a node that represents entering a privileged region with the node that represents exiting this region. We write  $Partner(n_2) = n_1$  and  $Partner(n_1) = n_2$  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.

<sup>8</sup>While, in general, call graphs have to take dynamic class loading in Java into account, the type of analysis described in this paper primarily seeks to validate security of particularly sensitive software components, not the entire distributed application. Therefore, we do not expect that an approximation of calls to methods of dynamically loaded classes will be required.

Figure 5 shows the PCG for the example in Figure 1. To improve readability, we group PCG nodes into boxes that represent methods and classes in the program (the main method appears in Figure 1 only for illustration and thus is not present in the PCG). In the figure, thick arrows from method call nodes to method boxes represent inter-procedural control flow edges from method call nodes to the first nodes of the methods represented by the boxes. To reduce the size of the PCG for this example, we omit calls to methods of class `Money`<sup>9</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.

Formally, a PCG is a tuple  $(N, H, E, kind)$ , where

- $N$  is the set of PCG nodes.
- $H \subseteq N$  is the set of *head nodes* of the PCG. A head node for a method marks the start of this method.
- $E$  is the set of edges, including both edges that represent control flow across methods (these edges go from method call nodes to head nodes of the corresponding method) and edges that represent control flow within methods.
- $kind$  is a labeling function  $N \rightarrow \{\text{call, check, enterPriv, exitPriv, head}\}$  that marks each node as a method call, permission checking, entrance to a privileged region, exit from a privileged region, or a method head node.

We define a function  $checked : N \rightarrow \text{Permissions}$  that returns permissions checked at `check` nodes. For example, for the PCG in Figure 5,  $checked(22) = \text{CustomerPermission}$ . For all non-`check` nodes this function returns the bottom element  $\perp$  from  $L_{Perm}$ . 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.

Removal of strongly connected components in the permission graph, described in Section 4.2, requires a modification of the PCG. For each permission  $p$  that was replaced by permission  $p'$  in the permission graph, we modify function  $checked$  to return  $p'$  for nodes for which it previously returned  $p$ .

For our analysis of permission flow, it is important to identify methods that may be called by hostile classes. We use predicate *Callable* on PCG nodes to distinguish the head nodes of such methods. For simplicity, we assume that all public and protected methods of public classes can potentially be accessed by hostile classes, and so we set the *Callable* predicates of head nodes of these methods to `true`. In reality, it may not be possible for a hostile class to call such methods. Let  $m$  be a public instance method of a public class  $C$ . If a hostile class cannot obtain any instances of class  $C$  from other accessible methods in the program, this hostile class will not be able to call  $m$  directly. It remains to be seen if our simple assumption produces results that are too imprecise in practice. (We believe that it is beneficial to force application programmers to insert permission checks at the beginning of security-sensitive methods, even if currently no dynamically loaded class can call such a method. This methodology may prevent security errors if the application is modified in the future to allow dynamically loaded classes access to such methods.)

## 6. DATA FLOW ALGORITHM FOR COMPUTING PERMISSION FLOW

<sup>9</sup>We assume that methods of class `Money` check no permissions and call (directly or indirectly) no methods that do. This optimization can be generalized, if it is determined that the called method is not important for checking permission-related properties.

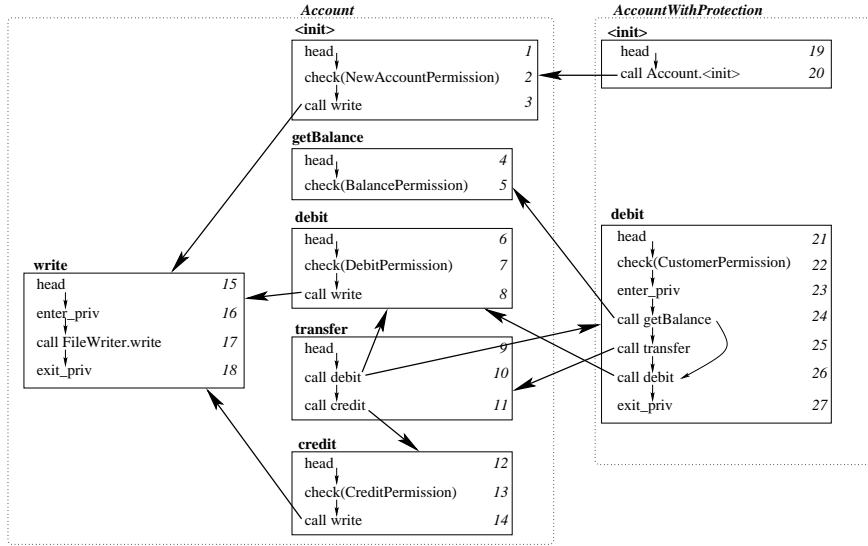


Figure 5: PCG for the online banking example in Figure 1

In this section we describe a forward-flow all-paths data flow analysis for computing the flow of permissions in a program, called *Permission Flow Analysis (PFA)*. For each node in the PCG, we differentiate between two categories of paths that involve this node, paths on which this node is executed inside a privileged region and paths on which this node is executed outside privileged regions. In Section 6.1, we describe the sets associated with each node in the PCG and give data flow equations for computing and propagating these sets. Section 6.2 presents a worklist formulation of this algorithm and Section 6.3 illustrates the algorithm on the banking example. Finally, Section 6.4 presents results about conservativeness and complexity of the PFA algorithm.

## 6.1 Data flow equations

With each node  $n$  in the PCG we associate two sets of permissions,  $priv(n)$  and  $unpriv(n)$ . Set  $priv(n)$  contains all permissions that are checked by the time  $n$  finishes executing for the first time on executions where a privileged region is entered but not exited by the time  $n$  is executed. Set  $unpriv(n)$  contains all permissions that are checked by the time  $n$  finishes executing for the first time on executions where node  $n$  executes outside privileged regions. Initially, both sets are set to  $\{\text{AllPermission}\}$  for all PCG nodes.

The PFA algorithm propagates the  $priv$  and  $unpriv$  sets along the edges of the PCG. If several edges enter a node  $n$ , the permission information flowing into  $n$  is *merged* before being propagated through  $n$ . To do this, we use the merge operation of the permission set lattice  $L_{2^{Perm}}$  to combine sets of permissions from different predecessors. The intuition behind this is that only permissions that are present in all reachable predecessors of  $n$  can be propagated 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:

$$\begin{aligned}
 IN_{priv}(n) &= \begin{cases} \bigcap_{p \in Preds(n)} priv(p) & \text{if } Preds(n) \neq \emptyset \\ \{\text{AllPermission}\} & \text{if } Preds(n) = \emptyset \end{cases} \\
 IN_{unpriv}(n) &= \begin{cases} \bigcap_{p \in Preds(n)} unpriv(p) & \text{if } Preds(n) \neq \emptyset \\ \{\text{AllPermission}\} & \text{if } Preds(n) = \emptyset \end{cases}
 \end{aligned} \tag{3}$$

Propagation function  $prop_n : (2^{Permissions}, 2^{Permissions}) \rightarrow (2^{Permissions}, 2^{Permissions})$  defines the way in which sets of permissions are propagated through node  $n$ . This function is detailed in Figure 6.

Privileged and unprivileged sets of permissions are propagated through the PCG by using equation (3) to compute  $IN_{priv}(n)$  and  $IN_{unpriv}(n)$  sets and the propagation function in Figure 6 to compute  $(priv(n), unpriv(n)) = prop_n(IN_{priv}(n), IN_{unpriv}(n))$ , until  $priv(n)$  and  $unpriv(n)$  stop changing for all nodes  $n$  in the PCG. At this point, the set of permissions  $CheckedThrough(n)$  that are checked on all executions terminating in a given node  $n$  is computed simply as  $priv(n) \sqcap unpriv(n)$ .

## 6.2 Worklist formulation of the PFA algorithm

Figure 7 gives pseudo-code for a worklist-based form of the PFA algorithm. Initialization of the worklist algorithm proceeds according to 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, head nodes of all methods potentially callable by hostile classes are placed on the worklist.

In the main loop of the worklist algorithm, a node  $n$  is taken from the worklist<sup>10</sup>. The  $IN_{priv}$  and  $IN_{unpriv}$  sets of  $n$  are computed by merging permission sets for all predecessors of  $n$ , according to equation (3). Equations from Figure 6 are 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. The algorithm terminates when the worklist becomes empty.

## 6.3 Example

We illustrate the PFA algorithm on the online banking example in Figure 1. For each node  $n$  in the PCG for this example (shown in Figure 5), we set initially  $priv(n) = unpriv(n) = \{\text{AllPermission}\}$ . The only private method in the two classes in this example is `write`, so we set predicate *Callable* to `true` for head nodes 1, 4, 6, 9, 12, 19, and 21 and to `false` for head node 15.

Figure 8 shows the results of running our algorithm on the PCG

<sup>10</sup>We do not restrict the order in which nodes can be taken from the worklist.

Condition	Value of $prop_n(IN_{priv}(n), IN_{unpriv}(n))$	Explanation
$kind(n) = \text{head}$ and $Callable(n) = \text{false}$	$(IN_{priv}(n), IN_{unpriv}(n))$	If a method cannot be called directly by hostile classes, the sets of all permissions reaching the point after the head node of this method are the same as the sets of all permissions reaching the point before this node.
$kind(n) = \text{head}$ and $Callable(n) = \text{true}$	$(IN_{priv}(n), \emptyset)$	If a method can be called directly by hostile classes, we conservatively assume that it is possible that these classes hold no permissions. Therefore, the set of permissions that are checked on all executions where the head node of this method is executed outside privileged regions is empty.
$kind(n) = \text{call}$	$(IN_{priv}(n), IN_{unpriv}(n))$	If a node represents a method call, the sets of all permissions reaching the point after the head node of this method are the same as the sets of all permissions reaching the point before this node.
$kind(n) = \text{check}$	$(IN_{priv}(n), IN_{unpriv}(n)) \sqcup \{\text{checked}(n)\}$	If a node represents checking of a permission, this permission is definitely checked after this node on all executions where this node appears outside privileged regions. For executions where this node appears inside privileged regions, the checking statement has no effect because of the privileged region semantics.
$kind(n) = \text{enterPriv}$	$(IN_{priv}(n) \sqcap IN_{unpriv}(n), \{\text{AllPermission}\})$	If a node represents an entrance to a privileged region, the point in the execution immediately after this node is executed is always inside a privileged region. We use the set $\{\text{AllPermission}\}$ as the set of all permissions checked on all executions where this node appears outside privileged regions to accommodate merging of this information for computing sets of permissions of successors of this node, according to rules (3).
$kind(n) = \text{exitPriv}$	$(IN_{priv}(\text{Partner}(n)), IN_{unpriv}(\text{Partner}(n)))$	If a node represents an exit from a privileged region, the point in the execution immediately after this node may or may not be outside privileged regions (privileged regions can be nested). Because permission checking is in effect “turned off” by privileged regions, we know that the set of permissions checked immediately after a privileged region is exited is the same as the set of permissions checked immediately before this privileged region is entered. Therefore, we copy the sets of permissions associated with the partner <code>enterPriv</code> node of this <code>exitPriv</code> node.

Figure 6: Propagation function

<p><b>Input:</b> A PCG <math>(N, H, E, kind)</math> and a lattice <math>L_{\mathcal{P}erm}</math> of permission sets.  <b>Output:</b> For each node <math>n \in N</math>, a set containing all permissions that must have been checked by the time the execution gets to <math>n</math>.</p> <pre> // initialize the priv and unpriv sets of all nodes (1) for all <math>n \in N</math>, (2)   set <math>priv(n) = unpriv(n) = \{\text{AllPermission}\}</math> end for // initialize the worklist (3) let <math>W</math> be an empty worklist with properties of a set (4) set <math>W = \emptyset</math> (5) for all <math>n \in H</math> (6)   if <math>Callable(n)</math> (7)     <math>W = W \cup \{n\}</math> end if end for // the main loop of the algorithm (8) while <math>(W \neq \emptyset)</math> // remove a node from the worklist (9)   select an arbitrary <math>n \in W</math> (10)  <math>W = W \setminus \{n\}</math> // merge the flow of permission information from the // predecessors of <math>n</math> (11)  compute sets <math>IN_{priv}(n)</math> and <math>IN_{unpriv}(n)</math> according to (3) (12)  compute the new <math>priv</math> and <math>unpriv</math> sets for <math>n</math> according to equations in Figure 6 // only update the worklist if the sets for <math>n</math> changed (13)  if this iteration changed either <math>priv(n)</math> or <math>unpriv(n)</math> (14)    <math>W = W \cup Succs(n)</math> end if end while (15) for all <math>n \in N</math> (16)  <math>CheckedThrough(n) = priv(n) \sqcap unpriv(n)</math> end for </pre>
--

Figure 7: A worklist version of the PFA algorithm

for the online banking example from Figure 5. For each node  $n$ , the pair of permission sets  $(priv(n), unpriv(n))$  is shown next to it (in some cases, to reduce clutter, the same pair of sets is re-used for several nodes).

Consider the path  $21 \rightarrow 22 \rightarrow 23 \rightarrow 24 \rightarrow 25 \rightarrow 9 \rightarrow 10 \rightarrow 21 \rightarrow 22$ . According to formula (3),  $IN_{priv}(21) = priv(10) = \{\text{AllPermission}\}$  and  $IN_{unpriv}(21) = unpriv(10) = \{\text{AllPermission}\}$  (node 10 is the only predecessor of node 21). According to equations in Figure 6,  $priv(21) = \{\text{AllPermission}\}$  and  $unpriv(21) = \emptyset$  (since node 21 is *Callable*). Since the *unpriv* set of 21 changed, its successor, node 22, is added to the worklist. When node 22 is taken from the worklist,  $IN_{priv}(22) = priv(21) = \{\text{AllPermission}\}$  and  $IN_{unpriv}(22) = unpriv(22) = \emptyset$ . Propagating these sets through node 22, we obtain  $priv(22) = \{\text{AllPermission}\}$  and  $unpriv(22) = \{\text{CustomerPermission}\}$ . For node 23,  $IN_{priv}(23) = priv(22) = \{\text{AllPermission}\}$  and  $IN_{unpriv}(23) = unpriv(22) = \{\text{CustomerPermission}\}$ . According to equations in Figure 6,  $priv(23) = \{\text{AllPermission}\} \sqcap \{\text{CustomerPermission}\} = \{\text{CustomerPermission}\}$  and  $unpriv(23) = \{\text{AllPermission}\}$ . These sets are propagated without change through nodes 24 and 25. Node 9 is *Callable*, so  $priv(9) = \{\text{CustomerPermission}\}$  and  $unpriv(9) = \emptyset$ . These sets are propagated without change through node 10. When sets of node 21 are re-computed,  $IN_{priv}(21) = priv(10) = \{\text{CustomerPermission}\}$ ,  $priv(21) = \{\text{CustomerPermission}\}$ , and  $unpriv(21) = \emptyset$ . After propagating these sets through node 22, we obtain  $priv(22) = \{\text{CustomerPermission}\}$  and  $unpriv(22) = \{\text{CustomerPermission}\}$ . This reflects the fact that there are executions on which node 22 is executed inside a privileged region (via a recursive call sequence where method `debit` of `AccountWithProtection` calls method `debit` of `Account`, which again calls `debit` of `AccountWithProtection`) and outside a privileged region. In both cases, permission `CustomerPermission` is always checked by the time the code for this node is executed.

## 6.4 Conservativeness and complexity

We proved that the PFA algorithm is conservative in the sense that if a permission  $p$  is not checked on at least one execution terminating in node  $n$ , then after the PFA algorithm terminates,  $p \notin \text{CheckedThrough}(n)$ . This proof first shows that the PCG conservatively models all possible executions of the program under analysis. Then a mapping between the program instructions and sets of PCG nodes is created and used in an argument by induction on the length of paths through the PCG.

We also proved that the worst-case complexity of the PFA algorithm is  $\mathcal{O}(|N|^3 |\text{Permissions}|^3)$ . This result uses the fact that the  $\text{priv}$ ,  $\text{unpriv}$ ,  $IN_{\text{priv}}$ , and  $IN_{\text{unpriv}}$  sets for all nodes are non-increasing according to the partial order of the permission set lattice. Therefore, the number of times a node can be placed on the worklist is bounded by the product of the number of predecessors of a node ( $\mathcal{O}(|N|)$ ) and the number of times the  $\text{priv}$  and  $\text{unpriv}$  sets for each of the predecessors can change (for each predecessor, this number is bounded by the height of the permission set lattice,  $|\text{Permissions}|$ ). The worst-case complexity bound for the algorithm is based on the worst-case bounds on all operations performed on each iteration of the algorithm. The complete proofs of conservativeness and complexity bounds of the PFA algorithm appear in the technical report version of this paper [26].

## 7. PERMISSION-BASED PROPERTIES

We can use the results of the PFA algorithm directly to answer simple questions of the form “is permission  $p$  checked on all executions terminating in node  $n$ ”? For example, the results of running the PFA algorithm for the banking system show that permission `CustomerPermission` is checked on all executions terminating in node 27.

In addition to checking such simple properties, with only a little extra work we can check properties of the form “on all executions terminating in node  $n$ , one of permissions  $p_1, \dots, p_k$  must be checked”. Without loss of generality, assume that set  $\{p_1, \dots, p_k\}$  is canonical. For example, for the banking system, we would like to check that on all executions terminating in node 16, at least one of permissions `NewAccountPermission`, `DebitPermission`, or `CreditPermission` is checked. We cannot use the results of the PFA algorithm for this system to check this property directly. The reason for this is that the algorithm uses the merge operation on the lattice to compute the flow of permissions into PCG nodes. Consider a simple property stating that on all executions terminating in  $n$ , either permission  $p_1$  or permission  $p_2$  must be checked, where  $p_1 \sqcap p_2 = \emptyset$ . Suppose that there are two executions terminating in  $n$  and on one of these executions, permissions  $\{p_1\}$  are checked, while on the other execution,  $\{p_2\}$  are checked. The merge of the two sets is an empty set, based on which we are forced to make the conclusion that the property does not hold, while in reality it does. To avoid this imprecision, for each property  $R$  of the form “on all executions terminating in  $n$ , at least one of permissions  $p_1, \dots, p_k$  holds”, we introduce a new permission  $p_R$ . This permission is added to the permission lattice  $L_{2^{\text{perm}}}$  automatically in the following way:

1. Each of the permissions  $p_1, \dots, p_k$  implies  $p_R$ :  $\forall i, 1 \leq i \leq k, p_i \sqsubseteq p_R$
2.  $p_R$  should directly imply the merge of  $p_1, \dots, p_k$ :  $(\bigcap_{1 \leq i \leq k} p_i) \sqsubseteq p_R$ . Note that because of the assumption that set  $\{p_1, \dots, p_k\}$  canonical,  $p_R$  is distinct from all  $p_1, \dots, p_k$ .

Figure 9 shows the permission lattice  $L_{\text{perm}}$  for the banking example, obtained from the permission lattice in Figure 4 by adding

a permission  $p_{16}$  used for checking the following property:

$$\begin{aligned} & \text{On all executions terminating in node 16, at least one of} \\ & \text{permissions } \text{NewAccountPermission}, \\ & \text{CreditPermission, DebitPermission holds} \end{aligned} \quad (4)$$

Consider the way the  $IN_{\text{priv}}$  and  $IN_{\text{unpriv}}$  sets for node 15 are computed:  $IN_{\text{priv}}(15) = \text{priv}(3) \sqcap \text{priv}(8) \sqcap \text{priv}(14) = \{\text{AllPermission}\} \sqcap \{\text{CustomerPermission}\} \sqcap \{\text{CustomerPermission}\} = \{\text{CustomerPermission}\}$ ,  $IN_{\text{unpriv}}(15) = \text{unpriv}(3) \sqcap \text{unpriv}(8) \sqcap \text{unpriv}(14) = \{\text{NewAccountPermission}\} \sqcap \{\text{DebitPermission}\} \sqcap \{\text{CreditPermission}\} = \{p_{16}\}$ . Using formulas (3) and equations from Figure 6, we propagate these sets to node 16, obtaining  $\text{priv}(16) = \{\text{CustomerPermission}\} \sqcap \{p_{16}\} = \emptyset$  and  $\text{unpriv}(16) = \{\text{AllPermission}\}$ . The merge of these two sets is empty and therefore we conclude that property (4) does not hold for the banking example.

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 caused by the fact that 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.

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 permission lattice in Figure 9, 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 permission lattice, shown in Figure 10. Running the PFA algorithm using this lattice results in  $\text{priv}(16) = \{p_{16}\}$  and  $\text{unpriv}(16) = \{\text{AllPermission}\}$ . Merging these two sets results in  $\{p_{16}\}$ , and so property (4) is satisfied.

At present we are investigating the possibility of checking properties of the form “on all executions terminating in node  $n$ , all properties in at least one of the sets of properties  $P_1, \dots, P_k$  holds”. Checking such properties cannot be enabled by a simple modification of the permission lattice. We believe that such properties can be handled by augmenting the PFA algorithm to reason about properties directly, similar to model checking [4] approaches. The augmented PFA algorithm would propagate through the PCG information about whether the property or parts of the property hold in individual nodes. At present, it is not clear if properties of this form would be very useful in practice.

## 8. CONCLUSIONS

In this paper, we have proposed a data flow algorithm for computing information about permissions checked in Java programs. For each statement in the program, our algorithm computes a conservative approximation of all permissions that are checked on all executions of the program that terminate in this statement. We assume that simple permissions-based security properties are used,

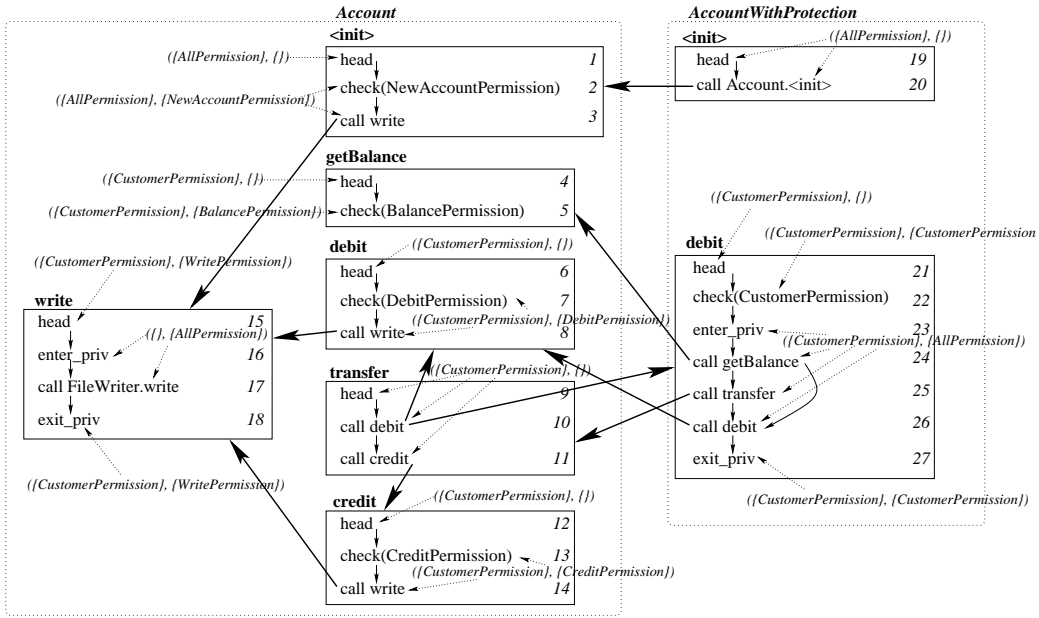


Figure 8: The PCG for the online banking example with  $(priv, unpriv)$  permission set pairs shown for each node.

specifying a set of permissions, such that at least one of permissions in this set must be checked on all executions terminating in a certain point in the program. Unlike the previously proposed approaches [16,28] 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.

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 ability to reduce size of the call graph-based model (if no permission check operations are performed in a method and in all methods that it calls directly or indirectly, then statements from this method can be omitted from the program model). An analyst using this tool would need to supply only simple properties that specify what permissions must be checked on all executions to specific points in the program. The tool would automatically build a representation of all permissions used by the program, incorporate information about the properties in this representation, extract an analysis model from the source code for the (partial) program, and run the analysis. We plan to investigate whether permission-related properties other than of the simple kind we use in this paper are needed.

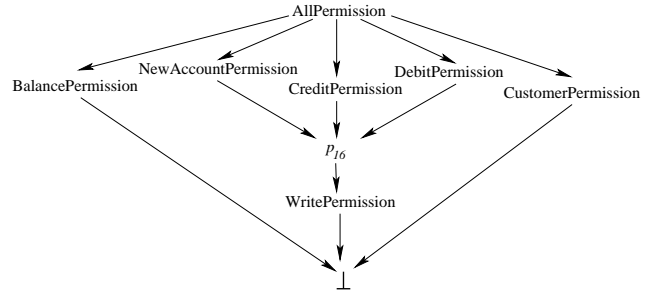


Figure 9: Permission lattice  $L_{Perm}$  for the online banking example in Figure 1, modified to accommodate property in Equation (4).

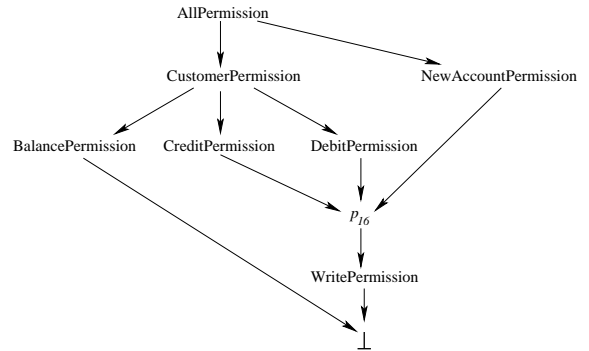


Figure 10: Permission lattice  $L_{Perm}$  for the online banking example in Figure 1 after CustomerPermission has been changed to imply permissions BalancePermission, CreditPermission, and DebitPermission.

## Acknowledgments

We are grateful to Phyllis Frankl, David Chase, Deng Yuetang, and other members of the Software Engineering group at Poly for help-

ful discussions of this work. We also thank anonymous ISSTA reviewers for insightful comments and suggestions for improvement of this paper.

## 9. 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] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *ISSTA 98: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 102–112, Mar. 1998.
- [4] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions of Programming Languages and Systems*, 8(2):244–263, Apr. 1986.
- [5] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.
- [6] D. Dean. The security of static typing with dynamic linking. In *Proceedings of the Fourth ACM Conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, Apr. 1997.
- [7] D. L. Dill. The murphi verification system. In *Proceedings of the Eighth International Conference on Computer Aided Verification*, pages 390–393, July/Aug. 1996.
- [8] 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.
- [9] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, June 1999.
- [10] 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.
- [11] 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*.
- [12] M. S. Hecht. *Flow Analysis of Computer Programs*. North-Holland, New York, 1977.
- [13] 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.
- [14] 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 54–61, June 2001.
- [15] G. J. Holzmann. The model checking SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [16] 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, May 1999.
- [17] A. Josang. Security protocol verification using SPIN. In *Proceedings of the First SPIN Workshop*, 1995.
- [18] 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.
- [19] X. Leroy and F. Rouaix. Security properties of typed applets. In *25th Symposium on Principles of Programming Languages (POPL)*, pages 391–403. ACM Press, January 1998.
- [20] 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, June 2001.
- [21] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer, New York, 1992.
- [22] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, 28(2):121–163, 1990.
- [23] J. C. Mitchell, M. Mitchell, and U. Stern. Automated analysis of cryptographic protocols using mur $\phi$ . In *Proceedings of the 1997 Conference on Security and Privacy*, pages 141–153, May 1997.
- [24] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 201–216, Jan. 1998.
- [25] 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.
- [26] G. Naumovich. A conservative algorithm for computing the flow of permissions in java programs. Technical Report TR-CIS-2001-07, Polytechnic University, Brooklyn, Dec. 2001. <http://cis.poly.edu/tr/tr-cis-2001-07.shtml>.
- [27] 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.
- [28] 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.
- [29] K. M. Olander and L. J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, Jan. 1992.
- [30] M. Pistoia, D. F. Reller, D. Gupta, M. Nagnur, and A. K. Ramani. *Java 2 Network Security*. Prentice-Hall, Aug. 1999.
- [31] 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)*.
- [32] 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.
- [33] E. Rose and K. H. Rose. Lightweight bytecode verification. In *Workshop "Formal Underpinnings of the Java Paradigm", OOPSLA'98*, Oct. 1998.
- [34] V. Shmatikov and J. Mitchell. Analysis of a fair exchange protocol. In *Proceedings of the Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 119–128, Feb. 2000.
- [35] 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.
- [36] 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.
- [37] Sun Microsystems. Java security architecture. <http://java.sun.com/products/jdk/1.2/docs/guide/security/spec/security-%specT0C.fm.html>, 1998.
- [38] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Conference on Object-Oriented*, pages 281–293, 2000.
- [39] 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.
- [40] 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.