Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection

Marco Pistoia[1], Robert J. Flynn[2], Larry Koved[1], and Vugranam C. Sreedhar[1]

[1] IBM Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA
{pistoia, koved, vugranam}@us.ibm.com
http://www.research.ibm.com/javasec

[2] Polytechnic University, 6 Metrotech Center, Brooklyn, NY 11201, USA
flynn@poly.edu
http://www.poly.edu

Abstract. In Java 2 and Microsoft .NET Common Language Runtime (CLR), trusted code has often been programmed to perform access-restricted operations not explicitly requested by its untrusted clients. Since an untrusted client will be on the call stack when access control is enforced, an access-restricted operation will not succeed unless the client is authorized. To avoid this, a portion of the trusted code can be made "privileged." When access control is enforced, privileged code causes the stack traversal to stop at the trusted code frame, and the untrusted code stack frames will not be checked for authorization. For large programs, manually understanding which portions of code should be made privileged is a difficult task. Developers must understand which authorizations will implicitly be extended to client code and make sure that the values of the variables used by the privileged code are not "tainted" by client code. This paper presents an interprocedural analysis for Java bytecode to automatically identify which portions of trusted code should be made privileged, ensure that there are no tainted variables in privileged code, and detect "unnecessary" and "redundant" privileged code. We implemented the algorithm and present the results of our analyses on a set of large programs. While the analysis techniques are in the context of Java code, the basic concepts are also applicable to non-Java systems with a similar authorization model.

1 Introduction

The Java 2 [28, 29, 17] and Microsoft .NET Common Language Runtime (CLR) [16] programming models are extensively used in different kinds of Internet applications. In such applications, it is essential that, when access to a restricted resource is attempted, all code currently on the call stack is authorized to access that resource. In Java 2, when access to a restricted resource is attempted, the `SecurityManager`, if active, triggers access-control enforcement by invoking `AccessController.checkPermission()`. This method takes a `Permission` object `p` as a parameter and performs a call-stack walk to verify that each caller in the current thread of execution has been granted the authorization represented by `p`. In CLR, the call-stack walk is performed by the `Demand()` method. In both platforms, a `SecurityException` is thrown if the declaring class of any one of the methods on the call stack does not have the appropriate authorization.

Often, however, trusted code has been programmed to perform access-restricted operations—such as writing to a log file—that its untrusted client did not explicitly request. Since the untrusted client will be on the call stack when access control is enforced, the operation will not succeed unless the client code is authorized as well. To avoid authorizing the client, which would constitute a violation of the Principle of Least Privilege [32], the portion of trusted code performing the restricted operation must be made *privileged*. In Java 2, this is done by wrapping that portion of trusted code into a call to `AccessController.doPrivileged()`. In CLR, the same result can be obtained by having the trusted code call the `Assert()` method. When access control is enforced, privileged code causes the call-stack walk to stop at the stack frame where `doPrivileged()` is invoked. As a result, client code is implicitly granted the right to perform the restricted operation while the current thread is executing.

Taking preexisting trusted code and understanding which portions of it should be made privileged is a difficult task. It is even more challenging when the trusted code is large or complex. Besides identifying the blocks of it that require authorizations, developers must understand which access rights the privileged code will implicitly grant to client code, and make sure that the variables used by the privileged code to access restricted resources are not *tainted*, meaning that their values cannot be arbitrarily influenced by the client code [35]. For example, if the privileged code is responsible for logging to a file, the name of the log file should not be tainted. Otherwise, an untrusted caller could invoke that privileged code and modify any file in the file system. As we shall see, a tainted variable can be considered *sanitized* if it satisfies certain preconditions.

This paper presents an interprocedural analysis for Java bytecode to solve the following problems:

1. Identify portions of trusted code that should be made privileged, with three objectives in mind:
   (a) Respect the Principle of Least Privilege by preventing unnecessary authorization requirements from propagating to client code
   (b) Ensure that no unnecessary `SecurityExceptions` are thrown due to the client's being insufficiently authorized
   (c) Ensure that there are no tainted variables in privileged code, unless they have been previously sanitized
2. Automatically detect if a tainted variable is *malicious* (used inside privileged code to access a restricted resource) or otherwise *benign*
3. Detect existing "unnecessary" and "redundant" privileged blocks of code and avoid introducing new ones

Privileged code is *unnecessary* if there is no path from it to any authorization check, and it is *redundant* if all the authorization checks it leads to are dominated by other privileged code. Unnecessary or redundant privileged code may lead to violations of the Principle of Least Privilege, especially as a result of subsequent code maintenance, and can be expensive from a performance point of view.

The rest of this section further discusses why privileged-code and tainted-variable analysis is important and summarizes the key contributions of this paper.

1.1 Trusted Code Access Control

When client code makes a call into trusted code, the trusted code often accesses restricted resources that the client never intended to, nor does it need to, directly access. For instance, assume that a Java program is authorized to open a network socket. To do so, it invokes `createSocket()` on the `LibraryCode` trusted class in Figure 1. As its code shows, on opening a socket on behalf of a client program, the trusted code is programmed to log the socket operation to a file. According to the Java 2 access-control model, both the trusted code and its client will need to be granted the `FilePermission` to modify the log file and the `SocketPermission` to create the socket connection, even though the client did not explicitly request to write to the log file. Granting the client code the access right to modify the log file would violate the Principle of Least Privilege. One way to circumvent this problem is to mark the portion of the trusted code responsible for logging as privileged. This prevents the call-stack inspection for the log operation from going beyond the `createSocket()` method, and temporarily exempts the client from the `FilePermission` requirement during the execution of `createSocket()`.

From a practical point of view, a Java developer must implement either the `PrivilegedExceptionAction` or `PrivilegedAction` interface, depending on whether the privileged code could throw a checked `Exception` or not, respectively. Both these interfaces have a `run()` method that, once implemented,

```
import java.io.*;
import java.net.*;
public class LibraryCode {
    private static String logFileName = "audit.txt";
    public static Socket createSocket(String host, int port)
            throws UnknownHostException, IOException {
        // Create the Socket
        Socket socket = new Socket(host, port);
        // Log the Socket operation to a file
        FileOutputStream fos = new FileOutputStream(logFileName);
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        PrintStream ps = new PrintStream(bos, true);
        ps.print("Socket " + host + ":" + port);
        return socket;
    }
}
```

Fig. 1. Library Code Propagating Authorization Requirements to Its Clients

```
import java.io.*;
import java.net.*;
import java.security.*;
public class LibraryCode2 {
    private static final String logFileName = "audit.txt";
    public static Socket createSocket(String host, int port) throws
            UnknownHostException, IOException, PrivilegedActionException {
        // Create the Socket
        Socket socket = new Socket(host, port);
        // Log the Socket operation to a file using doPrivileged()
        File f = new File(logFileName);
        PrivWriteOp op = new PrivWriteOp(host, port, f);
        FileOutputStream fos = (FileOutputStream)
                AccessController.doPrivileged(op);
        BufferedOutputStream bos = new BufferedOutputStream(fos);
        PrintStream ps = new PrintStream(bos, true);
        ps.print("Socket " + host + ":" + port);
        return socket;
    }
}
class PrivWriteOp implements PrivilegedExceptionAction {
    private File f;
    PrivWriteOp (File f) {
        this.f = f;
    }
    public Object run() throws IOException {
        return new FileOutputStream(f);
    }
}
```

Fig. 2. Library Using Privileged Code

must contain the portion of trusted code performing the restricted operation not directly requested by the client. Next, the `PrivilegedExceptionAction` or `PrivilegedAction` instance is passed as a parameter to the `doPrivileged()` method, which will invoke the instance's `run()` method. Class `LibraryCode2` in Figure 2 is obtained by modifying class `LibraryCode` in Figure 1. The main modification consists of wrapping the call to the `FileOutputStream` constructor in a privileged block to prevent client code from requiring a `FilePermission`.

Frequently, code is not written with security as a concern, or it is written to run on a version of the Java Runtime Environment (JRE) prior to 1.2.[1] When a Java 2 `SecurityManager` is finally turned on for a particular application, `SecurityExceptions` are thrown due to access control violations. It can be very difficult to understand which portions of trusted code should be made privileged. In practice, this problem is solved empirically. The developer tests the trusted code with sample client code that makes calls into the trusted code. Typically,

[1] The Java 2 fine-grained access control model was introduced in version 1.2.

the client code is granted only a limited number of access rights, while the trusted code is granted sufficient authorizations, such as `AllPermission`. The developer then notes all the `SecurityException`s generated when running the test cases and distinguishes between two categories of `SecurityException`s:

1. The `SecurityException`s due to the client code's attempting to access some protected resources through the trusted code without the adequate authorizations
2. The `SecurityException`s due to the trusted code's attempting to access some restricted resources on its own without using privileged code

Eliminating a `SecurityException` of Category 2 requires inspecting the trusted source code, identifying which portion of it is responsible for accessing the restricted resource, and making that portion privileged. A `SecurityException` of Category 1 can instead be eliminated by granting the client code the necessary access rights, but this operation must be performed cautiously because granting authorizations to the client could hide `SecurityException`s of Category 2. Manually performing this task is difficult, tedious, and error-prone. After modifying the trusted code or the client security policy, the developer must rerun the test cases. This process must be repeated, possibly many times, until there are no more authorization failures. Additionally, `doPrivileged()` requirements in the trusted code may remain undiscovered due to an insufficient number of test cases, which makes production code potentially unstable.

1.2 Tainted Variables

Another security concern when inserting `doPrivileged()` calls is the risk that the privileged code uses tainted variables to access restricted resources. Consider, for example, the `GetSocket()` utility class shown in Figure 3. Both `host` and `port` are tainted variables, since an untrusted client can arbitrarily set them. Their use in privileged code to open a socket makes them a potential security risk. Conversely, variable `userName`, though tainted and used in privileged code, is benign since its value is not used to access a restricted resource.

Sometimes, it may be necessary to use tainted variables inside privileged code to access restricted resources. In such cases it is important to perform *sanity checks* on those variables to verify that they satisfy certain preconditions [3]. For example, in the code of Figure 3, the programmer could *sanitize* `host` and `port` and make them untainted by refusing to execute the privileged code if, for example, the `host` value does not end with `.edu` and the `port` value is different from `443`.

In general, manually ensuring that no malicious tainted variables are used inside privileged code is time consuming and error prone. It requires:

1. Identifying all the unsanitized malicious tainted variables (`host` and `port` in Figure 3) and separate them from the benign ones (`userName`)
2. Determining all the control- and data-flow paths in the execution of the program that would allow an unsanitized malicious tainted variable to be used inside some privileged code to access restricted resources

```
import java.net.*;
import java.security.*;
public class GetSocket {
    public static Socket getSocket(final String host, final int port,
            final String userName) throws Exception {
        Socket s;
        PrivOp op = new PrivOp(host, port, userName);
        try {
            s = (Socket) AccessController.doPrivileged(op);
        }
        catch (PrivilegedActionException e) {
            throw e.getException();
        }
        return s;
    }
}
class PrivOp implements PrivilegedExceptionAction {
    private String host, userName;
    int port;
    PrivOp(String host, int port, String userName) {
        this.host = host;
        this.port = port;
        this.userName = userName;
    }
    public Object run() throws Exception {
        System.out.println("Received request from user " + userName);
        return new Socket(host, port);
    }
}
```

Fig. 3. Helper `getSocket()` Method with Tainted Parameters

Having a tool that automatically determines if code candidate to become privileged uses unsanitized, malicious tainted variables to access restricted resources helps when deciding whether making that code privileged is appropriate. For example, the code of Figure 1 has two instructions that could be made privileged:

1. `Socket socket = new Socket(host, port);`
2. `FileOutputStream fos = new FileOutputStream(logFileName);`

If Instruction 1 is made privileged, then parameters `host` and `port`, which are tainted, will constitute a security exposure since they are used to access a restricted resource. This is an indication that Instruction 1 should not be made privileged. Conversely, parameter `logFileName` is not tainted. This is an indication that Instruction 2 could be made privileged.

1.3 Contributions

From a privileged-code analysis perspective, the set of code components involved in the execution of a program is logically partitioned into three disjoint subsets:

1. The *fixed components*, which normally include the JRE libraries and are not suitable, or candidates, for modification
2. The *modifiable components*, which are those considered for modification and privileged-code placement, and are typically trusted
3. The *client components*, which make calls into the modifiable components and are often not available at analysis time

This paper presents an interprocedural privileged-code placement and tainted-variable analysis algorithm. The algorithm assumes that both the sets of fixed and modifiable components are available to the analysis, whereas the presence of the client components is statically modelled. The interprocedural analysis described in this paper achieves the following results:

1. For each authorization check triggered by a modifiable component, the analysis identifies the modifiable component's code location that, from a control-flow perspective, is the closest to the authorization check, which minimizes the risks of violating the Principle of Least Privilege.
2. Code locations candidate for becoming privileged are identified with a precision that goes to the level of the program counter within a method (and the source-code line number where available).
3. The analysis provides an explanation as to why a call to `doPrivileged()` is recommended or not.
4. The analysis detects which authorizations will be implicitly granted to client code as a result of calling `doPrivileged()`.
5. The analysis minimizes the risks of introducing unnecessary or redundant privileged code in a modifiable component.
6. If unnecessary or redundant privileged code is already present in a modifiable component, the analysis will detect it.
7. The analysis distinguishes between malicious and benign tainted variables.
8. The analysis detects if unsanitized malicious tainted variables are used in privileged code to access restricted resources.

We implemented this analysis framework in a security-analysis tool called Mandatory Access Rights Certification of Objects (MARCO). In this paper, we present our experience in using MARCO on a set of components, some of which contained more than 20,000 classes. While the analysis techniques described in this paper are in the context of Java code, the basic concepts are also applicable to privileged-code placement issues in non-Java systems, including CLR.

1.4 Organization of This Paper

Section 2 introduces the control- and data-flow frameworks on which the MARCO tool is based. Section 3 describes an *access-rights analysis algorithm* for computing authorization requirements of Java code. Section 4 shows how the access-rights analysis algorithm can be enhanced to compute modifiable-component code locations that are closest to the authorization checks. The *privileged-code placement algorithm* described in Section 4 minimizes the chances of introducing unnecessary or redundant privileged code. Additionally, if unnecessary or redundant privileged code is already present, the algorithm will detect it. Section 5 presents *a tainted-variable analysis algorithm* for detecting potential misuses of tainted variables in code that is already privileged, or that is a candidate for becoming privileged as a result of executing the privileged-code placement algorithm. Section 6 presents our experience with running the MARCO tool on complex commercial-quality code. Section 7 describes previous results in the area of authorization, privileged-code, and tainted-variable analysis, and explains why the work presented in this paper is innovative with respect to those results. Finally, Section 8 summarizes the most important results presented in this paper.

2 Foundations of the Analysis Framework

The first analysis step is to construct an augmented, domain-specific invocation graph called an Access-Rights Invocation Graph (ARIG) [24]. An ARIG is a directed multi-graph $G = (N, E)$, where N is a set of nodes and E is a set of edges, with the following characteristics:

- Each node in the graph:
  - Represents a context-sensitive method invocation.
  - Is uniquely identified by its *calling context*:
    - The target method
    - The receiver and parameters values
  - Contains the following *state*:
    - The target method
    - For instance methods, an allocation site for the method's receiver
    - All parameters to the method, represented as a vector of sets of possible allocation sites
    - A set of possible return values from the target method, represented as a set of allocation sites
  - Is associated with a class loader name, corresponding to the name of the class loader that would load the method's declaring class at run time.
- Each labelled[2] edge $e = (m, n) \in E$ points from a call site in the method represented by node m to the target method represented by node n.
- An ARIG allows for bidirectional traversal.

[2] For simplicity, in this paper, we indicate the edges of an ARIG $G = (N, E)$ as pairs of nodes of the form (m, n), where $m, n \in N$. However, the edges of G are actually triplets of the form (m, n, w), where $m, n \in N$ and w is a call site in the method represented by m and pointing to the target method represented by n. In this sense, G is a multi-graph because there may exist multiple edges between any two nodes m and n, and those edges are distinguishable from each other based on the call-site information, which acts as a *label*. The call-site information contains the program counter at which w occurs.

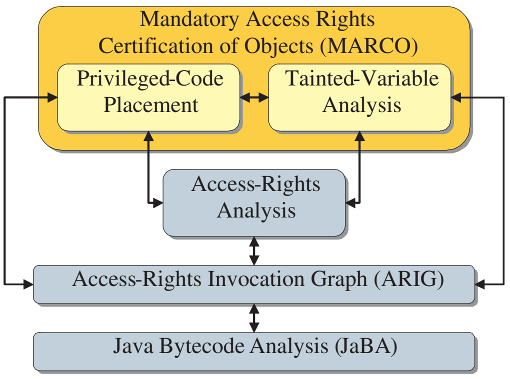

Mandatory Access Rights
Certification of Objects (MARCO)
Privileged-Code Placement
Tainted-Variable Analysis
Access-Rights Analysis
Access-Rights Invocation Graph (ARIG)
Java Bytecode Analysis (JaBA)

Fig. 4. Architecture of the Analysis Framework

An ARIG is used to execute an access-rights analysis. The results of the access-rights analysis and the ARIG itself are used by the MARCO tool to perform privileged-code placement and tainted-variable analysis. As Figure 4 shows, an ARIG is constructed using the Java Bytecode Analysis (JaBA) framework, which adopts a Control-Flow Analysis (CFA) [25] disambiguating between heap objects according to their allocation sites, with extra context for `Permission` objects. Specifically, JaBA is:

- *Path insensitive* [19] because it does not evaluate conditional statements and conservatively assumes that each conditional branch out of a conditional statement will be executed
- *Intraprocedurally flow sensitive* [31] because it considers the order of execution of the instructions within each basic block, accounting for local-variable kills [22] and casting of object references
- *Interprocedurally flow insensitive* [31] because it uses the conservative assumption that all instance and static fields are subject to modification at any time due to multi-threading
- *Context sensitive* [31] because its interprocedural analysis uniquely distinguishes each node by its calling context, with a context-sensitivity policy similar to Agesen's Cartesian Product Algorithm (CPA) [1]
- *Field sensitive* [31] because an object's fields are represented distinctly

An ARIG is domain-specific in that it is tailored to access-rights analysis, privileged-code placement, and tainted-variable analysis needs. Its domain-specific characteristics are described in the remainder of this section.

2.1 Modelling Multi-threading

In Java 2, when access to a restricted resource is attempted from within a child thread, all the code in the child thread and in all its ancestor threads must be granted the right to access that resource. This behavior can be modelled by identifying all the `run()` nodes in G whose receiver is a `Thread` object. For each of such nodes r, with receiver `t`, the node c representing the invocation of the `Thread` constructor that instantiated `t` in the parent thread is identified, and a new edge (c, r) is added to E. At the same time, the edge (s, r), where s represents the invocation of `start()` on `t`, is removed from E.

2.2 Extra Context for `Permission` Objects

The `Permission` parameter passed to `AccessController.checkPermission()` is frequently instantiated by the `SecurityManager`. For example, when the `SecurityManager`'s `checkWrite()` method is invoked, it instantiates a `FilePermission` and passes it to the `SecurityManager`'s `checkPermission()` method, which finally passes it to `AccessController.checkPermission()`. One problem is that different `FilePermission` objects instantiated through calls to `checkWrite()` in different parts of the program will all share the same type and allocation site. Therefore, JaBA would represent them as if they were the same object, yielding overly conservative results. The solution we adopted was to add extra context to `Permission` objects. Specifically, the context used to represent a `Permission` object `p` is not just the type and the allocation site of `p`, but also the node containing the allocation site of `p`. Therefore, if $m, n \in N$ are `checkWrite()` nodes in the ARIG such that the parameters for the method calls they represent are the `String`s `file1` and `file2`, respectively, the `FilePermission` allocated in m will be distinguished from the one allocated in n because $m \neq n$, even though both `FilePermission`s share the same type and allocation site. To avoid building an unnecessarily large invocation graph, this specialization is only applied to `Permission` objects allocated in the `SecurityManager`.

2.3 Propagation of String Constants

The constructor of a `Permission` object `p` takes zero or more `String` objects as parameters. As we shall see, the fully qualified `Permission` class name and the `String`s passed to the constructor of `p` uniquely identify the authorization requirement represented by `p`. For Java 2 authorization-related analyses, keeping track of string constants is, therefore, essential. For this reason, an ARIG includes propagation of string constants, unless these are dynamically generated.

2.4 Modelling of Callbacks

When building an invocation graph modelling the execution of a program, the method entry points can have parameters, which may include the receiver object, `this`. JaBA offers two options:

1. If the modifiable components being analyzed are part of a self-contained application, the analysis is typically treated as a *closed-world analysis*—one in which all the code executed at run time is also available during the analysis. In this case, JaBA uses Class Hierarchy Analysis (CHA) [11] to build the class hierarchy rooted at the parameter's declared type. When a callback from

that parameter object is encountered, JaBA models it by looking for all the possible implementations of the invoked method in the class hierarchy.
2. If the modifiable components under analysis are part of a library, the analysis is said to be an *open-world* or *incomplete-program analysis* [31] because the values and object sources of those parameters are part of the client application, which typically is only available at run time, unless the declared types of those parameters are final. If a callback from a parameter object of a non-final type occurs, JaBA, conservatively, does not model it because no control- and data-flow details on that callback are available at analysis time. However, JaBA records that a callback has been encountered. Potentially, each of such callbacks could require `AllPermission` at run time.

3 Access-Rights Analysis for Privileged Code

In this section, we present a simple data-flow analysis model for propagating access-rights and privileged-code requirements along an ARIG $G = (N, E)$. To compute the portions of modifiable-component code that should be made privileged, it is necessary to statically model the Java 2 authorization subsystem. Recall that, in Java 2, the run time enforces authorization by ultimately making a call to `checkPermission()` with a parameter `p` of type `Permission` representing the resource access being attempted. From what we said in Section 2.3, for authorization purposes, `p` can be characterized solely based on `p`'s *permission ID*, which consists of `p`'s fully-qualified class name and the `String` instances used to instantiate `p`.[3] For example, if `p` was instantiated with the statement

```
Permission p = new java.io.FilePermission("audit.txt", "write")
```

then `p`'s permission ID is `java.io.FilePermission "audit.txt", "write"`.

Let P be the universe of all the permission IDs associated with the code being analyzed. A function $\Pi : N \rightarrow 2^P$ can be defined that maps each node $n \in N$ to the set of permission IDs representing the `Permission` objects necessary to execute the method represented by n. Permission IDs represent authorization requirements. Determining privileged-code placement in a set of modifiable components involves propagating permission IDs across the ARIG representing the execution of those components.

3.1 Identification of `checkPermission()` Nodes

The first step of the algorithm is to iterate over all the nodes of G to identify those that correspond to `checkPermission()` method calls. For each of such nodes a, all the possible `Permission` allocation sites that have flowed to the

[3] In fact, in the JRE reference implementation, authorizations are granted to programs and principals by just listing the corresponding permission IDs in a flat-file policy database, called the *policy file* [29].

formal argument are identified,[4] and the permission IDs are computed from the corresponding constructor nodes. This phase requires $\mathcal{O}(|N|)$ time.

3.2 Reverse Propagation of Permission IDs

The Java 2 authorization subsystem mandates that, at the point where `checkPermission()` is invoked with a `Permission` parameter `p`, all the code on the execution thread's stack be granted the authorization represented by `p`. This can be modelled by identifying the node a corresponding to the `checkPermission()` call and propagating the permission ID corresponding to `p` backwards to all the predecessors of a, recursively. Thus, each node $n \in N$ is mapped to a (possibly empty) set of permission IDs, obtained as the union of the permission ID sets propagated from n's successors as follows:

$$\Pi(n) = \bigcup_{m \in Succ(n)} \Pi(m)$$

where $Succ(n) = \{m \in N | (n, m) \in E\}$. When for some $n \in N$, $\Pi(n)$ changes as a result of this propagation, $\Pi(m)$ is unioned with $\Pi(n)$ for all $m \in Pred(n)$, where $Pred(n) = \{m \in N | (m, n) \in E\}$.

The reverse propagation of permission ID sets just described can be formalized in terms of data flow. Using a standard data-flow notation [22, 2, 25], we define *data-flow sets* $GEN(n)$ and $KILL(n)$ for each node $n \in N$ as follows:

- $GEN(n)$ contains the permission IDs *generated* by node n. Such permission IDs correspond to the authorizations checked at the method represented by node n. For the Java 2 access-control model, $GEN(a) \neq \varnothing$ if and only if a is a `checkPermission()` node. In particular, for any such node a, $GEN(a)$ contains exactly the permission IDs corresponding to the authorizations checked at the method represented by a in the ARIG.
- $KILL(n)$ contains the permission IDs *killed* by node n. Such permission IDs correspond to authorization requirements whose propagations on the call stack stop at the predecessors of node n. According to the Java 2 access-control model, if $d \in N$ represents a `doPrivileged()` method invocation, $KILL(d)$ is the universe P of all the permission IDs defined in the ARIG. This is because, in Java 2, a call to `doPrivileged()` does not extend authorizations to client code selectively, in a fine-grained fashion, but does it in a coarse-grained fashion.[5] For any other node $n \in N$, $KILL(n) = \varnothing$.

[4] Even though `checkPermission()` takes only one `Permission` parameter, that parameter may correspond to more than one object in the ARIG model, since JaBA is path insensitive and interprocedurally flow insensitive.

[5] Unlike the Java 2 `doPrivileged()` method, the CLR `Assert()` method shields client code from authorization requirements in a fine-grained fashion [7]. Library code can assert a specific `IPermission` object, and only the authorization represented by that object will be implicitly granted to the client code currently on the stack. To model this behavior correctly, the $KILL$ set of the asserting method's node would only have to contain the permission IDs of the asserted `IPermission` objects.

It is therefore convenient to introduce a function $NodeType : N \to \{check, grant, other\}$. For each $n \in N$, $NodeType(n)$ is defined as follows:

$$NodeType(n) = \begin{cases} check, \text{ if } n \text{ is a } \texttt{checkPermission()} \text{ node;} \\ grant, \text{ if } n \text{ is a } \texttt{doPrivileged()} \text{ node;} \\ other, \text{ otherwise.} \end{cases}$$

The *check* nodes are those representing `checkPermission()` method calls, which trigger authorization checks, while the *grant* nodes are those representing calls to `doPrivileged()`, through which the callers on the thread stack are implicitly granted authorizations. The *other* nodes do not affect the data flow. The following pseudo-code formalizes the assignment of the data-flow sets:

```
1:  for each node $n$ {
2:     switch($NodeType(n)$) {
3:        case $check$ :
4:           $GEN(n) = \{p \in P | p \text{ is checked at } n\}$
5:           $KILL(n) = \varnothing$
6:        case $grant$ :
7:           $GEN(n) = \varnothing$
8:           $KILL(n) = P$
9:        case $other$ :
10:          $GEN(n) = \varnothing$
11:          $KILL(n) = \varnothing$
12:    }
13: }
```

The data-flow equations for each node $n \in N$ are defined in the usual way as follows:

$$\begin{aligned} OUT(n) &= (IN(n) \cup GEN(n)) - KILL(n) \\ IN(n) &= \bigcup_{m \in Succ(n)} OUT(n) \end{aligned}$$

where $OUT(n)$ and $IN(n)$ are the sets of permission IDs propagated from n and reaching n, respectively. The data-flow analysis just described converges to a fixed point in $\mathcal{O}(|E||P|)$ time since $(2^P, \subseteq)$ is a finite lattice and the *data-flow functions* $OUT, IN : N \to 2^P$ are monotonic with respect to the lattice's partial order, $\subseteq$ [18].

3.3 Permission ID Propagation from `doPrivileged()` Nodes

According to the Java 2 authorization subsystem, authorization requirements propagated upwards via a `doPriviledged()` node must not propagate beyond the predecessors of the `doPrivileged()` node. This can be modelled as follows: When a `doPrivileged()` node d is encountered during the reverse propagation of permission IDs described in Section 3.2, its permission ID set, $\Pi(d)$, is propagated to d's predecessors only after the propagation algorithm for all the other nodes has terminated. The propagation of $\Pi(d)$ upwards must not be performed recursively. If n is a node in $Pred(d)$ and $\Pi(n)$ changes as a result of the propagation of $\Pi(d)$ from d, $\Pi(n)$ is not transmitted to the nodes in $Pred(n)$. One data-flow equation is sufficient to describe this one-step propagation:

$$IN(n) = IN(n) \cup \bigcup_{\substack{d \in Succ(n) \\ NodeType(d) = grant}} IN(d)$$

This equation has an effect only for those nodes that have a *grant* node as a successor. The time complexity of this one-step propagation is $\mathcal{O}(|E|)$.

3.4 Complexity

The access-rights analysis converges in $\mathcal{O}(|E||P|)$ time. When the analysis terminates, for each node $n \in N$, the data-flow set $IN(n)$ will be equal to the set $\Pi(n)$ and will represent the authorizations required to execute the method represented by n with n's calling context.

4 Privileged-Code Placement

This section describes how the propagation algorithm described in Section 3 can be augmented to automatically detect which portions of modifiable-component code should be made privileged while minimizing the risks of violating the Principle of Least Privilege, with a precision that goes to the level of the program counter within a method. For each privileged-code location it recommends, the algorithm provides an explanation. Additionally, this section shows how to compute the authorizations that privileged code will implicitly grant to client code. Finally, this section describes how the algorithm detects existing unnecessary or redundant privileged code, and avoids inserting new privileged code that is unnecessary or redundant.

4.1 Insertion of `doPrivileged()` Calls

In Java 2, class loaders are organized as a tree T, called the *class-loading delegation tree* [28]. JaBA models the class-loading system and associates a class loader name with every node in an ARIG. The privileged-code placement process is configured by assuming that all the classes in the modifiable components will be loaded by a designated class loader, called the *component loader*. A *boundary edge* in G is any edge $e = (m, n) \in E$ such that m is associated with the component loader and n is associated with a different class loader in T. If $\Pi(n) \neq \varnothing$, then the call represented by e is guaranteed to lead to the Java 2 authorization subsystem. Such a call is a candidate for becoming privileged. For example, e may be the edge resulting from calling the constructor of `FileOutputStream`

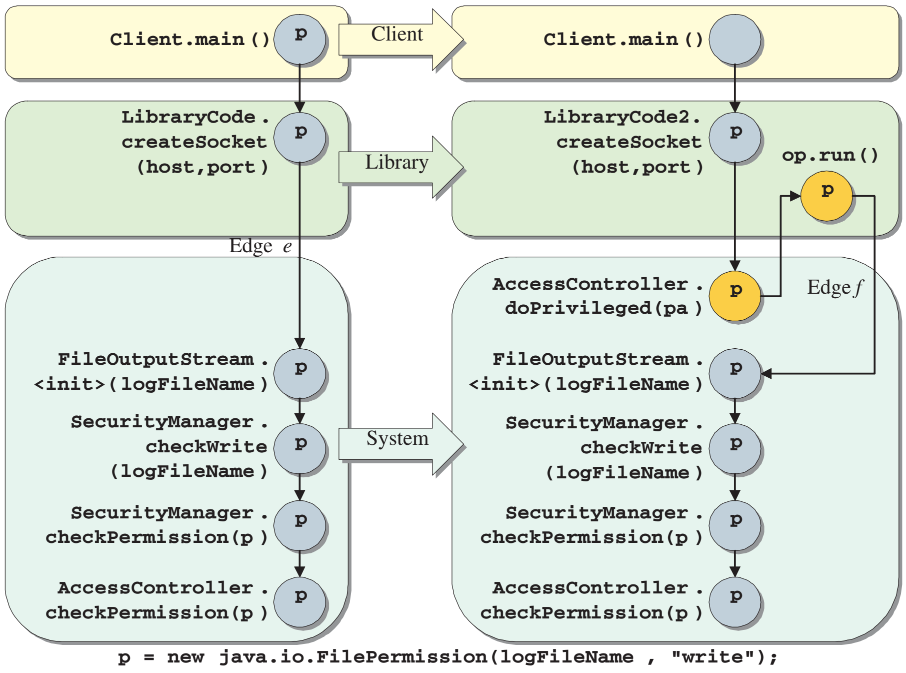


Fig. 5. Changes in the ARIG after Making Library Code Privileged

from method `createSocket()` in Figure 1. Figure 2 shows how to wrap the `FileOutputStream` constructor call into a privileged block, and Figure 5 shows the corresponding ARIGs. Notice how the ARIG on the right in Figure 5 reflects the presence of `doPrivileged()` by not propagating the `FilePermission` requirement beyond the invocation of `createSocket()`, exempting client code from the `FilePermission` requirement, as desired.

The algorithm described in Section 3 can be augmented to identify any boundary edge $e = (m, n)$ such that $\Pi(n) \neq \varnothing$. The information contained in e and $\Pi(n)$ is sufficient to determine the exact portion of modifiable-component code that is a candidate for becoming privileged along with an explanation, and to identify the authorizations that the privileged code will implicitly grant to client code:

1. The class name, method signature, and program counter that constitute a possible `doPrivileged()` location can be obtained from node m and the call site in e. Since e is a boundary edge, from a control-flow perspective the location computed by this algorithm is, in the modifiable-component code, the closest to the authorization check. This ensures that only the portion of modifiable-component code effectively leading to an authorization check will be made privileged, which minimizes the risks of violating the Principle of Least Privilege.

2. The authorizations implicitly granted to clients if a call to `doPrivileged()` is inserted are represented by the permission IDs in $\Pi(n)$.
3. As an explanation, the fully qualified signature of the method being invoked at node n and causing the authorization requirements in $\Pi(n)$ can be obtained from node n itself. If a more detailed explanation is desired, all the paths from n to the `checkPermission()` nodes in the ARIG subgraph rooted at n can be reported. Such paths are those through which the authorization requirements in $\Pi(n)$ have propagated up to n.

The privileged-code placement algorithm can be customized. Instead of recommending the privileged-code locations that, in the modifiable components, are the closest to the authorization checks, the algorithm could, for example, identify the privileged-code locations closest to those components' entry points. With this approach, however, code not requiring authorizations may become unnecessarily privileged.

4.2 Detecting Unnecessary or Redundant Privileged Code

An unnecessary `doPrivileged()` call may result from changes made to modifiable-component code during code development or maintenance. A call to `doPrivileged()` that was originally considered necessary no longer triggers an authorization check after the change. A redundant `doPrivileged()` call may result from poor code design or from integrating different components so that a call to `doPrivileged()` that was once considered necessary because it led to an authorization check becomes redundant because other `doPrivileged()` calls now dominate the authorization check. As we observed in Section 1, unnecessary or redundant privileged code should be made unprivileged for security and performance reasons. The algorithm described in Section 3 can be augmented to identify unnecessary or redundant calls to `doPrivileged()` by simply detecting any `doPrivileged()` node d in the graph such that $\Pi(d) = \varnothing$.

If a code instruction does not require authorizations, it is a poor security practice to make it privileged [35]. The following instruction in the `run()` method of Figure 3 has been made privileged even though it does not access a restricted resource:

```
System.out.println("Received request from user " + userName);
```

Such an instruction should be made unprivileged even though, in this case, $\Pi(d) \neq \varnothing$. The privileged-code placement algorithm can easily detect unnecessarily privileged instructions. Let d be a `doPrivileged()` node and r its `PrivilegedAction` or `PrivilegedExceptionAction` `run()` successor. If $\Pi(r) \neq \varnothing$ and there exists $n \in Succ(r)$ such that $\Pi(n) = \varnothing$, then the method invocation represented by n should be made unprivileged.

4.3 Avoiding Unnecessary or Redundant Privileged Code

The permission ID set $\Pi(n)$ associated with the head node n of a boundary edge $e = (m, n)$ must be non-empty for the privileged-code placement algorithm to recommend a call to `doPrivileged()`. Therefore, except for those cases in which the access-rights analysis conservatively reports unrealizable authorization requirements, none of the `doPrivileged()` calls recommended by the privileged-code placement algorithm are unnecessary or redundant. Furthermore, since the privileged-code placement algorithm precisely identifies the method invocations that should be made privileged, no code instruction will become unnecessarily or redundantly privileged as a result of executing the algorithm, except, again, for the cases of conservativeness.

However, it should be observed that, if the analysis is performed after a call to `doPrivileged()` has been inserted, any edge from the `PrivilegedAction` or `PrivilegedExceptionAction`'s `run()` node into a different class loader's name space will also be, by definition, a boundary edge. For example, in Figure 5, after the `FileOutputStream` constructor has been wrapped in a privileged block, the edge f from the `op.run()` node to the `FileOutputStream.<init>()` node is a boundary edge, and the permission ID set associated with the head node is non-empty. Reporting a privileged-code requirement would make the existing call to `doPrivileged()` redundant. To avoid this situation, any boundary edge originating from a `PrivilegedAction` or `PrivilegedExceptionAction`'s `run()` node such that the only predecessors of the `run()` node are `doPrivileged()` nodes is automatically excluded a candidate for `doPrivileged()`.

4.4 Complexity

The privileged-code placement algorithm is obtained by augmenting the access-rights analysis algorithm in a way that does not affect the algorithm's complexity and convergence except for a constant factor. Therefore, executing the privileged-code placement algorithm still requires $\mathcal{O}(|E||P|)$ time.

5 Tainted-Variable Analysis

We refer to the data that either originate from an untrusted source or that can be derived from an untrusted source as being *tainted* [27]. Tainted data and the variables that hold or reference it can be used for certain kinds of overwrite attacks [27], such as overwriting the name of a file or jump address. Sometimes, however, it may be necessary to use a tainted variable when accessing restricted resources. In such cases, the data can be *sanitized* and made untainted by performing sanity checks on it before using it in restricted operations [3]. Sanity checks are usually domain or component specific. We assume that, for a specific application, there is an associated library containing the sanity checks on that application's tainted variables.

A tainted variable is not necessarily a security problem. It may constitute a security problem if it is also a *privileged* variable, meaning that it is used inside privileged code [35]. Even a privileged tainted variable is not necessarily a security problem. In fact, we distinguish two types of privileged tainted variables: if a privileged tainted variable is used to access a restricted resource, we will call it *malicious*, otherwise we will call it *benign*. Since authorization checks are not performed beyond the stack frame invoking `doPrivileged()`, an untrusted client application could exploit a malicious variable to have the privileged code access restricted resources on its behalf.

In Figure 2, variable `logFileName` is not tainted because its value cannot be set by a client application. In Figure 3, the `host` and `port` parameters are both tainted because their values can be set by any client application and no sanity check is performed on them. Additionally, they are both privileged because they are used inside privileged code, and they are malicious because they are are used to access a restricted resource. An untrusted client, with no `SocketPermission`, can invoke `getSocket()` on the trusted library and have the library open an arbitrary socket connection on its behalf. Variable `userName`, though tainted and privileged, is benign because it is not used to perform any restricted operation.

This section presents a simple interprocedural tainted-variable analysis algorithm that augments the privileged-code placement algorithm described in Section 4. The objective of the tainted-variable analysis is both to detect existing malicious variables and to avoid the introduction of new malicious variables when making new code privileged.

The first step of the tainted-variable analysis algorithm is to compute the initial set S of tainted variables, which is the union of the following two sets:

– Set S_1, containing the modifiable-component instance and static fields that can be modified by client code
– Set S_2, containing all the parameters to the modifiable components' public and protected entry methods, including the receiver objects for non-static methods

Set S_2 can be computed easily. However, it should be observed that if a package in a modifiable component is not sealed [29], then S_2 should contain the parameters to the package's default-scope methods as well.

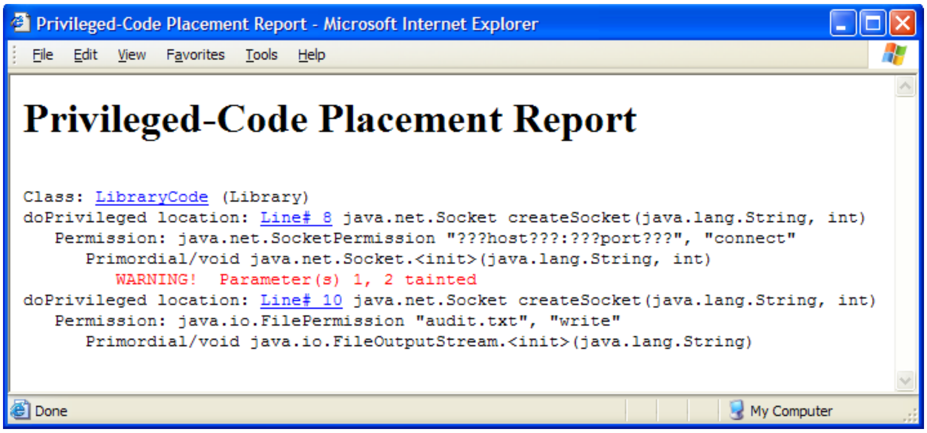
A tainted variable can, directly or indirectly, taint other variables, for example through assignments. The second step of the tainted-variable analysis algorithm consists of identifying existing privileged variables in the modifiable components and detect if they are tainted. Any standard interprocedural program-slicing analysis algorithm [36] can be used to detect value flows from tainted variables to privileged variables. Our algorithm uses a program-slicing algorithm that, for any privileged variable `x`, constructs a slice of `x` and then checks if the slice contains variables in set S. If so, `x` is potentially tainted as well. It remains to be seen whether `x` is benign or malicious. The access-rights analysis algorithm gives us the answer. Since `x` is privileged, `x` must be used in at least one privileged instruction. For `x` to be benign, it must be $\Pi(n) = \varnothing$ for any node n representing a method executing any privileged instruction that contains `x`. If there exists a node n representing a method in which a privileged instruction containing `x` is executed, such that $\Pi(n) \neq \varnothing$, then we conservatively assume that `x` is malicious, in which case we look for a sanity check on `x`. If a sanity check for `x` exists and it can be determined that `x` passes it, then `x` is considered sanitized. Otherwise, the tainted-variable analysis reports a potential security risk.

If x is not an existing privileged variable, but it would become so by making new code privileged as a result of executing the privileged-code placement algorithm, the tainted-variable analysis proceeds exactly as before. The only difference is that it will report that the code containing x can be made privileged only if x is benign or sanitizable.

6 Experimental Results

Context- and intraprocedurally flow-sensitive static analysis has a reputation for requiring significant processing power and memory. We have performed privileged-code and tainted-variable analysis on parts of `rt.jar`, large commercial middleware, and the Standard Performance Evaluation Corporation Java Business Benchmark 2000 (SPECjbb2000) program [34]. The simplest library that we analyzed is the `LibraryCode` class in Figure 1. Figure 6 shows the HyperText Markup Language (HTML) output produced by the MARCO tool. For better usability, the HTML output has links to the source code anchored to the line numbers where a call to `doPrivileged()` should be inserted. As expected, the tool reported two possible privileged-code placements, but detected that both the parameters to the `Socket` constructor were tainted. This was an indication that only the call to the `FileOutputStream` constructor could be safely made privileged. The results reported by the MARCO tool on all the other benchmarks were also correct based on source-code manual inspection and subsequent testing. In particular, the tool detected when existing privileged code was unnecessary or redundant, and appropriately distinguished between malicious and benign tainted variables.

Most recently, we analyzed Eclipse V3.0 [13] to identify which portions of the plug-in code should be made privileged in order to enable Eclipse to run


Privileged-Code Placement Report - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Done
My Computer

Privileged-Code Placement Report

```
Class: LibraryCode (Library)
doPrivileged location: Line# 8 java.net.Socket createSocket(java.lang.String, int)
   Permission: java.net.SocketPermission "???host???:???port???", "connect"
      Primordial/void java.net.Socket.<init>(java.lang.String, int)
         WARNING!  Parameter(s) 1, 2 tainted
doPrivileged location: Line# 10 java.net.Socket createSocket(java.lang.String, int)
   Permission: java.io.FilePermission "audit.txt", "write"
      Primordial/void java.io.FileOutputStream.<init>(java.lang.String)
```

Fig. 6. Privileged-Code Placement Report on `LibraryCode`

Table 1. Analysis Results

Eclipse Plug-in	*Classes*	*Methods*	*Time (sec)*	*Nodes*	*Edges*	*Instr. (bytes)*	*doPriv.*
`ant`	2245	14799	4668	169539	1833305	818342	1908
`core.runtime`	1265	7233	1379	59771	134191	421094	353
`osgi`	1069	6091	814	62031	141397	362482	256
`tomcat`	2804	19885	5793	197709	471957	1066369	2011
`ui`	2910	16299	11254	191752	1843518	891270	150
`ui.forms`	972	4497	4430	29199	59605	286739	14

with a Java 2 `SecurityManager` enabled. The results reported in Table 1 are from running the MARCO tool on an IBM Personal Computer with an Intel 1.6 GHz Pentium M processor and 1 GB of Random Access Memory (RAM), and with operating system Microsoft Windows XP SP2. The MARCO tool has been implemented in Java. We ran it both as a stand-alone application and inside Eclipse V3.0 using Sun Microsystems' JRE V1.4.2_02. JRE functionality was made part of the analysis scope by including the JRE V1.4.2_02 system and extension libraries. To reduce the size of the analysis scope, the MARCO tool was customized to build the analysis scope based on the plug-in dependencies. The number of classes in the analysis scope was still greater than 20,000. Table 1 shows, for some Eclipse plug-ins, how many classes and methods were included in the ARIG, the time employed to run the whole analysis (including the ARIG construction, which on average takes 96% of the total time), the number of nodes and edges in the ARIG, the instructions count, and the number of `doPrivileged()` locations suggested by the tool. On average, 50% of the code portions candidate to become privileged contained malicious tainted variables—an indication that `doPrivileged()` should not be used. The `osgi` plug-in was the only one that already contained calls to `doPrivileged()`. The total number was 29, and 8 of those were unnecessary or redundant.

Other analyses were performed on large commercial products (over 20,000 classes in the analysis scope), based on the JRE V1.1 access-control model. The goal was to identify their privileged-code requirements to allow them to successfully run with the Java 2 access-control model enabled.

7 Related Work

Privileged code has historic roots in the 1970's. The Digital Equipment Corporation (DEC) Virtual Address Extension/Virtual Memory System (VAX/VMS) operating system had a feature similar to the `doPrivileged()` method in Java 2 and the `Assert()` method in CLR. The VAX/VMS feature was called *privileged images*. Privileged images were similar to UNIX `setuid` programs, except that privileged images ran in the same process as all the user's other unprivileged programs. This meant that they were considerably easier to attack than UNIX `setuid` programs because they lacked the usual separate process/separate address space protections. One example of an attack on privileged images is demonstrated in a paper by Koegel, Koegel, Li, and Miruke [23].[6]

More recently, static and dynamic analysis techniques have both been used for modelling authorization algorithms. Much of the work has focused on performance optimizations or on providing alternatives to the existing approaches employed by Java 2 [29, 28] and CLR [16]. Pottier, Skalka, and Smith [30] extend and formalize Wallach's security passing style [41] via type theory using a λ-calculus, called λ_{sec}. However, their approach does not model all of Java's authorization characteristics, including multi-threaded code and analysis of incomplete programs [31], nor does it compute the authorization object, which often includes identifying the `String` parameters to the `Permission` object's constructor. Bartoletti, Degano, and Ferrari [5] are interested in optimizing performance of run-time authorization testing. This is done by eliminating redundant tests and relocating others as is needed. Additionally [6], they investigate ways in which program transformations can preserve security properties in existing code, particularly in the context of Java. Specifically, the transformations they study include redundant authorization tests elimination, dead code elimination, method inlining, and an eager evaluation strategy for stack inspection. While their model takes privileged code into account, they assume that privileged code has already been inserted, and do not solve the problem of detecting which portions of library code should be made privileged. Banerjee and Naumann [4] apply denotational semantics to show the equivalence of *eager* and *lazy* semantics for stack inspection, provide a static analysis of *safety*, and identify transformations that can remove unnecessary authorization tests. Significant limitations to this approach are that the analyses are limited to a single thread, and incomplete-program analyses are not supported. Jensen, Le Métayer, and Thorn [20] focus on proving that code is secure with respect to a global security policy. Their model adopts operational semantics to prove the properties, using a two-level temporal logic, and shows how to detect redundant authorization tests. They assume all of the code is available for analysis, and that a call graph can be constructed for the code. Felten, Wallach, Dean, and Balfanz have studied a number of security problems related to mobile code [39, 12, 41, 8, 40, 10, 9]. In particular, they present a formalization of stack introspection, which examines authorization based on the principals currently active in a thread stack at run time (*security state*). An authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [41]. Each method is modified so that it passes a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. By running the application and encoding the security state, the security passing

[6] In a private communication with Dr. Paul A. Karger [21], he indicated that privileged images had been a very significant source of security attacks in the VAX/VMS operating system, and required many patches and updates over the years. He did extensive work on resolving those problems at DEC in the 1979-1980 timeframe.

style explores subgraphs of the comparable invocation graph, and discovers the associated security states and authorizations. The purpose of their work is to optimize the authorization performance, while ours is to discover which portions of library code should be made privileged. Our approach analyzes all the possible execution paths, even those that may not be discovered by a limited number of test cases. Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [14] describe a system that inlines reference monitors into the code to enforce specific security policies. Their objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. Koved, Pistoia, and Kershenbaum [24] describe an algorithm and system for computing Java 2 security authorization requirements for existing Java code. Their algorithm, which is the starting point for this paper, covers many of the subtle aspects of Java 2 security, including authorization requirements for multi-threaded applications and analysis of incomplete programs [31], for the computation of an ARIG.

The notion of tainted variables became known with the Perl language. In Perl, using the `-T` option allows detecting tainted variables [38]. Shankar, Talwar, Foster, and Wagner present a tainted-variable analysis for CQual using constraint graphs [33]. To find format string bugs, CQual uses a type-qualifier system [15] with two qualifiers: *tainted* and *untainted*. The types of values that can be controlled by an untrusted adversary are qualified as being tainted, and the rest of the variables are qualified as untainted. A constraint graph is constructed for a CQual program. If there is a path from a tainted node to an untainted node in the graph, an error is flagged. Newsome and Song [27] propose a dynamic tainted-variable analysis that catches errors by monitoring tainted variables at run time. Data originating or arithmetically derived from untrusted sources, such as the network, are marked as tainted. Tainted variables are tracked at run time, and when they are used in a dangerous way an attack is detected. Volpano, Irvine, and Smith [37] relate tainted-variable analysis to enforcing information flow policies through typing. Ashcraft and Engler [3] also use tainted-variable analysis to detect software attacks due to tainted variables. Their approach provides user-defined sanity checks to untaint potentially tainted variables. In Java 2, Enterprise Edition (J2EE), access rights are defined in terms of operations on components, instead of the data encapsulated and used by the components. Naumovich and Centonze [26] address the need for specifying access rights on data. Access right support for data can simplify sanity checks for tainted variables. For instance, a tainted variable is benign for those clients who have access rights over the data referenced by the tainted variable.

8 Conclusion

In this paper, we presented an interprocedural analysis for safely adding privileged code in order to ensure that no unnecessary access rights are granted to client code, and that tainted variables are not exploited. Our approach for privileged-code and tainted-variable analysis is built on top of an access-rights analysis and uses an Access-Rights Invocation Graph (ARIG). As part of the analysis, we solve a number of other related problems, including identification of unnecessary and redundant privileged code and flagging when tainted variables are benign or malicious. We have implemented the analysis described in this paper and are currently using it to identify security violations due to privileged code in large libraries and applications. Our analysis technique scales well enough to produce usable results on large applications and libraries. While the analysis techniques described in this paper are in the context of Java code, the basic concepts are applicable to privileged-code placement and tainted-variable analysis issues in non-Java-based systems as well.

Acknowledgments

The authors would like to thank their colleagues at the IBM T. J. Watson Research Center: Julian Dolby, Ted Habeck, Paul Karger, Aaron Kershenbaum, Michael Steiner, Sam Weber, and Xiaolan Zhang for their invaluable technical contributions to this paper. Thanks also to the ECOOP 2005 reviewers for their insightful comments and suggestions, which helped to improve this paper.

References

1. Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2–26. Springer-Verlag, August 1995.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, January 1986.
3. Ken Ashcraft and Dawson Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 143–159, Oakland, CA, USA, May 2002. IEEE Computer Society.
4. Anindya Banerjee and David A. Naumann. A Simple Semantics and Static Analysis for Java Security. Technical Report CS2001-1, Stevens Institute of Technology, Hoboken, NJ, USA, July 2001.
5. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Static Analysis for Stack Inspection. In *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science*, volume 54, Amsterdam, The Netherlands, 2001. Elsevier.
6. Massimo Bartoletti, Pierpaolo Degano, and Gian Luigi Ferrari. Stack Inspection and Secure Program Transformations. *International Journal of Information Security*, 2(3):187–217, August 2004.
7. Frédéric Besson, Tomasz Blanc, Cédric Fournet, and Andrew D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pages 61–75, Pacific Grove, CA, USA, June 2004. IEEE Computer Society.
8. Drew Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, 1997. ACM Press.

9. Drew Dean, Edward W. Felten, and Dan S. Wallach. Java Security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Silver Spring, MD, USA, 1996. IEEE Computer Society Press.
10. Drew Dean, Edward W. Felten, Dan S. Wallach, and Dirk Balfanz. Java Security: Web Browsers and Beyond. Technical Report 566-597, Princeton University, Princeton, NJ, USA, February 1997.
11. Jeffrey Dean, David Grove, and Craig Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 77–101, Aarhus, Denmark, August 1995. Springer-Verlag.
12. Richard Drews Dean. *Formal Aspects of Mobile Code Security.* PhD thesis, Princeton University, Princeton, NJ, USA, January 1999.
13. Eclipse Project, `http://www.eclipse.org`.
14. Úlfar Erlingsson and Fred B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, CA, USA, May 2000. IEEE Computer Society.
15. Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, Berlin, Germany, June 2002.
16. Adam Freeman and Allen Jones. *Programming .NET Security.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.
17. Li Gong, Gary Ellison, and Mary Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, Reading, MA, USA, second edition, May 2003.
18. George Grätzer. *General Lattice Theory.* Birkhäuser, Boston, MA, USA, second edition, January 2003.
19. Sumit Gulwani and George C. Necula. Path-sensitive Analysis for Linear Arithmetic and Uninterpreted Functions. In *11th Static Analysis Symposium*, volume 3148 of *LNCS*, pages 328–343. Springer-Verlag, August 2004.
20. Thomas P. Jensen, Daniel Le Métayer, and Tommy Thorn. Verification of Control Flow Based Security Properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103, Oakland, CA, USA, May 1999.
21. Paul A. Karger, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, USA. Private communication, 17 December 2004.
22. Gary A. Kildall. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 194–206, Boston, MA, USA, 1973. ACM Press.
23. John F. Koegel, Rhonda M. Koegel, Zhiming Li, and Dattaram T. Miruke. A Security Analysis of VAX VMS. In *ACM '85: Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, pages 381–386. ACM Press, 1985.
24. Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.
25. Steven S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, June 1997.
26. Gleb Naumovich and Paolina Centonze. Static Analysis of Role-Based Access Control in J2EE Applications. *SIGSOFT Software Engineering Notes*, 29(5):1–10, September 2004.

27. James Newsome and Dawn Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium*, San Diego, CA, USA, February 2005. IEEE Computer Society.
28. Marco Pistoia, Nataraj Nagaratnam, Larry Koved, and Anthony Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.
29. Marco Pistoia, Duane Reller, Deepak Gupta, Milind Nagnur, and Ashok K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.
30. François Pottier, Christian Skalka, and Scott F. Smith. A Systematic Approach to Static Access Control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.
31. Barbara G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.
32. Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, September 1975.
33. Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, USA, August 2001.
34. Standard Performance Evaluation Corporation Java Business Benchmark 2000 (SPECjbb2000), `http://www.spec.org`.
35. Sun Microsystems, Security Code Guidelines, `http://java.sun.com`.
36. Frank Tip and T. B. Dinesh. A Slicing-based Approach for Locating Type Errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, 2001.
37. Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.
38. Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, third edition, July 2000.
39. Dan S. Wallach. *A New Approach to Mobile-Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, January 1999.
40. Dan S. Wallach, Dirk Balfanz, Drew Dean, and Edward W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, 1997. ACM Press.
41. Dan S. Wallach and Edward W. Felten. Understanding Java Stack Inspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.