Access Rights Analysis for Java

Larry Koved
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598
koved@us.ibm.com

Marco Pistoia
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598
pistoia@us.ibm.com

Aaron Kershenbaum
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, New York 10598
aaronk@us.ibm.com

ABSTRACT

Java™ 2 has a security architecture that protects systems from unauthorized access by mobile or statically configured code. The problem is in manually determining the set of security access rights required to execute a library or application. The commonly used strategy is to execute the code, note authorization failures, allocate additional access rights, and test again. This process iterates until the code successfully runs for the test cases in hand. Test cases usually do not cover all paths through the code, so failures can occur in deployed systems. Conversely, a broad set of access rights is allocated to the code to prevent authorization failures from occurring. However, this often leads to a violation of the "Principle of Least Privilege."

This paper presents a technique for computing the access rights requirements by using a context sensitive, flow sensitive, interprocedural data flow analysis. By using this analysis, we compute at each program point the set of access rights required by the code. We model features such as multi-threading, implicitly defined security policies, the semantics of the Permission.implies method and generation of a security policy description. We implemented the algorithms and present the results of our analysis on a set of programs. While the analysis techniques described in this paper are in the context of Java code, the basic techniques are applicable to access rights analysis issues in non-Java-based systems.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Computer–aided software engineering (CASE).

D.2.4 [**Software/Program Verification**]: Format methods, Validation.

D.2.5 [**Testing and Debugging**]: Code inspections and walk-throughs, Diagnostics, Testing tools (e.g., data generators, coverage testing), Tracing.

General Terms

Security, Languages.

Keywords

Security, call graph, invocation graph, data flow analysis, Java security, access rights.

1. Introduction

Java™ 2 has a security architecture intended to protect client and server systems from dynamically installed (e.g., mobile code) or statically configured malicious code [13] [14] [15] [23]. Applet code is downloadable from the Internet into a Web browser [23] [19], and uploadable via RMI [22] into a server application. The Java 2 security system contains an authentication subsystem and an authorization subsystem. This paper focuses on the authorization subsystem, automating the determination of access rights needed to execute the code.

Prior to deploying application or library code in Java, a critical question arises: "What Java access rights are required to allow the code to execute?" In practice this problem is solved empirically. The developer reads documentation for libraries used (including the Java run-time libraries) and deduces the required access rights. Unfortunately, this documentation is often missing, misleading, or out of date. In the absence of reliable documentation, the developer executes the new code and observes authorization failures. The developer then grants additional access rights and retests. The developer repeats this process, possibly many times, until there are no authorization failures. However, required access rights requirements can remain undiscovered due to an insufficient number of test cases, rendering the code unstable.

An analogous situation arises in systems where mobile code is dynamically installed and the system administrator (e.g., the Web browser user) must determine the set of access rights to provide. The system administrator usually relies on the code developers'/distributors' recommendations, with the risk that too many access rights are granted and security holes are created. Alternatively, the system administrator runs the code with a smaller set of access rights, examines failures, and incrementally adds access rights as is necessary. This process is tedious and error-prone. As before, insufficient testing results in improper authorizations, creating security exposures or application instability.

This paper describes a technique based on context sensitive data and control flow analyses to automatically determine access rights required by Java programs or libraries. We use a modified interprocedural invocation graph, called an access rights invocation graph (ARIG), to compute the access rights.

In Java, access rights are modeled using the Permission class hierarchy. The root of the class hierarchy is the abstract class java.security.Permission. By default, all Permissions are "approval" of access rather than "denial." A Permission object is an instance of a subclass of java.security.Permission. For instance,

```
perm = new FilePermission("/tmp/abc", "read");
```

creates a Permission object to read the file /tmp/abc. In our analysis, we compute the set of Permission objects to associate with each program point by constructing an ARIG to propagate the access rights. An ARIG consists of nodes corresponding to AccessController's checkPermission and doPrivileged methods, which are the boundary nodes, as well as all nodes in the invocation graph in all paths between the boundary nodes and from the boundary nodes to the root nodes.

To summarize, the main contributions of this paper are:

- We present a context sensitive, flow sensitive analysis for computing the access rights requirements of a program.
- We model features such as multithreading, implicitly defined security policies, semantics of the Permission.implies method, and the generation of a security policy description.
- We use a modified invocation graph, an ARIG, to propagate access rights.
- We implemented the algorithms and present test results from the use of our tool.

·Our analysis technique is scalable enough to produce usable results on problems with an analysis scope of over 20,000 classes.

2. Prior Work

Both static and dynamic analysis techniques are employed in modeling security and authorization. Much of this work has been applied to eliminate or minimize redundant authorization tests or define alternatives to the current approach to defining authorization points within code.

Pottier, Skalka and Smith [25] extend and formalize Wallach's security passing style [31] via type theory using a λ-calculus, called λ_{sec}. Pottier, et al, were unable to model all of Java's authorization characteristics, including multithreaded code and "open world" analysis. Nor does it consider computing the authorization object, which often includes identifying the String parameters to the Permission objects' constructor. All of these strongly affect the completeness of an authorization analysis.

Jensen, Métayer and Thorn [17] focus on proving that code is secure with respect to a global security policy. Their model uses 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, though they do not do so themselves. The results are also limited by an assumption that security properties can be expressed solely in terms of the control flow and call graph of the program. For Java, essential authorization information is based on values (usually string constants) propagated to authorization tests.

Bartoletti, Degano and Ferrari [5] are interested in optimizing performance of run-time authorization testing, by eliminating redundant tests and relocating others as is needed. The reported results apply operational semantics to model the run-time stack. Similarly, 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 (the absence of security errors), 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 require whole program analysis. Also, the Permission.implies and Permissions.implies methods, including AllPermission, are not modeled. Modeling these classes and methods are important when simplifying the access rights policy descriptions so that the results are usable.

In the aforementioned works, assumptions are made that (1) call graph algorithms are available to translate the theoretical approach into a practical implementation, and (2) there is an authorization object, *p*, and a single authorization point, the checkPermission method. For Java 2 this is not correct. Almost all of the code in the Java runtime calls one of the SecurityManager authorization methods, though many of these methods subsequently call AccessController.checkPermission. Many of the well-known call graph and data flow algorithms [16] are too conservative to correctly identify authorization requirements. In this paper we describe an invocation graph and data flow analysis that minimizes the conservativeness to get more accurate authorization information.

Felten, Wallach, Dean and Balfanz have studied a number of security problems related to mobile code [32] [11] [31] [9] [30] [10] [8]. In particular, they present a formalization of stack introspection, which examines authorization based on the principals (signers and/or origin of the code) currently active in a Thread stack at run-time, as is found in Java. In particular, an authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [31]. Each method is modified so that it passes a security token as part of each method invocation. The token represents an encoding of the active principals (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, security passing style explores subgraphs of the comparable invocation graph, and discovers the security states and authorizations associated with those states. Wallach assumes that all authorization tests are temporally invariant, so that once an authorization test succeeds or fails in a particular security state, it will always succeed or fail in that state. In Java 2 this invariance does not exist. A Permission.implies invocation can revoke a class' access rights, or may use state unrelated to the runtime stack when determining the result of the test. In practice, especially in web server environments, access rights are revoked while the JVM is running. Our approach is not to optimize the authorization performance, but to discover authorization requirements by analyzing all possible paths through the program, 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 [12] describe a system that inlines reference monitors into the code to enforce specific security policies. The objective is to define a security policy and then inject authorization points into the code. This approach can reduce or eliminate redundant authorization tests. We examine the authorization issue from the perspective of examining an existing system containing authorization test points. Through static analysis, we discover how the security policy needs to be modified / updated to enable the code to execute.

3. Invocation Graph Characteristics

We use a path-insensitive, flow-sensitive, context-sensitive invocation graph. A *path-insensitive* invocation graph analyzes all paths through all basic blocks in each method. Because Java 2 authorization is based on associating rights with classes, a path-insensitive invocation graph construction algorithm is sufficient. The invocation graph is *flow-sensitive* for intraprocedural analysis because it considers the order of execution of the instructions within each basic block, accounting for local variable kills and casting of object references. The invocation graph (interprocedural analysis) is *context-sensitive* because it uniquely distinguishes each node by its *calling context*: the target method, receiver and parameters values.

For the purposes of describing the Permission culling algorithm, the invocation graph has the following characteristics:

- Each node in the graph represents a context sensitive method invocation.
- Each node in the graph is uniquely identified by its calling context, so no two nodes in the graph have the same calling context.
- Each node in the graph contains the following state:
  - The target method.
  - For instance methods, an allocation site (or type) for the method's receiver.
  - All parameters to the method, represented as a vector of sets of possible allocation sites (or possible types).
  - A set of possible return value allocation sites (or types) from this method.
- The edges in the graph are directed, where each edge points from a call site to a target method.
- The graph is rooted and may be cyclic.
- Our representation of the graph allows bi-directional traversal, even though the edges in the graph are unidirectional. Therefore, from any node within the graph, we can find all of its predecessor nodes.

In addition to the invocation graph construction, we use a data flow analysis with a precision to the level of allocation sites (CFA(1) [27]) and include the propagation of string constants. In a limited number of cases, data flow on Permission objects is computed using a CFA(2)-like algorithm to reduce the conservativeness of the analysis (see Section 4.). We are particularly interested in the string constants since they are used as parameters to Permission object constructors. The string constant values passed to the constructor fully qualify the access rights requirement.

4. Authorization Model – Access Rights Invocation Graph (ARIG)

Each Java application class is loaded into the Java Virtual Machine and is associated with a set of rights, or privileges, granted to the code. Statically determining this set of rights is nontrivial because it involves identifying, as accurately as possible, the precise set of methods callable from any point in a program execution. If any method is omitted, the analysis is incomplete. Conversely, when the analysis is overly conservative, the large number of false positives violates the Principle of Least Privilege [26], rendering the analysis ineffective for practical use.

We model the Java 2 authorization algorithm using a graph and set theoretic approach as follows. Let $\mathbf{G} = (\mathbf{N}, \mathbf{E})$ be the invocation graph representing a program $\mathcal{P}$. The nodes are described by $\mathcal{N} = \{\, n(M, R, \mathbf{P}) \mid M$ is the target method, with receiver R and k parameters $\mathbf{P} = \langle P_i \rangle_{i=1,\ldots,k}$, where parameter i can have possible types $P_i\}$. Each node represents the intraprocedural analysis of a method, including the virtual call sites within the method, and subsumes the control flow graph for that method. The edges are described by:

$$\mathbf{E} = \{ e(n_p(M_p, R_p, \mathbf{P}_p), n_s(M_s, R_s, \mathbf{P}_s)) \mid M_s \text{ is invoked within } M_p \}.$$

Given a node $n \in \mathbf{N}$, we define $\Gamma^+(n) := \{ n' \mid \exists e(n, n') \in \mathbf{E} \}$ and $\Gamma^-(n) := \{ n' \mid \exists e(n', n) \in \mathbf{E} \}$, known as the outward and inward adjacencies of n, respectively.

We can extend these definitions to sets of nodes. Thus, given $N \subseteq \mathbf{N}$, we define $\Gamma^+(N) := \bigcup_{n \in N} \Gamma^+(n)$ and $\Gamma^-(N) := \bigcup_{n \in N} \Gamma^-(n)$.

We also define:

- $N_{\text{root}} := \{ n \in \mathbf{N} \mid \Gamma^-(n) = \Phi \}$, the set of root nodes in the invocation graph
- $N_{\text{dp}} := \{ n(M, R, \mathbf{P}) \mid M \text{ is AccessController.doPrivileged} \}$
- $N_{\text{cp}} := \{ n(M, R, \mathbf{P}) \mid M \text{ is AccessController.checkPermission} \}$
- $N_{\text{start}} := N_{\text{root}} \cup \Gamma^-(N_{\text{dp}})$
- $N_{\text{stop}} := N_{\text{cp}}$

For any node n, $\mathbf{RP}(n)$ indicates the set of required Permissions for n. Similarly, given a set of nodes $N \subseteq \mathbf{N}$, $\mathbf{RP}(N)$ denotes the required Permissions for the nodes in N. This implies that $\mathbf{RP}(N) = \bigcup_{n \in N} \mathbf{RP}(n)$. For each node $n \in \mathbf{N}$, the algorithm determines $\mathbf{RP}(n)$ by starting from $\mathbf{RP}(N_{\text{stop}})$, and tracing paths back from nodes in N_{stop} to nodes in N_{start}.

For each node n in N_{stop}, $\mathbf{RP}(n)$ is defined to be the set containing the single element p, where p is the Permission being checked at n. Thus, if we define $\mathbf{CP}(n)$ to be the Permission checked at n itself, we have $\mathbf{CP}(n) = \begin{cases} \{p\}, \forall n \in N_{\text{cp}} \\ \Phi, \forall n \notin N_{\text{cp}} \end{cases}$

For any node $n \in \mathbf{N}$, if there is a path $\pi(n,s)$ from n to another node s, n requires all the Permissions that s does. Therefore, it must be $\mathbf{RP}(n) \supseteq \mathbf{RP}(s)$. We thus compute $\mathbf{RP}(n)$ recursively from $\mathbf{RP}(n) = \mathbf{CP}(n) \cup \bigcup_{s \in \mathbf{N} \mid \exists \pi(n,s)} \mathbf{RP}(s)$.

Since, in general, $\mathcal{G}$ contains cycles, the algorithm is a fixed-point computation, starting with estimates for $\mathbf{RP}(s)$ for every s in N_{stop} and working backwards, using the Γ^{-} function, along paths towards nodes in N_{start}. This process associates a set of required Permissions $\mathbf{RP}(n)$ with each node n in N_{start}. More precisely, it computes $\mathbf{RP}(n)$ for all $n \in \mathbf{N}$.

Finally, $\mathbf{RP}(C)$, the set of Permissions required for a class C, can then be computed as $\mathbf{RP}(C) = \bigcup_{n \in N(C)} \mathbf{RP}(n)$, where $N(C)$ is the sets of nodes n whose methods are declared in class C. Indeed, in this manner, we can compute $\mathbf{RP}(N)$ for any set of nodes $N \subseteq \mathbf{N}$.

Note that Permissions propagated upwards via a doPriviledged node do not propagate beyond the predecessor of the doPrivileged node. Thus, the above definition (or the algorithm based on it) must be refined to replace $\Gamma^{-}(n)$ by $\Gamma^{-}(n,p)$, where

$$\Gamma^{-}(n,p) = \begin{cases} \phi, \text{when } p \text{ was propagated upwards to } n \\ \qquad \text{via a doPrivileged node} \\ \Gamma^{-}(n), \text{otherwise} \end{cases}$$

Lastly, many security authorization tests in Java 2 are made through calls to methods in the SecurityManager class. In the reference implementation of this class, most of the SecurityManager authorization tests are performed by calling AccessController.checkPermission with an appropriate Permission object. Details about the classes, objects and algorithms employed by Java 2 authorization are treated in-depth in Gong's and Pistoia's Java 2 security books [15] [23].

It can be seen that the data flow analysis described above does indeed converge to a fixed point by observing that the transfer function relating the value of $\mathbf{RP}(n)$ at the output of any invocation graph node, n, to its value at the input to that node is in fact the identity function and the value of $\mathbf{RP}(n)$ at the input to a node n is formed from the values at the outputs of nodes in $\Gamma^{-}(n)$ by means of a set union operation. Thus, $\mathbf{RP}(n)$ is monotone, specifically it is a non-decreasing function as our computation proceeds. The values of the $\mathbf{RP}(n)$ at each invocation site form a lattice [18] and, since the set of types within our analysis scope is finite, we are guaranteed that the computation converges to a unique fixed point in finite time, regardless of the order in which we visit the nodes in the invocation graph.

It should be noted also that uniquely identifying a node in the ARIG by its calling context does not introduce any additional conservativeness to the analysis. In fact, any two invocations of the same method with the same calling context would generate the same invocation subgraph. Therefore, they would require the same set of Permissions.

To clarify all of the concepts introduced in this section, consider the following simple example. A class C implements a method, methodC, which takes as argument the name of a file, and returns a FileInputStream for that file, as shown next:

```
FileInputStream methodC(String fileName) {
   return new FileInputStream(fileName);
}
```

Creating a FileInputStream involves a security check.

methodC is called from within methodA in class A and methodB in class B, the only difference being that while methodA calls methodC with a specific parameter, `"file1.txt"`, methodB calls methodC with two possible parameters depending on the value of a boolean expression, as shown in the two following snippets of code:

```
FileInputStream methodA() {
   String fileName = "file1.txt";
   C c = new C();
   return c.methodC(fileName);
}

FileInputStream methodB() {
   String fileName;

   if (Math.random() > 0.5)
      fileName = "file.txt";
   else
      fileName = "file2.txt";

   C c = new C();
   return c.methodC(fileName);
}
```

Evaluating the boolean expression in methodB cannot be done during the static analysis of the code. Therefore, the conservative and most secure approach requires that the execution of both the branches be considered. Therefore, the call to methodC must be represented taking into account both the parameters `"file1.txt"` and `"file2.txt"`. The following figure shows a simplified version of the invocation graph representing the program:

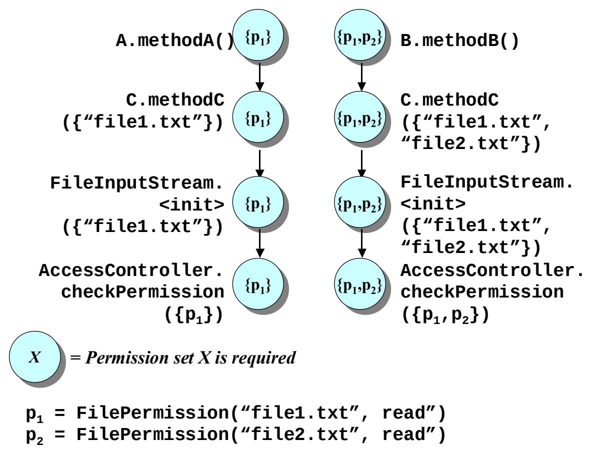

A.methodA()
{p₁}
C.methodC ({"file1.txt"})
FileInputStream.<init> ({"file1.txt"})
AccessController.checkPermission ({p₁})
{p₁,p₂}
B.methodB()
C.methodC ({"file1.txt", "file2.txt"})
FileInputStream.<init> ({"file1.txt", "file2.txt"})
AccessController.checkPermission ({p₁,p₂})
X = Permission set X is required
p₁ = FilePermission("file1.txt", read")
p₂ = FilePermission("file2.txt", read")

Figure 1. Sample Program ARIG Graphical Representation

As the ARIG in Figure 1, shows, classes B and C require both Permissions p_1 and p_2, while class A require only p_1.

5. The Permission Culling Algorithm

We describe an algorithm to statically identify the set of rights required by each analyzed class. The algorithm identifies paths in an invocation graph [2] [3] [6] [7] [28] [29] leading to AccessController.checkPermission nodes. By using a data flow analysis [21] the algorithm determines the set of possible Permission objects that could be passed as the argument to this method.

5.1 The Basic Permission Culling Algorithm

The basic algorithm uses the ARIG previously described to identify the set of Permissions representing the access rights required for each analyzed call site. The algorithm then aggregates these Permissions by the calling methods and their declaring classes. The algorithm identifies all nodes in each path bounded by any N_{start} node and an N_{stop} node and associates a set of Permissions with each of the nodes in the path. In a running system, each checkPermission method call has a single Permission object passed as an argument. In our analysis, which is path insensitive, this argument is a set of possible Permissions.

Each method in the path, and the method's declaring class, is marked as requiring the set of Permissions. In addition, the String parameters from Permission constructors are obtained through the data flow analysis. The parameter values provide necessary qualification of the authorization requirement. A typical authorization is described by constructor call `FilePermission("/tmp/file1","write")`. Pseudo code for this algorithm is in Appendix 1. The following figure graphically represents the Basic Permission Culling algorithm:

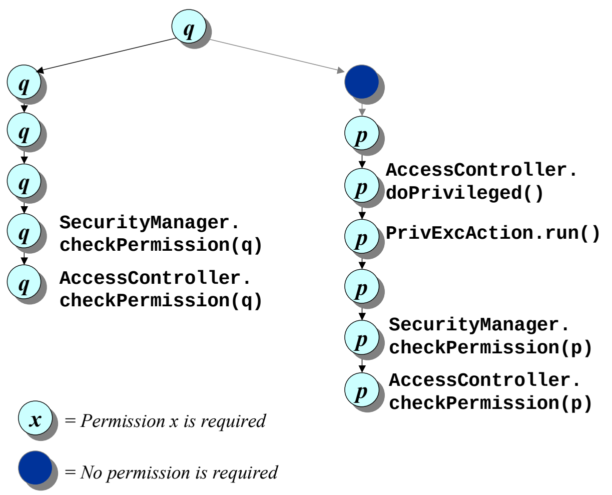

q
SecurityManager.checkPermission(q)
AccessController.checkPermission(q)
p
AccessController.doPrivileged()
PrivExcAction.run()
SecurityManager.checkPermission(p)
AccessController.checkPermission(p)
x = Permission x is required
= No permission is required

Figure 2. The Basic Permission Culling Algorithm

5.2 Reducing the Conservativeness of the Access Rights Analysis

The basic algorithm as described leads to an overly conservative result as is shown in the following figure. The subgraph represents part of the standard Java 2 SecurityManager control and data flow:

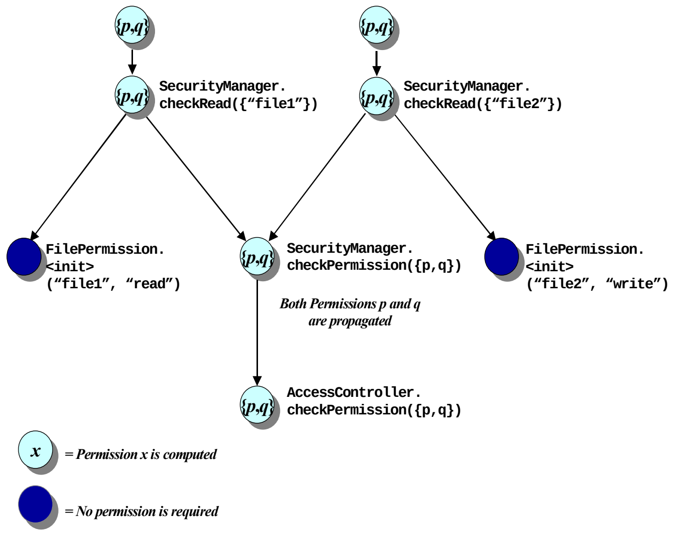

{p,q}
SecurityManager.checkRead({"file1"})
SecurityManager.checkRead({"file2"})
FilePermission.<init> ("file1", "read")
SecurityManager.checkPermission({p,q})
FilePermission.<init> ("file2", "write")
Both Permissions p and q are propagated
AccessController.checkPermission({p,q})
x = Permission x is computed
= No permission is required

Figure 3. Conservativeness Introduced by the SecurityManager

The problem is that the FilePermission allocation site corresponds to two different FilePermission constructor calls as a result of the two different paths leading to the checkRead method. The Basic Permission Culling Algorithm propagates the Permission set $\{p,q\}$ even when only p or q, but not both, is required. This violates the Principle of Least Privilege. We selectively reduce the conservativeness by using the node of the Permission's allocation site (a CFA(2)-like approach [27]) to differentiate the Permission allocations in the SecurityManager. In practice, this approach appears to be sufficient. The following figure shows the result:

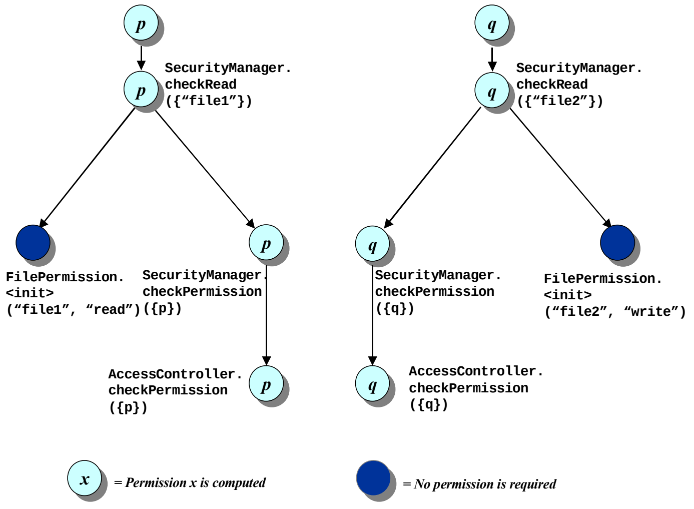


Figure 4. Conservativeness Reduction through Selective CFA(2)

We define $\Gamma^{-}(n,P)$, the inward adjacency of a node n with respect to a set P of Permissions, as the set of predecessor nodes of n with respect to the allocation sites of the Permissions in P. The algorithm proceeds as before, with $\Gamma^{-}(n,P)$ in place of $\Gamma^{-}(n)$. In practice, this refinement is sufficient for those Permissions allocated in one of the SecurityManager check methods that exhibit the behavior similar to the checkRead method described above.

5.3 Threads

The construction of a new Thread object does not cause it to start execution. Nominally, a call to the Thread.start method results in the new Thread beginning execution at the Thread.run method. The new Thread can be started by a different Thread other than that which created it, or by a method in a class other than that which created the Thread. The run method becomes the root (starting) method for the Thread.

According to the Java 2 authorization model, a new Thread requires that all predecessor nodes of the newly created Thread's constructor node also require Permission set P. Also, $\Gamma^{-}(n)$, where n is a Thread.run node, does not require P. We extend the Γ^{-} function as follows:

$$\tilde{\Gamma}^{-}(n,P)=\begin{cases} n_{\mathrm{Th}}, \text{ if } n \text{ is a Thread.run node} \\ \Gamma^{-}(n,P), \text{ otherwise} \end{cases}$$

where n_{Th} is the Thread constructor node for n's receiver.

The pseudo code is in Appendix 2. Graphically, we are rewriting the ARIG to contain a predecessor edge from a Thread.run call site to the Thread constructor as is shown in the following figure:

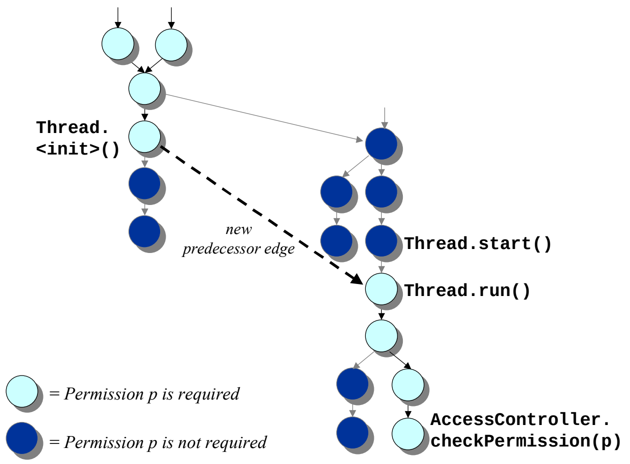


Figure 5. Modeling Threads

5.4 doPrivileged with an AccessControlContext

In addition to the AccessController.doPrivileged method described above, another form of the doPrivileged method takes an AccessControlContext instance as an argument. In addition to the previously described behavior, authorization tests include all of the predecessor nodes of the node where the AccessControlContext object was allocated. This is modeled similarly to how we model new Threads, by augmenting $\tilde{\Gamma}^{-}$ to include an edge from the doPrivileged node to the node where the AccessControlContext was allocated. Specifically, we extend $\tilde{\Gamma}^{-}$ as follows:

$$\hat{\Gamma}^{-}(n,P)=\begin{cases} N_{\mathrm{ACC}}(n), \text{ if } n \text{ is a doPrivileged node with an} \\ \qquad\qquad \text{AccessControlContext} \\ \tilde{\Gamma}^{-}(n,P), \text{ otherwise} \end{cases}$$

where $N_{\mathrm{ACC}}(n)$ is the set of nodes where AccessControlContext.<init> is invoked to create any of n's AccessControlContext parameters.

6. Computational Experience

The practical usefulness of an ARIG depends on the conservativeness of the underlying invocation graph and data flow analysis. In this section we discuss the trade-offs of using relatively less conservative invocation graphs, and we explain through practical results why we selected a context sensitive, selectively CFA(2) invocation graph. We discuss also the limitations that naturally arise in the identification of string constants representing the parameters to Permission constructors. Finally, we show some experimental results.

6.1 Analysis Conservativeness

Previous work using static analysis for Java authorization analysis has not discussed the implications of conservativeness of invocation graph and data flow analysis techniques. The conservativeness of the invocation graph and data flow analysis greatly affects the Permission Culling Algorithm results. An overly conservative control and data flow analysis is likely to determine access rights requirements for classes that do not need them, thus violating the Principle of Least Privilege. Class Hierarchy Analysis (CHA)-style graph construction algorithms [7] result in java.security.Permission and all of its subclasses being required for all classes needing authorization. A similar result holds for Rapid Type Analysis (RTA)-style algorithms [2], except that the access rights requirements for any class needing any Permission will include all of those Permissions that have allocation sites within the call graph. A CFA(0) call graph [27] would lead to overly conservative results because it does not consider the calling contexts. Without the calling contexts, it not possible to differentiate the AccessController.checkPermission calls. Similarly for the calls to the SecurityManager, where CFA(2) is used to selectively differentiate the required Permissions, as shown in Section 5.2. In addition, doPrivileged nodes would be collapsed, causing all nodes prior to doPrivileged to require the same set of Permissions, even though the required Permissions should usually be different. Again, this is overly conservative, resulting in a violation of the Principle of Least Privilege.

Experience has shown that context-sensitive invocation graphs yield less conservative results. Propagation-based call graph construction algorithms have been studied extensively, and differ primarily in the number of sets used to approximate run-time values of expressions [16]. The *Cartesian Product Algorithm* (CPA) [1] uses an approach based on parametric polymorphism. Given a method invocation to analyze, CPA computes the Cartesian product of the types of the actual arguments to the method. The invocation graph we use is similar to Agesen's, except we use what he refers to as *megamorphic sets* to represent parameters. Also, we consider data polymorphism, while Agesen does not. The invocation graph construction algorithm we use is similar the one described by Plevyak and Chien [24].

To minimize conservativeness, we use a graph construction algorithm that is context sensitive, flow sensitive, and path insensitive. By *flow sensitive* we mean that the analysis considers the order of execution of instructions both intra- and inter-procedurally thus improving the accuracy of the resulting graph. Part of our graph construction is flow sensitive for local variables, including support for local *field kills* – local fields that are overridden by subsequent assignments to the same field – and type casting. However, our handling of instance and class (static) fields is flow-insensitive because we use the weak assumption that all instance and class fields are subject to modification at any time due to multi-threading. To compensate for class and instance field flow-insensitivity, the data flow analysis tracks field values by allocation site. The practical problem that arises is that an allocation site does not directly map to a node in the invocation graph, thus making the analysis somewhat more conservative than we would otherwise like. Specifically, there is a one-to-many mapping of allocation sites to nodes in the graph. However, we have observed, by closely examining output from our tool and corresponding source code, that using allocation sites is sufficient to compensate for imprecision resulting from being interprocedurally flow insensitive.

Path insensitive intraprocedural analysis considers all paths through a method. This is conservative because input values or the values of constants defined within the program are not considered. For example, in the statement:

```
if (false) exp1 else exp2
```

both exp1 and exp2 are evaluated even though, in practice, exp1 never gets executed. While conservatively correct, this may result in additional graph nodes being generated for paths not occurring in practice. The net effect is that all required Permissions are included, though some additional Permissions may be included that are not strictly necessary in all executions of a program. A future version of the tool could consider adding a level of path sensitivity to minimize this conservative characteristic.

The graph construction algorithm that we use is selectively CFA(2). In particular, we use the node of the Permission's allocation site to differentiate Permission allocations in the SecurityManager. The CFA(1) portion of our algorithm defines the calling context to be the allocation sites of the possible receivers and parameters. Unfortunately, this approach alone does not distinguish between two different Permissions allocated at the same bytecode offset in the SecurityManager (see Section 5.2). As a result, it would be impossible to distinguish between two different Permissions of the same type, where the Permission object is allocated at the same location in the SecurityManager. As we explained in Section 5.2, the selective use of a CFA(2)-like approach eliminates this additional level of conservativeness without significantly impacting CPU and memory usage.

6.2 Limitations on String Constant Identification

As a practical matter, parameters to Permission constructors are string constants. In some cases, the parameter value may be available only at run-time. For example, it may be required that the name of a file to be opened be specified as input. To improve the analysis, it is possible to provide some metadata to reflect the run-time values. In other cases, the parameter value could be the result of a computation (e.g., string concatenations). A slightly more sophisticated data flow analysis is required in such circumstances.

Application	Classes		Methods Analyzed	Instruction bytes	Analysis time (sec.)			Nodes	Edges	Heap size (MB)
	Scope	Analyzed			Call graph	ARIG	Total			
javadoc	6,720	635	2,874	190,394	162	8	170	21,042	60,935	128
GetProperty	5,446	378	1,545	91,471	32	6	38	9,326	15,743	65
CountMain	5,451	383	1,552	91,809	32	6	38	9,339	15,763	65
java.lang.*	5,445	472	2,319	126,105	51	6	57	13,578	23,250	93
ECPerf Corp	21,291	1,912	2,854	426,685	94	7	101	18,330	36,479	120

Table 1. Comparison of Analyses

6.3 Experimental Results

Context and flow sensitive static analysis has a reputation for requiring significant processing power and memory. We have performed authorization analysis on a number of sample programs (see Appendix 3), parts of rt.jar, selected middleware, the Java API documentation generator (javadoc), and part of the ECPerf benchmark program. The results reported in Table 1. are from running our analysis on a system with an AMD Athalon 933 MHz processor, Windows 2000 SP2 with 768 MB RAM and using the Java Development Kit (JDK) V1.3.1. JDK V1.3 functionality was made part of the analysis scope by including the JDK V1.3 rt.jar.

Performance is improved by ignoring methods that do not lead to calls to AccessController.checkPermission, but whose data flow analysis requires a substantial amount of time. By forcing the underlying invocation graph to ignore the class constructor for sun.io.CharacterEncoding as well as all methods in classes Object, String, StringBuffer, and NullPointerException in package java.lang, and classes TimeZone and SimpleTimeZone in package java.util, we improved execution time significantly without affecting the resulting Permission sets identified. In particular, this optimization is a consequence of the fact that the analysis could be performed incrementally. Once it has been established that the invocation of a particular library will never lead to a security check, the data flow analysis for that library can be avoided.

The simplest example that we analyzed is an application called GetProperty, whose main method was the only root method and contained the following two lines of code:

```
System.setSecurityManager(new SecurityManager());
System.out.println(System.getProperty("user.home"));
```

The authorization requirements produced were precisely those that we expected based on examination of the source code:

```
java.lang.RuntimePermission "createSecurityManager"
java.lang.RuntimePermission "setSecurityManager"
java.util.PropertyPermission "user.home", "read"
```

The next example, CountMain, is more interesting because it makes use of privileged code. The method main was the only root method. The analysis computed access rights requirements that exactly reflected the presence of privileged code in the application. The source code for CountMain, as well as the computed access rights requirements, is reported in Appendix 3.

We also ran an analysis on the packages java.lang, java.lang.ref, and java.lang.reflect in rt.jar (part of the run-time classes for Java); all the public and protected methods of classes in these packages were considered as root nodes, including non-abstract public and protected methods in abstract classes, because they represent all the possible entry points that a program running on top of a library could invoke. The entire rt.jar V1.3 was included in the analysis scope. 2,811 root nodes were generated from 1,018 root methods, of which 336 were static methods, and the remaining 682 were instance methods. The number 2,811 comes from the fact that each static method gets counted once, because it has no receiver, and each instance method is weighted by the number of receivers. This results in an average of 3.629 receivers per root instance method.

The tool has successfully been run on large Java programs, the largest of which contain over 20,000 classes. Table 1. shows the statistics of running the analysis on the Corp part of the J2EE benchmark called ECPerf. To analyze this benchmark, the Java runtime plus all of the Enterprise JavaBeans (EJB) runtime classes are needed, resulting in an analysis scope of over 21,000 classes.

Other analyses were performed on large products (over 20,000 classes in the analysis scope) based on the Java security model of the JDK V1.1 platform. The goal was to identify its access rights requirements to allow it to successfully run with the Java 2 security model enabled.

7. Generation of a Security Policy Description

In Java 2, every concrete Permission class is required to implement the implies method. Given two Permission objects, *p* and *q*, when `p.implies(q)` returns `true` it means that any code that is granted *p* is also automatically granted *q*. Since we identify the String parameters to the Permission objects' constructors, we instantiate the Permission objects during the analysis, and use their implies methods to filter out those Permissions that are already implied by other Permissions. This minimizes the list of required access rights. When the parameters to a Permission constructor are not determinable, we impose that the resulting Permission cannot imply any other Permissions, even though stronger Permissions, such as java.security.AllPermission, still imply it. This also allows us to generate a security policy file containing the access right requirements needed at run-time.

The access rights requirements are minimal modulo the conservativeness of the analysis and the possible inability to determine some string constants. The resulting policy file is useful for defining new security policy, update an existing policy, or validate whether a path through the program will result in an authorization failure.

8. Conclusions

For a given application or classes in a library, we are able to conservatively identify the set of Java 2 Permissions required for each class in the analysis scope. By using a context-sensitive invocation graph, we are able to accurately identify the classes in each path that contains a call to the Java 2 security authorization subsystem. Our level of precision is far greater than that required for Java 2 security because we are able to identify access rights requirements to the level of methods and call sites, rather then the coarser granularity of classes or libraries. A refinement of the Java 2 authorization algorithm could result in the minimization of authorization, bringing us closer to the application of the Principle of Least Privilege.

By using the analysis technique described in this paper, we can determine the access rights requirements of mobile code, such as applets, servlets, and code that exploits mobile code features of RMI. Prior to loading the mobile code, it is possible to prompt the administrator or end-user to authorize or deny the code access rights for restricted resources protected by the Java 2 authorization subsystem.

Automating the process of determining required access rights changes the relationship between the developer of the code and the administrator / end-user. Instead of relying solely on recommendations from the developer, or resorting to trial-and-error testing of the code to determine required access rights, our tool can analyze the code and make its own recommendations and/or validate recommendations made by the developer. This shifts the relationship from one that *requires* that the developer be trusted, to something that can be verified.

While the analysis described here is specific to Java, the basic techniques can be applied to resource access rights determination in other type safe languages. With stronger analysis techniques, it may even be possible to apply the same approach to languages that lack type explicit safety but could rely on other mechanisms such as typed assembly language [20].

9. Acknowledgment

The authors would like to thank Dr. Sreedhar Vugranam for his invaluable technical contributions to this paper.

10. References

[1] O. Agesen. *The Cartesian Product Algorithm: Simple and precise type inference of parametric polymorphism*. In Proceedings of ECOOP ’95, Aarhus, Denmark, August 1995. Springer-Verlag, 1995.

[2] D.F. Bacon and P.F. Sweeney. *Fast static analysis of C++ virtual function calls*. In Proceedings of the Eleventh Annual Conference on Object-Oriented Programming Systems, Languages, Systems and Applications (OOPSLA’96), San Jose, CA. 1996, 324-341, ACM Press, New York. Also in ACM SIGPLAN Notices 31(10).

[3] D.F. Bacon. *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. PhD thesis, Computer Science Division, University of California, Berkeley, Dec. 1997. Report No. UCB/CSD-98-1017.

[4] A. Banerjee and D. A. Naumann. *A Simple Semantics and Static Analysis for Java Security*. Stevens Institute of Technology, CS Report 2001-1, July 2001.

[5] M. Bartoletti, P. Degano, and G. Ferrari. *Static Analysis for Stack Inspection*. Proceedings of ConCoord, Lipari, Italy, 6-8 July 2001, ENTCS 54, Elsevier Science B. V., 2001.

[6] C. Chambers, D. Grove, G. DeFouw and J. Dean. *Call graph construction in object-oriented languages*. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA’97), 108-124, Oct. 5-9, 1997, ACM Press, New York. Also in ACM SIGPLAN Notices 32(10).

[7] J. Dean, D. Grove and C. Chambers. *Optimization of object-oriented programs using static class hierarchy analysis*. In Proceedings of the Ninth European Conference on Object-Oriented Programming (ECOOP’95). Aarhus, Denmark, Aug. 1995. W. Olthoff, Ed., Springer-Verlag, 77-101.

[8] D. Dean, E.W. Felten, and D.S. Wallach. *Java Security: From HotJava to Netscape and Beyond*. Proceedings of the 1996 IEEE Syposium on Security and Privacy (Oakland, California), IEEE, May 1996.

[9] D. Dean. *The Security of Static Typing with Dynamic Linking*. Proceedings of the Fourth ACM Conference on Computer and Communications Security. (Zürich, Switzerland), April 1997.

[10] D. Dean, E. W. Felten, D.S. Wallach, and D. Balfanz. *Java Security: Web Browsers and Beyond*. Internet Beseiged: Counter Cyberspace Scofflaws, D.E. Denning and P.J. Denning, eds. ACM Press (NY, NY), October 1997.

[11] R.D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, New Jersey, January 1999.

[12] Ú. Erlingsson and F.B. Schneider. *IRM Enforcement of Java Stack Inspection*. Proceedings IEEE Symposium on Security and Privacy, pp. 246-255, Oakland, California, May 2000.

[13] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. *Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2*. Proceedings of the USENIX Symposium on Internet Technologies and Systems, 103-112, Monterey, CA., December 1997.

[14] L. Gong and R. Schemers. *Implementing Protection Domains in the Java Development Kit 1.2*. Proceedings of the Internet Society Symposium on Network and Distributed System Security, 125-134, San Diego, CA., March 1998.

[15] L. Gong. Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation. Addison-Wesley, Reading, MA. 1999.

[16] D. Grove and C. Chambers. *A Framework for Call Graph Construction Algorithms*. ACM TOPLAS, Vol. 23, No. 6, November 2001.

[17] T. Jensen D. Le Métayer and T. Thorn. *Verification of control flow based security properties*. IRISA, Publication interne n° 1210, October 1998.

[18] G. A. Kildall. *A Unified Approach to Global Program Optimization*. Proceedings of Principles of Programming Languages, pp. 194-206, 1973.

[19] G. McGraw and E.W. Felten. Securing Java™. John Wiley & Sons, Inc., New York. 1999.

[20] G. Morrisett, D. Walker, K. Crary, and N. Glew. *From system F to Typed Assembly Language*. In ACM Transactions on Programming Languages and Systems, 21(3):528-569, May 1999.

[21] S.S. Muchnick. Advanced Compiler Design And Implementation. Morgan Kaufmann Publishers, San Diego, CA, 1997.

[22] R. Oberg. Mastering RMI: Developing Enterprise Applications in Java and EJB. John Wiley & Sons, Inc., New York. 2001.

[23] M. Pistoia., D.F. Reller, D. Gupta., M. Nagnur., A.K. Ramani. Java™ 2 Network Security, Second Edition. Prentice Hall PTR, New Jersey, 1999.

[24] J. Plevyak and A.A. Chien. *Precise Concrete Type Inference for Object-Oriented Languages*. ACM OOPSLA'94, Object-Oriented Programming Systems, Languages and Applications, pp. 324-340, Portland, Oregon, October 1994.

[25] F. Pottier, C. Skalka and S. Smith. *A Systematic Approach to Static Access Control*. D. Sands (Ed.): ESOP 2001, LNCS 2028, pp.30-45, 2001. Springer-Verlag, Berlin Heidelberg 2001.

[26] Saltzer J.H. and M.D.Schroeder. *The Protection of Information in Computer Systems*. Proceedings of the IEEE 63 9 (Sept.1975), 1278-1308.

[27] O. Shivers. *Control-flow Analysis in Scheme*. ACM SIGPLAN Notices, 23(7):164-174, July 1988. Proceedings of the ACM SIGPLAN 1988 Conference on Programming Languages Design and Implementation.

[28] V. Sundaresan, L. Hendren, C. Razafimahefa, R. Vallee-Rai, P. Lam, E. Gagnon and C. Godin. *Practical Virtual Method Call Resolution for Java*. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2000), 264-280, Oct. 15-19, 2000, ACM Press, New York. Also in ACM SIGPLAN Notices 35(10).

[29] F. Tip and J. Palsberg. *Scalable Propagation-Based Call Graph Construction Algorithms*. In Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97), 264-280, Oct. 15-19, 2000, ACM Press, New York. Also in ACM SIGPLAN Notices 35(10).

[30] D.S. Wallach, D. Balfanz, D. Dean, E.W. Felten. *Extensible Security Architectures for Java*. 16th Symposium on Operating Systems Principles (Saint-Malo, France), October 1997.

[31] D.S. Wallach and E.W. Felten. *Understanding Java Stack Inspection*. Proceedings of the 1998 IEEE Symposium on Security and Privacy (Oakland, California), May 1998.

[32] D.S. Wallach. *A New Approach to Mobile Code Security*. PhD thesis, Princeton University, Princeton, New Jersey, January 1999.

Appendix 1

The following pseudo-code embodies the Basic Permission Culling Algorithm.

```
// Identify all start and stop nodes in the graph.
// The start set includes all nodes in the call graph root set.
Set startSet = new Set(rootNodes);
Set stopSet = new Set();// Initially, the stop set is empty.
// Identify additional start nodes and the stop nodes
// by iterating over all nodes in the call graph.
Iterator nodesIter = graphNodes.iterator();
while (nodesIter.hasNext())
   Node node = nodesIter.next();
    if (isDoPrivileged(node)
       startSet.add(node);
       doPrivSet.add(node);
   elseif (isCheckPermission(node))
       stopSet.add(node);

// Find all nodes between stop to start nodes and get the required Permission.
// For each node, get its method and associated class.
// Associate the Permission with each class identified.

// For each checkPermission(perm), identify Permission
// perm and the classes needing perm.
Iterator stopIter = stopSet.iterator();
while (stopIter.hasNext())
   Node stopNode = stopIter.next();
   // Get the Permission from the checkPermission() node
   RequiredPermission perm = getPermission(stopNode);

   // Using a work list algorithm, find all nodes in all paths that
   // are bounded by the nodes in the start set and the stop node.
   // Note: The graph may be cyclic.
   Set pathNodes = getPathsNodes(stopNode, start);
   // For each such node, get the node's method and the class that
   // declared that method. Add the Permission as being required for the class.
   Iterator nodesIter = pathNodes.iterator();
   while (nodesIter.hasNext())
      Node node = nodesIter.next();
      nodePerm.add(node, perm);
      Method method = node.getMethod();
      Class declaringClass = method.getDeclaringClass();
      // Remember that this class needs this Permission
      requiredPerms.add(declaringClass, perm);

// Propagate the Permissions at the doPrivileged node to all of
// its predecessors as is required by Java 2
Iterator doPrivIter = doPrivSet.iterator();
while (doPrivIter.hasNext))
   Node node = doPrivIter.next();
   Set reqPerms = nodePerm.get(node);
   PropagatePermsToPredecessorNodesClass(node, reqPerms);
```

At the end of this algorithm, each class in the requiredPerms map is mapped to the set of the Permission objects that it requires. From the allocation sites, we identify the string constants used in the Permission constructors. These string constants are used to report the required access rights for each class.

Appendix 2

We build a lookup table that maps Thread allocation sites to the graph nodes where the respective inherited AccessControlContext constructor is called. This mapping allows us create the replacement predecessor edge for the Thread.run method.

The following pseudo-code embodies the basic algorithm.

```
// For all Thread allocation sites, build a table that maps the
// Thread to its constructor.
Map threadConstructorMap = new Map();

// Iterate over all of the object allocations, selecting Thread allocations.
Iterator allocIter = allocationSites.iterator();
while (allocIter.hasNext())
       AllocSite allocSite = allocIter.next();
       if (allocSite.getClass() instanceof java.lang.Thread)
              threadConstructorMap.add(allocSite, node);
```

Now, when we reach a Thread.run node in the invocation graph, we can find its new predecessors by looking up the Thread allocation site and use it as the replacement predecessor edge. From the algorithm above, the getNodes method is suitably modified to use `allocCallSites` to find replacement predecessor nodes when searching the call graph.

Appendix 3

To illustrate computational experience with Permission analysis, we have made use of an application called CountMain. It is an interesting test case because it contains privileged code. From its main method, CountMain creates and sets a new SecurityManager, and then instantiates a CountFileCaller1 object and a CountFileCaller2 object, as shown in the following code:

```
import java.io.FilePermission;

public class CountMain {
    public static void main(String[] args) {
           System.setSecurityManager(new SecurityManager());
           CountFileCaller1.main(args);
           CountFileCaller2.main(args);
    }
}
```

The purpose of both CountFileCaller1 and CountFileCaller2 is to read the file C:\AUTOEXEC.BAT from the local file system. The code for CountFileCaller1 is shown next:

```
public class CountFileCaller1 {
   public static void main(String[] args) {
      try {
         System.out.println("Instantiating CountFile1...");
         CountFile1 cf = new CountFile1();
      }
      catch(Exception e) {
         System.out.println("" + e.toString());
         e.printStackTrace();
      }
   }
}
```

The following is the code for CountFileCaller2:

```
public class CountFileCaller2 {
   public static void main(String[] args) {
      try {
         System.out.println("Instantiating CountFile2...");
         CountFile2 cf = new CountFile2();
         cf.countChars();
      }
      catch(Exception e) {
            System.out.println("" + e.toString());
            e.printStackTrace();
         }
      }
   }
```

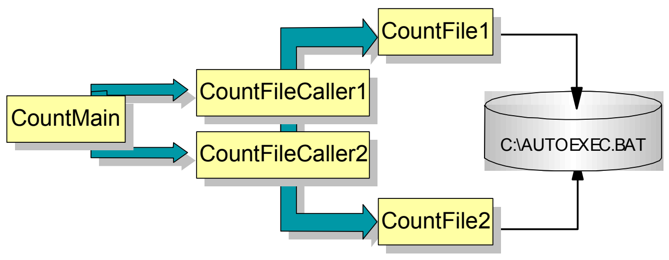
To perform the file read operation, CountFileCaller1 uses a supporting class called CountFile1, whereas CountFileCaller2 makes use of CountFile2. The difference between these two supporting classes is that CountFile1 wraps the code that performs the file read operation into a privileged block, whereas CountFile2 does not. This is evident by looking at the source code for CountFile1:

```
import java.io.*;
import java.security.*;

class PrivExcAction implements PrivilegedExceptionAction {
   public Object run() throws FileNotFoundException {
      FileInputStream fis = new FileInputStream("C:\\AUTOEXEC.BAT");
      try {
         int count = 0;
         while (fis.read() != -1)
            count++;
         System.out.println("Hi! We counted " + count + " chars.");
      }
      catch (Exception e) {
         System.out.println("Exception " + e);
      }
      return null;
   }
}

 public class CountFile1 {
    public CountFile1() throws FileNotFoundException {
       try {
          AccessController.doPrivileged(
                         new PrivExcAction());
       }
       catch (PrivilegedActionException e) {
          throw (FileNotFoundException) e.getException();
       }
    }
 }
```

CountFile2 attempts to gain file read access without using Privileged code, as shown next:

```
import java.io.*;

public class CountFile2 {
   int count=0;
   public void countChars() throws Exception {
      FileInputStream fis =
                                new FileInputStream("C:\\AUTOEXEC.BAT");
      try {
         while (fis.read() != -1)
            count++;
         System.out.println("We counted " + count + " chars.");
      }
      catch (Exception e) {
         System.out.println("No characters counted");
         System.out.println("Exception caught" + e.toString());
      }
   }
}
```

The following figure shows a graphical representation of the CountMain program structure:



The CountMain program will run as long as CountFileCaller2, CountFile1, CountFile2, and CountMain itself are all granted the Permission to read the file C:\AUTOEXEC.BAT. This requirement is waived for CountFileCaller1, which is temporarily given the Permission because CountFile1 invokes doPrivileged. In addition to that Permission, CountMain also needs the Permissions to create and set a new SecurityManager. The analysis reflected these Permission requirements exactly, as shown in the following table:

Classes	Permissions Determined by the Permission Culling Algorithm

CountMain	`java.io.FilePermission "C:\AUTOEXEC.BAT", "read"`
	`java.lang.RuntimePermission "createSecurityManager"`
	`java.lang.RuntimePermission "setSecurityManager"`

CountFileCaller1	
CountFile1 PrivExcAction CountFileCaller2 CountFile2	`java.io.FilePermission "C:\AUTOEXEC.BAT", "read"`