Access-rights Analysis in the Presence of Subjects

Paolina Centonze[1], **Marco Pistoia**[2], **and Omer Tripp**[2]

1 **Iona College**
New Rochelle, New York, USA
`pcentonze@iona.edu`

2 **IBM T. J. Watson Research Center**
Yorktown Heights, New York, USA
`{pistoia,otripp}@us.ibm.com`

Abstract

Modern software development and run-time environments, such as Java and the Microsoft .NET Common Language Runtime (CLR), have adopted a declarative form of access control. Permissions are granted to code providers, and during execution, the platform verifies compatibility between the permissions required by a security-sensitive operation and those granted to the executing code. While convenient, configuring the access-control policy of a program is not easy. If a code component is not granted sufficient permissions, authorization failures may occur. Thus, security administrators tend to define overly permissive policies, which violate the Principle of Least Privilege (PLP).

A considerable body of research has been devoted to building program-analysis tools for computing the optimal policy for a program. However, Java and the CLR also allow executing code under the authority of a *subject* (user or service), and no program-analysis solution has addressed the challenges of determining the policy of a program in the presence of subjects.

This paper introduces Subject Access Rights Analysis (SARA), a novel analysis algorithm for statically computing the permissions required by subjects at run time. We have applied SARA to 348 libraries in IBM WebSphere Application Server – a commercial enterprise application server written in Java that consists of >2 million lines of code and is required to support the Java permission- and subject-based security model. SARA detected 263 PLP violations, 219 cases of policies with missing permissions, and 29 bugs that led code to be unnecessarily executed under the authority of a subject. SARA corrected all these vulnerabilities automatically, and additionally synthesized fresh policies for all the libraries, with a false-positive rate of 5% and an average running time of 103 seconds per library. SARA also implements mechanisms for mitigating the risk of false negatives due to reflection and native code; according to a thorough result evaluation based on testing, no false negative was detected. SARA enabled IBM WebSphere Application Server to receive the Common Criteria for Information Technology Security Evaluation Assurance Level 4 certification.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages, D.4.6 Security and Protection

Keywords and phrases Static Analysis, Security, Access Control

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2015.222

1 Introduction

Modern software development and run-time environments, such as Java and the Microsoft .NET CLR, have adopted a form of declarative access control. Developers do not have to encode access-control-policy definition and enforcement capabilities inside their applications on a case-by-case basis because these capabilities are integrated within the underlying

29th European Conference on Object-Oriented Programming (ECOOP'15).
Editor: John Tang Boyland; pp. 222–246
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

platforms. Given the highly distributed nature of today's applications, and the fact that code can be dynamically loaded, it is essential to restrict resource access based on code providers. At run time, when a security-sensitive resource is about to be accessed, the access-control enforcement mechanism verifies that all the code responsible for that resource access is sufficiently trusted.

Decoupling access-control definition and enforcement from the application code promotes code portability and reusability, and minimizes the risk of security holes. Nevertheless, configuring the access-control policy of an application can be complicated. A security administrator must be informed of all the security-sensitive operations that an application may attempt to perform at run time and grant the program components all the permissions necessary to complete those operations. If some of the necessary permissions are not granted, run-time authorization failures will occur. On the other hand, granting unnecessary permissions would constitute a violation of a fundamental security rule, known as the Principle of Least Privilege (PLP) [33]. Therefore, it is essential that exactly the permissions necessary for the program to execute without authorization failures be granted.

1.1 Existing Approaches

For these complications, researchers have studied extensively how program analysis can be used to automatically infer an *optimal* access-control policy: one that is neither too restrictive nor too permissive. Numerous approaches have been proposed to address this problem in Java and CLR [31, 21, 7, 6, 36, 12, 29, 8, 38, 10, 37, 11, 13, 23]. More recently, permission analysis has been extended to Android [14, 4]. Hybrid techniques have also been attempted. For example, in order to better disambiguate the resources guarded by the various permissions and the mode in which those resources are accessed, static permission analysis has been integrated with string analysis [16], and in order to mitigate the false positives arising during static analysis and the false negatives typical of dynamic analysis, a hybrid static/dynamic approach has been studied [9]. Solutions have also been proposed that simultaneously integrate access-control and information-flow enforcement [5, 1, 15, 30].

What is critically missing from all existing approaches is treatment of *subjects*. These are users or services that map to one or more identities, called *principals*, each of which can be granted permissions. The concept of subject exists in Java and the CLR. Henceforth, we focus our discussion on Java for space and readability. In Java, the security component responsible for subject-based access control is the Java Authentication and Authorization Service (JAAS) [24]. Developers can invoke specific JAAS APIs to cause parts of a program to be executed under the authority of a subject. However, it is burdensome to determine manually the access-control policy of a program in the presence of subject-executed code. This amounts to resolving the principals associated with each subject, the permissions granted to each subject as a result of such associations, and the portions of the program executed under the authority of each individual subject.

1.2 This Paper

The work presented in this paper was motivated by the requirement to port IBM WebSphere Application Server[1] to the Java permission- and subject-based security model. While performing this complex task, we discovered severe access-control violations. Of the 348

[1] http://ibm.com/software/products/appserv-was

libraries comprising the application server, totaling over 2 million lines of code (MLOC), only 102 libraries had an associated policy with subject permissions in place. Those policies, which had been defined based on manual code inspection and unit testing, exhibited numerous errors. The remaining 246 policies did not account for subjects at all. Furthermore, developers had often used the subject APIs incorrectly, thereby introducing bugs in the code that could not be fixed by simply modifying a security policy.

In this paper, we show how we filled the gap between the need for a subject-sensitive access-rights analysis and the lack of an automated analysis framework to achieve this. We have formulated Subject Access Rights Analysis (SARA), the first comprehensive framework for access-rights analysis, whose purpose is to automatically determine which methods in the program under analysis are executed under the authority of a given subject at run time.

The input to SARA is the object code of a program. SARA statically builds a specialized representation of the program in the form of a context-sensitive call graph [32], so as to capture precisely run-time permission requirements. SARA annotates the call graph with subject-granted access rights and permission requirements, and uses this information to determine the subject-granted permissions under which a method will be executed at run time. This information is used to decide which access rights should be granted to subjects and code to prevent run-time authorization failures, and which access rights should be revoked to prevent PLP violations.

When applied to IBM WebSphere Application Server, SARA was able to synthesize access-control policies for the 105 libraries previously missing a specification. SARA was also able to detect all the errors we discovered manually in existing policies authored by seasoned developers and system administrators, and automatically revised the respective policies to eliminate all the errors. When given as input the 348 libraries of IBM WebSphere Application Server, SARA detected 263 PLP violations, 219 cases of policies with missing permissions, and 29 bugs that led code to be unnecessarily executed under the authority of a subject. SARA corrected all these vulnerabilities automatically, and additionally synthesized fresh policies for all the libraries, with a false-positive rate of 5% and an average running time of just over 100 seconds per library. SARA implements several mechanisms for mitigating the threat of false negatives due to the presence of native code or reflective method calls in the code under analysis. Based on a thorough evaluation of the results, no false negative was detected. SARA enabled IBM WebSphere Application Server to receive the Common Criteria for Information Technology Security Evaluation Assurance Level (CC EAL) 4 certification.[2]

2 Technical Background

This section describes how the Java and CLR stack-inspection mechanism works, focusing on Java as a reference.

2.1 Basic Concepts

In order to prevent confused-deputy attacks [20], when access to a restricted resource is attempted, all code currently on the call stack must be authorized to access that resource. In Java, the `SecurityManager`, if active, triggers access-control enforcement by invoking method `checkPermission` in class `AccessController`. This static method takes a `Permission` object `p` as a parameter and performs a stack walk backwards to verify that every caller in the

[2] http://commoncriteriaportal.org/

current thread of execution has been granted the access right represented by p. If that is not the case, a SecurityException is thrown.

Though programmatic security is possible, access rights are preferably granted declaratively, in a policy database external to the application code. This enhances code portability and reusability. Access rights are by default denied to all code and subjects. Untrusted code and subjects will only be allowed to perform basic operations that cannot harm the system. For a restricted operation to succeed, all the code on the thread's stack must be explicitly granted the right to execute that operation.

Access rights are represented as objects of type Permission. Each Permission type must be a subclass of the Permission abstract class. When a Permission object is constructed, it can take zero, one, or two String objects as parameters. If present, the first parameter is called the *target* of the Permission object and represents the resource being protected; the second parameter is called the *action* of the Permission object and represents the mode of access. The target and action are used to better qualify the resource guarded by the Permission object. For example, the following line of code can be used to construct a Permission object representing the right to access the log.txt file in read/write mode:

```
Permission p = new FilePermission("log.txt", "read,write");
```

Given a Permission object p, p's fully-qualified Permission class name along with p's target and action, if any, uniquely identify the authorization requirement represented by p. Therefore, for authorization purposes, a Permission object p can be characterized solely based on p's *permission identifier*, which consists of p's fully-qualified class name and the values of the String instances used to instantiate p. In fact, authorizations are granted to programs and principals by simply listing the corresponding permission identifiers in a flat-file policy database dubbed the *policy file*.

The Permission class contains an abstract method, implies, which itself takes a Permission object as a parameter and returns a boolean value. Every non-abstract Permission subclass must implement implies. If p and q are two objects of type Permission such that p.implies(q) returns true, then this means that granting p implicitly grants q as well. For example, p could be the Permission object constructed above and q could be the Permission object constructed with the following line of code:

```
Permission q = new FilePermission("log.txt", "write");
```

If p is an instance of AllPermission, then p.implies(q) returns true for any Permission object q.

Access rights may be granted to code based on the *code source*, which is a combination of the *code base* – the network location from which the code is coming – and the certificates of the entities that digitally signed the code. Access rights are granted to classes. At run time, each class is loaded by a class loader. When it loads a class, a class loader constructs a *protection domain* characterizing the origin of the class being loaded, and associates it with the class itself. A protection domain, which is represented as an object of type ProtectionDomain, encapsulates the class' code source (represented as a CodeSource object) and an object of type PermissionCollection containing all the Permission objects corresponding to the access rights granted to that code source. When invoked with an argument p of type Permission, method checkPermission performs a stack traversal backwards, and verifies that each of the ProtectionDomains in the current thread of execution contains at least one Permission that implies p. The set of Permissions effectively granted to a thread of execution is, therefore, the intersection of the sets of Permissions implied by all the ProtectionDomains associated with the stack.

2.2 Subjects-based Authorization

Authorization decisions can also be based on the subject executing the code. In Java, a subject is represented as an object of type `Subject`. When a subject is authenticated, the corresponding `Subject` instance is populated with associated identities called *principals*, represented as objects of type `Principal`. A subject may have multiple associated principals. For example, if the subject is a person, two of that subject's principals could be that person's name and social security number, depending on which means the person used for authentication.

Access rights are granted to code and principals, though not directly to subjects. The set of access rights granted to a subject is the union of the sets of access rights granted to the subject's authenticated principals. The `Subject` class exposes static methods `doAs` and `doAsPrivileged` to perform a restricted operation with the access rights granted to a subject.

- `doAs` accepts two parameters: The first is a `Subject` object, and the second is either a `PrivilegedAction` or `PrivilegedExceptionAction` object `o`. The code in the `run` method of argument `o` is executed with the intersection of the sets of `Permissions` granted to the code on the call stack. However, `doAs` adds the `Permissions` granted to the subject's principals to the stack frames following the call to `doAs`.
- `doAsPrivileged` is similar to `doAs`, but accepts an additional third parameter: an `AccessControlContext` object. This object encapsulates an array of `ProtectionDomain` objects. Just like `doAs`, `doAsPrivileged` also adds the `Permission` objects granted to the subject's principals to the subsequent stack frames. However, unlike the case of `doAs`, when the `checkPermission` method is called with a `Permission` parameter `p` after a call to `doAsPrivileged`, the predecessors of `doAsPrivileged` are not required to exhibit a `Permission` that implies `p`. Rather, `doAsPrivileged` demands the `ProtectionDomains` that are embedded in the `AccessControlContext` passed to it as a parameter to exhibit such a `Permission`.

A library can execute a security-sensitive operation without propagating the corresponding permission requirements to its clients. This is done by wrapping that operation into the `run` method of a `PrivilegedAction` or `PrivilegedExceptionAction` object `o`, and then by calling `AccessController.doPrivileged` with `o` as a parameter. As a motivation for this, consider the case of a library that has been designed to open socket connections on behalf of its clients. For any such socket connection, it is justified to demand that both the library and the client be granted the relevant `SocketPermission`. However, if that library additionally logs to a file the details of the connections that it opens, then it would not be appropriate to demand the `FilePermission` of the client. Only the library should be demanded that. If a malicious client were granted that `Permission`, then that client could compromise the integrity of the log file.

We define subject-executed code as *unnecessary* if it does not lead to any call to `checkPermission`, and as *redundant* if it leads to a call to `checkPermission` only through a `doPrivileged` call. Our objective, accomplished by SARA, is to detect subject-executed code that is unnecessary or redundant. Not only can such code constitute a PLP violation, but it can further negatively affect the performance of the program. SARA also detects and automatically corrects policies that are overly restrictive or overly permissive. An overly restrictive policy is one that does not grant the application code or the subjects executing it sufficient rights to meet the security requirements of the application, in which case run-time authorization failures may occur, causing the application to crash. An overly permissive policy grants subjects or code unnecessary permissions, which constitutes a PLP violation.

3 Technical Overview

In this section, we provide a high-level summary of the technical description appearing in the next two sections. We survey the flow and milestones of the SARA framework and highlight its main technical novelties.

3.1 Call-graph Representation

The first step of SARA is to model the behavior of the program `P` in the form of a call graph. The call-graph representation conveys the global control flow of the program. This is done by iteratively computing the structure of calls within `P` starting from the entry-point methods (*i.e.*, those that are externally invocable). Since some (in reality, most) of the call sites are virtual, requiring resolution of the receiver to disambiguate the target method(s), call-graph construction is interleaved with pointer analysis [18]. While other call-graph construction techniques exist [35], combining pointer analysis into call-graph construction boosts precision and makes points-to information available to downstream analyses, which we indeed utilize in SARA.

In the iterative process, call-graph construction resolves call sites into target methods, and meanwhile, pointer analysis resolves variables, array elements and object fields into abstractions of run-time objects. Specifically, each object is abstracted as its allocation site, which ensures finitely many object abstractions. Resolution of call sites may reveal more allocation sites to be tracked by the pointer analysis. At the same time, disambiguation of virtual call sites based on the points-to image of the receiver variable may enable resolution of more call sites. This process is iterated to fixpoint.

3.2 Permission Hierarchy

The next step is to model the partial order between `Permission` identifiers according to the `implies` relation, introduced in Section 2.1. Doing so purely statically is a serious challenge. `implies` is typically implemented as a series of nontrivial tests on its `Permission` argument, which are hard to track accurately in a static manner. At the same time, preciseness is crucial when modeling the permission structure, as this is the foundation underlying all the analysis steps to follow.

Given these considerations, we have adopted a novel strategy in SARA of allocating `Permission` instances dynamically and invoking their `implies` method in a sandboxed environment. For this, SARA first traverses the call graph to retrieve all the allocation sites of `Permission` subclasses. Then, for each allocation site in turn, a string analysis is applied to resolve constructor arguments into concrete values, as explained in Section 6.2. Assuming for now that the resolution is successful, SARA can recover the hierarchical relationship between `Permission` instances precisely by instantiating the instances and invoking `implies` in a sandboxed environment over all instance pairs. Sandboxing is achieved by activating the `SecurityManager`, and at the same time depriving the `Permission` receiver object of any permission. In this way, `implies` is prevented from performing any security-sensitive operation.

For cases where the string analysis fails, in part or in whole, we fall back on a per-permission specification of conservative argument approximations. As an example, inability to resolve the file pattern specified as the first argument of `FilePermission` is resolved conservatively as `"<<ALL_FILES>>"`, a notation used in Java to symbolize all the possible files in the file system. In situation where only a substring can be disambiguated, SARA resolves the file name partially. For example, if the name of a file guarded by a `FilePermission` object is obtained by concatenating `String` constant `"C:/"` with the dynamic value of `String` variable `x`, SARA computes the file name as `"C:/*"`, which is more precise than `"<<ALL_FILES>>"`. Similarly, inability to resolve the second argument, denoting the access mode, is resolved as `"read,write,execute,delete"`.

For application-specific `Permission` subclasses, the user of SARA has the option to extend the specification. If this is not done and the string analysis is unable to resolve a `String` value, we conservatively resolve the `Permission` as `AllPermission`. We comment, from our experience, that we rarely encountered failures by the string analysis.

3.3 Access-rights Annotations

Having a model of the program's calling structure and the relationships between the various `Permission` objects, SARA identifies, within the call graph, invocations of the access-rights-related APIs: namely `checkPermission` and `doPrivileged`. At a given call site invoking `checkPermission`, SARA retrieves a conservative approximation of the checked `Permission`s as the points-to set of the argument.

SARA then statically simulates the run-time stack inspection mechanism by propagating the requirement to possess the permission(s) arising at `checkPermission` backwards to all the transitive callers of `checkPermission`. As in the concrete semantics, backward propagation is aborted at `doPrivileged` callers. At this stage, every call-graph node is mapped to a set of required `Permissions`. The naive solution, as formulated by existing approaches, is to stop the analysis at this point and missing permissions to relevant code to ensure normal execution. However, this ignores the presence of `Subjects`, as we next explain.

3.4 Subject-specific Annotations

To account for `Subjects`, SARA next traverses the call graph in order to identify `doAs` and `doAsPrivileged` calls, wherein the first parameter is of type `Subject`. Again thanks to points-to information, that parameter is resolved into one or more abstract `Subject` instances. SARA then consults the points-to graph per each of the instances. In particular, the `Subject` class has a field `principals` of type `Set` that stores all the corresponding principals of a subject. As discussed in Section 2.1, it is the `Principal`s rather than the `Subject` that are granted `Permission`s, and so SARA uses the points-to graph to compute (i) the set of `Principal` object abstractions pointed to by the `principals` field and (ii) their respective constructor values via constant propagation. Next, `Permissions` are extracted from the policy based on the *principal identifiers*, consisting of concrete `Principal` subtypes and initialization arguments. The respective `Permissions` of a particular `Subject` object flowing into a `checkPermission` call are the union of all the `Permission`s granted to its corresponding `Principal`s, as determined conservatively via the points-to information.

With this information, SARA revisits the annotations computed in the previous step. The goal is to relax the demand for missing `Permission`s if these are provided already under the authority of a `Subject`. This is done as follows: At a given `doAs` or `doAsPrivileged` invocation point, SARA resolves the potential `Subjects` via the points-to map. Because the points-to information is conservative, the `Permissions` *guaranteed* to be provided at that point are the intersection, rather than union, of the `Permission`s held by the different `Subjects`. At this point, the `Permission`s shared by all `Subject`s are propagated forward and unioned with the `Permission`s that the given call-graph nodes already have.

This additional analysis step is significant: Naively the code may appear to be missing Permissions, thereby soliciting the system administrator to grant it undue access rights. Those can then be abused either by the code itself or by a malicious user. As an illustration, assume a text editor that requires a special Permission to edit a specific document. The subject owning that document possesses that Permission. Ignoring the subject, however, would lead the analysis tool to a recommendation that the code itself should own that Permission, effectively enabling other users to access the document.

3.5 Error Analysis

The final step is to analyze the artifacts computed by SARA and apply corresponding corrections or synthesis. The first scenario is detection of missing Permissions under consideration of Subjects. The solution, similarly to existing approaches, is to grant the missing Permissions. However, unlike existing approaches, SARA grants the Permissions by default to the subject rather than to the code. This is the more conservative policy, as granting Permissions to the code enables the respective operations per all subjects as well as the code itself. This is, however, configurable if the user wishes to override the default policy.

The second scenario arises when there is duplicity across the Permissions granted to the code and subject. That is, the code already possesses a needed Permission, but there is also a call to doAs or doAsPrivileged to enable that same Permission via the Subject. This form of redundancy constitutes a PLP violation. For the same rationale explained above, by default SARA revokes the Permission from the code rather than the subject, though again this is a configurable choice.

4 Static-analysis Framework

The static-analysis framework presented in this paper uses graph theory to represent the execution of a program and lattice theory to model the flow of information in the program. A program is modeled as a call graph and its associated points-to graph. A *call graph* is a directed graph $G = (N, E)$, in which nodes correspond to method invocations. Two nodes $n_1, n_2 \in N$ are connected by an edge $(n_1, n_2) \in E$ iff the analysis conservatively establishes that the method represented by n_1 may invoke the method represented by n_2 at run time [18]. A *points-to graph* is a directed bipartite graph in which each vertex corresponds to either a program variable or an object abstraction, and an edge indicates a points-to relation [3]. Both the call graph and the points-to graph presented in this paper are tailored to authorization analysis.

4.1 Permission Abstraction

When designing an algorithm that computes data flow over a graph, for Tarski's theorem to guarantee convergence to a fixed point in polynomial time, it is important to make sure that (i) the data-flow functions defined at each node map a lattice into itself, (ii) the lattice is complete and has finite height, and (iii) the data-flow functions are monotonic with respect to the lattice's partial order [17]. Although in Section 3 we intuitively talked about partial order between Permissions as well as unions and intersections of Permission sets, a more precise discussion is needed in order to formalize the analysis and guarantee its convergence. This section defines the lattice used as domain and codomain for the data-flow functions employed by SARA.

4.1.1 Permission Graph

The relationships induced by the `implies` methods among the `Permission` objects associated with a program under analysis can be modeled using a *permission graph* $H = (P, F)$, where P is a set of `Permission` objects and $F \subseteq P \times P$ is a set of edges of the form $\mathtt{p} \to \mathtt{q}$, where $\mathtt{p}, \mathtt{q} \in P$. Building the permission graph during the analysis requires detecting all the `Permission` objects that appear in the program, since each `Permission` object corresponds to an element of P. To build F, each $\mathtt{p} \in P$ must be explicitly instantiated. This operation is done during the analysis using reflection based on `p`'s permission identifier. Next, for each pair $(\mathtt{p}, \mathtt{q}) \in P \times P$, SARA runs `p.implies(q)` to decide whether $\mathtt{p} \to \mathtt{q} \in F$ or not.

4.1.2 Permission Lattice

The `implies` method does not necessarily induce a partial order on P. For instance, if `p` and `q` are two different `Permission` objects with the same permission identifier, then $\mathtt{p} \to \mathtt{q}$ and $\mathtt{q} \to \mathtt{p}$. Therefore, H does not have the structure of a lattice, which would be desirable when performing data-flow analysis. However, H can be transformed into a lattice as follows. Given $\mathtt{p}, \mathtt{q} \in P$, let $\mathtt{p} \overset{*}{\to} \mathtt{q}$ denote the presence of a path from `p` to `q` in H. `Permission` objects $\mathtt{p}, \mathtt{q} \in P$ belong to the same Strongly Connected Component (SCC) of H iff $\mathtt{p} \overset{*}{\to} \mathtt{q}$ and $\mathtt{q} \overset{*}{\to} \mathtt{p}$. In this case, `p` and `q` are said to be *equivalent*, denoted by $\mathtt{p} \equiv \mathtt{q}$. Let $H_{\equiv} = (P_{\equiv}, F_{\equiv}) = (P^{SCC}, F^{SCC})$ be the Directed Acyclic Graph (DAG) induced by collapsing each SCC of H into a single node. It is easy to see that $H_{\equiv}$ has the structure of a partially ordered set.

$H_{\equiv}$ can be given the structure of a lattice by adding two new elements to $P_{\equiv}$: $\top = \mathit{AllPermission}$, denoting the SCC containing all instances of the `AllPermission` class (if such an SCC does not already exist), and $\perp$, which denotes absence of authorizations. As we have already observed, an instance of `AllPermission` implies any other `Permission`. Therefore, $\forall p \in P_{\equiv}.\ \top \to p$. Additionally, we impose that $\forall p \in P.\ p \to \perp$. Let $\overline{P_{\equiv}}$ be the augmented set $P_{\equiv} \cup \{\top, \perp\}$, and let $\overline{F_{\equiv}}$ be the superset of $F_{\equiv}$ obtained by adding into $F_{\equiv}$ the edges involving the new elements $\top$ and $\perp$. It is easy to prove that the graph $\overline{H_{\equiv}} = (\overline{P_{\equiv}}, \overline{F_{\equiv}})$ represents the lattice $(\overline{P_{\equiv}}, \sqsupseteq)$, where $\sqsupseteq$ is the partial-order relation defined on a subset of $\overline{P_{\equiv}} \times \overline{P_{\equiv}}$ by $p \sqsupseteq q \iff p \overset{*}{\to} q, \forall p, q \in \overline{P_{\equiv}}$. This lattice is called the *permission lattice*. Its top and bottom elements are $\top$ and $\perp$, respectively. The *meet* and *join* operations $\sqcap, \sqcup \colon \overline{P_{\equiv}} \times \overline{P_{\equiv}} \to \overline{P_{\equiv}}$, induced by $\sqsupseteq$ on $\overline{P_{\equiv}}$, are defined as follows, respectively:

1. $p \sqcap q = r$, where $r \in \overline{P_{\equiv}}$ is such that $p, q \sqsupseteq r \wedge r \sqsupseteq x, \forall x \in \{\overline{P_{\equiv}} : p, q \sqsupseteq x\}$
2. $p \sqcup q = r$, where $r \in \overline{P_{\equiv}}$ is such that $r \sqsupseteq p, q \wedge x \sqsupseteq r, \forall x \in \{\overline{P_{\equiv}} : x \sqsupseteq p, q\}$

4.1.3 Permission-set Lattice

The authorization analysis performed by SARA is modeled as a data-flow problem wherein *sets* of access rights (rather than *single* access-right elements) are propagated through the call graph. The analysis performs meet and join operations on those sets. Therefore, it is necessary to lift the meet and join operations defined in Section 4.1.2 over elements of $\overline{P_{\equiv}}$ to sets of elements. In other words, it is necessary to define a lattice structure over the powerset $\mathcal{P}(\overline{P_{\equiv}})$.

Naturally, $\mathcal{P}(\overline{P_{\equiv}})$ has the lattice structure defined by regular set inclusion, $\supseteq$, which induces set intersection, $\cap$, and set union, $\cup$, as meet and join operations. However, the lattice $(\mathcal{P}(\overline{P_{\equiv}}), \cap, \cup)$ would not be appropriate for authorization analysis. For example, let `p` and `q` be two `Permission` objects instantiated through the two following lines of code:

```
Permission p = new FilePermission("C:\\*", "read");
Permission q = new FilePermission("C:\\log.txt", "read,write");
```

Following Section 4.1.2, let p and q represent their respective SCCs in the permission graph. SARA forward propagates elements of $\mathcal{P}(\overline{P_\equiv})$ across the call graph, where the operation applied to the lattice elements must be meet. If n is a node in the call graph, n_1 and n_2 are two predecessors of n, and SARA has established that n_1 and n_2 will be potentially executed with access rights represented by sets $\{p\}$ and $\{q\}$, respectively, then the safest assumption if the $(\mathcal{P}(\overline{P_\equiv}), \cap, \cup)$ lattice structure were adopted would be that n will be executed with an empty set of access rights since $\{p\} \cap \{q\} = \varnothing$. This result is overly conservative. Instead, let `r` be any `FilePermission` object instantiated through a line of code similar to the following:

```
Permission r = new FilePermission("C:\\log.txt", "read");
```

Since $p \sqcap q = r$, it is more desirable to conclude that n will be executed with access-right set $\{r\}$. Conversely, computing the permission requirements induced by the stack inspection mechanism requires backward propagation of $\mathcal{P}(\overline{P_\equiv})$ elements across the call graph. In such cases, the operation performed on the lattice elements must be join. If the set union operation, $\cup$, were applied to sets $\{p\}$ and $\{q\}$, then the result would be $\{p, q\}$. However, $p \sqcup q = v$, where v is the SCC of any `FilePermission` object `v` instantiated through a line of code similar to the following:

```
Permission v = new FilePermission("C:\\*", "read,write");
```

Therefore, it is more desirable to have a join operation on $\mathcal{P}(\overline{P_\equiv})$ that, once applied to $\{p\}$ and $\{q\}$, gives v as a result.

A more meaningful lattice structure can be given to $\mathcal{P}(\overline{P_\equiv})$ based on the lattice structure of $(\overline{P_\equiv}, \sqsupseteq)$. A set $Q \in \mathcal{P}(\overline{P_\equiv})$ is defined as *canonical* iff

$$\forall p, q \in Q.\ p \neq q \Longrightarrow (p \not\sqsupseteq q) \wedge (q \not\sqsupseteq p)$$

Intuitively, if $Q \in \mathcal{P}(\overline{P_\equiv})$ is canonical, then no element in Q implies any other element in Q except itself. A *canonical reduction* function $\chi : \mathcal{P}(\overline{P_\equiv}) \to \mathcal{P}(\overline{P_\equiv})$ can be introduced that maps any set $Q \in \mathcal{P}(\overline{P_\equiv})$ to its subset $\chi(Q) \in \mathcal{P}(\overline{P_\equiv})$ obtained by removing from Q all the elements that are implied by some other element of Q. Formally, $\chi(Q) = \{q \in Q : \forall r \in Q.\ r \not\sqsupseteq q\}$. The χ function is well defined because for any $Q \in \mathcal{P}(\overline{P_\equiv})$, there exists one and only one canonical set corresponding to Q [28]. Functions $\sqcap, \sqcup : \mathcal{P}(\overline{P_\equiv}) \times \mathcal{P}(\overline{P_\equiv}) \to \mathcal{P}(\overline{P_\equiv})$ are defined, respectively, as follows:

1. $\forall Q, R \in \mathcal{P}(\overline{P_\equiv}).\ Q \sqcap R = \chi(\{q \sqcap r : q \in Q, r \in R\})$
2. $\forall Q, R \in \mathcal{P}(\overline{P_\equiv}).\ Q \sqcup R = \chi(Q \cup R)$

It is easy to prove that both $\sqcap$ and $\sqcup$ are commutative, associative and mutually absorptive functions. Thus, $(\mathcal{P}(\overline{P_\equiv}), \sqcap, \sqcup)$ is a lattice, called the *permission-set lattice*, and $\sqcap$ and $\sqcup$ are its *meet* and *join* operations, respectively. For a given program or library, P is finite. This implies that the permission-set lattice associated with a program or library is also finite, and therefore complete. Its top element is the set $\{\mathit{AllPermission}\}$. Its bottom element is the empty set, $\varnothing$.

The partial order $\sqsupseteq$ induced by $\sqcap$ and $\sqcup$ on $\mathcal{P}(\overline{P_\equiv})$ is obtained as follows:

$$\forall Q, R \in \mathcal{P}(\overline{P_\equiv}).\ Q \sqsupseteq R \Longleftrightarrow Q \sqcap R = R$$

or, equivalently:

$$\forall Q, R \in \mathcal{P}(\overline{P_\equiv}).\ Q \sqsupseteq R \Longleftrightarrow Q \sqcup R = Q$$

Since the permission-set lattice is finite, its height, $\mathcal{H}(\mathcal{P}(\overline{P_{\equiv}}))$, is finite too. Specifically, $\mathcal{H}(\mathcal{P}(\overline{P_{\equiv}})) \in \mathcal{O}(|P|)$. Finally, the *difference operator*, $- : \mathcal{P}(\overline{P_{\equiv}}) \times \mathcal{P}(\overline{P_{\equiv}}) \to \mathcal{P}(\overline{P_{\equiv}})$, is obtained as follows:

$$\forall Q, R \in \mathcal{P}(\overline{P_{\equiv}}).\ Q - R = \chi(\{q \in Q : \nexists r \in R, r \sqsupseteq q\})$$

4.2 Permission-specific Call Graph

The first step in the analysis is to construct an augmented call graph, which we refer to as a *Permission-specific Call Graph* (PCG). SARA's call-graph and pointer-analysis implementation is based on the Watson Libraries for Analysis (WALA) static-analysis framework,[3] which allows for analysis of Java bytecode. A PCG is a directed multigraph $G = (N, E)$, where N is a set of nodes and E is a set of edges.

A node $n \in N$ represents a context-sensitive method invocation, and is uniquely identifiable via its *calling context*, which consists of (i) the concrete target method and (ii) the receiver and parameter values. n carries the following state: (i) the target method, (ii) object abstractions representing the receiver and parameters (in SARA, these are the concrete objects' allocation sites), and (iii) object abstractions representing the return value of the method. Therefore, a PCG is *context-sensitive* [32], because it uniquely distinguishes different invocations to the same methods by the calling context, with a context-sensitivity policy similar to Agesen's Cartesian Product Algorithm (CPA) [2]. With this policy, if a given method in a program is invoked twice, each time with different parameters according to the analysis abstraction, the PCG will model these two invocations as two distinct nodes.

An edge $e = m \xrightarrow{c} n \in E$ carries a label c denoting the call site through which the method `m` at m invokes the method `n` at n. Hence, G is a multigraph because `m` can invoke `n` multiple times, each time at a different call site. In the remainder of this paper, however, we will omit the label of an edge e, ad will simply write $e = (m, n)$, unless it is necessary to distinguish between different calls to `n` made by `m`.

The PCG has a unique node, $\overline{n} \in N$, called the PCG *root*. It represents external invocation of the entry points of the program under analysis. A node $n \in N$ is constructed iff it has been statically established that the method represented by n can be invoked during execution of the program under analysis. This guarantees that PCG nodes are reachable from $\overline{n}$. By inference, the PCG is a connected multi-graph.

We utilize Control-Flow Analysis (CFA) [27] during PCG construction to disambiguate heap objects according to their allocation sites. In addition to the CPA context sensitivity, the PCG construction process enforces the following properties:

- *Path insensitivity* [19], because it does not evaluate conditional statements and conservatively assumes that all branches are admissible
- *Intraprocedural flow sensitivity* [32], 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
- *Interprocedural flow insensitivity* [32], because it makes the conservative assumption that all instance and static fields are subject to modification at any time, which may happen in the presence of other execution threads
- *Field sensitivity* [32], because an object's fields are represented distinctly in the solution (motivated, for example, by the `principals` field of a `Subject`)

[3] `http://wala.sf.net`

```
grant
 Principal javax.security.auth.x500.X500Principal
    "CN=John Doe, OU=Res, O=Org, C=US"
 Principal javax.security.auth.kerberos.KerberosPrincipal "jd@us.res.org" {
  permission java.util.PropertyPermission "user.home", "read";
  permission java.io.FilePermission "log.txt", "read";
  permission java.security.SecurityPermission "getPolicy";
};
```

Figure 1 Granting Access Rights to `Principal`s.

Among all the design choices listed above, the most important one is the CPA context sensitivity: While naively the call graph would represent each method as a single node, it is possible to impose several different representations of the same method inside a call graph per different contexts governing the execution of the method. SARA distinguishes between invocations of a method based on parameter values (including the receiver). The reason behind this choice is that access-rights-related information is largely encoded as parameter values.

For example, the `String` values flowing into a call to the `FilePermission` constructor determine which file(s) are guarded by the `FilePermission` object being constructed and what access modes are allowed. Therefore, in order to statically distinguish `FilePermission` objects protecting different files, it is necessary to disambiguate calls to the `FilePermission` constructor made with different arguments.

Similarly, the `Permission` parameter passed to the `checkPermission` method determines what access right will be demanded of all the callers on the stack or the subject associated with the current thread. Thus, from a security perspective, CPA context sensitivity is crucial since an authorization analysis needs to distinguish between different calls to `checkPermission` based on the `Permission` argument passed to it. Failure to do so would result in all the `checkPermission` calls in the program being represented as a single PCG node, which in turn would lead to conservatively joining together all the `Permission` requirements identified in the program.

5 The SARA Algorithm

This section describes the various steps of the SARA algorithm.

5.1 Permission-set Lattice Construction

According to the JAAS architecture, at run time, an authenticated `Subject` is granted the `Permission`s of the `Principal`s encapsulated into it. `Permission`s in Java 2 are granted to a `Principal` based on two pieces of information:

1. The `Principal`'s fully qualified class name
2. The `Principal`'s *name*, which is the `String` value returned by the `getName` method of the `Principal` object

It is possible to include more than one `Principal` field in a `grant` statement, as in the policy file snippet of Figure 1. If a `grant` entry contains more than one `Principal` field, then the `Permission`s in the `grant` statement are awarded to the `Subject` associated with the current `AccessControlContext` only if that `Subject` contains *all* those specified `Principal`s.

Let $\mathcal{D}$ be the security policy database associated with the program under analysis, and let $|\mathcal{D}|$ indicate the number of grant statements in $\mathcal{D}$. The first step of the SARA algorithm is to scan $\mathcal{D}$, retrieve from it the permission identifiers that are mapped to each Principal class and name pair, and use reflection to instantiate the Permission objects corresponding to those permission identifiers. Such Permission objects form the set P, which can be used to construct the permission graph $H = (P, F)$ and permission lattice $(\overline{P_{\equiv}}, \sqsupseteq)$, representing the access rights *granted* to the Subjects of a program based on their Principals and the policy specified in $\mathcal{D}$.

It should be observed that it is not necessary to preconstruct the permission-set lattice $(\mathcal{P}(\overline{P_{\equiv}}), \sqcap, \sqcup)$; subsets of the permission lattice will be met at each node $n \in N$ every time an edge $e \in E$ of the form $e = (m, n) : m, n \in N$ is traversed during the fixed-point iteration described in Section 5.3. As will be explained in Section 5.3, in the worst case, the meet operation will have to be applied $\mathcal{H}(\mathcal{P}(\overline{P_{\equiv}}))$ times for each edge of the graph.

5.2 SARA Initialization

Let D be the subset of N consisting of all the nodes representing calls to doAs and doAsPrivileged. The first initialization step consists of computing D by simply iterating over N.

Both doAs and doAsPrivileged take a Subject object as one of the parameters. Let n be any element of D. SARA identifies the set $\psi_D(n) = \{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j\} \subseteq S$ of all the possible Subject allocation sites that may have flowed to the formal Subject argument of the method call represented by n, where S is the union of all the doAs and doAsPrivileged Subject parameter sets in G. This defines a function $\psi_D : D \to \mathcal{P}(S)$ that maps nodes of D to subsets of S.

An authenticated Subject is granted all the Permissions of its associated Principals. Therefore, for each Subject allocation site $\mathtt{s} \in S$, SARA identifies the set $\psi_S(\mathtt{s}) = \{\mathtt{u}_1, \mathtt{u}_2, \ldots, \mathtt{u}_i\}$ of U containing all the Principal allocation sites representing the equivalence classes of the Principal objects encapsulated in s in the analysis model, where U indicates the set of all the Principal allocation sites pointed to by the elements of S in the points-to graph associated with the PCG. Computing $\psi_S(\mathtt{s})$ requires identifying all the Principal instance keys in $\overline{N}$ pointed to by the principals Set field in s.

As discussed in Section 5.1, given a Principal object u encapsulated in an authenticated Subject s, two pieces of information about u are important when s needs to be authorized to access a restricted resource: the fully qualified class name of u and the name of u. These two pieces of information can be obtained statically.

The fully qualified class name of u is easy to retrieve in the analysis model since that information is stored into the allocation-site representation of u created by SARA. Let C indicate the set of all the fully qualified class names of the Principal allocation sites in U. Then function $v_C : U \to C$ maps each Principal allocation site to its fully qualified class name.

All the non-abstract classes inheriting from the Principal interface must implement instance method getName. At run time, for each Principal object u encapsulated in an authenticated Subject s, the Java authorization system invokes getName on u every time doAs or doAsPrivileged is called with s as the Subject parameter. This is to determine the Permissions granted to u by $\mathcal{D}$, which are added to the ProtectionDomains of the methods following doAs or doAsPrivileged on the stack. As a result, for any $\mathtt{u} \in U$, N contains at least one node representing the invocation of getName on u. It is possible that N contains more than one such node if u is found in different getName receiver sets in the PCG. Let

```
grant  Principal $\mathtt{c}_1$ "$\mathtt{t}_1$"  ...  Principal $\mathtt{c}_h$ "$\mathtt{t}_h$" {
         permission $\mathtt{p}_1$;  ...  permission $\mathtt{p}_k$;
};
```

Figure 2 Generic `Principal`-based `grant` Statement.

```
1:   for $l := 1$ to $h$ do
2:     if $\psi_{\mathtt{s},g}(\mathtt{c}_l) = \varnothing$ then
3:       return $\varnothing$
4:   for $l := 1$ to $h$ do
5:     boolean $found$ := false
6:     for each $x \in \psi_{\mathtt{s},g}(\mathtt{c}_l)$ do
7:       if $\mathtt{t}_l \in \upsilon_T(\mathtt{u}_{rx})$ then
8:         $found$ := true
9:         break
10:    if $found$ = false then
11:      return $\varnothing$
12:  return $\{p_1\} \sqcup \{p_2\} \sqcup \ldots \sqcup \{p_k\}$
```

Figure 3 Algorithm to Compute $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g)$.

T be the set of all the `String` constants that have flowed to the return values of all the `Principal` `getName` PCG nodes. For any $\mathtt{u} \in U$, the set $\upsilon_T(\mathtt{u})$ of all the elements of T that could have flowed to the return values of all the `getName` PCG nodes having `u` in the receiver set is identified.

Let g be any `grant` statement in $\mathcal{D}$, similar to those shown in Figure 1. In general, g will be of a form similar to the one in Figure 2, which asserts that a `Subject` `s` is granted `Permissions` $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_k \in P$ if `s` encapsulates at least h `Principal` objects with fully qualified class names $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h \in C$ and names $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_h \in T$, respectively, where the number of such `Principal` objects may be less than h if $\exists l_1, l_2 \in \{1, 2, \ldots, h\} : (\mathtt{c}_{l_1}, \mathtt{t}_{l_1}) = (\mathtt{c}_{l_2}, \mathtt{t}_{l_2})$.

Let $n \in D$ be the node examined above, with $\psi_D(n) = \{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j\} \in \mathcal{P}(S)$. For $r \in \{1, 2, \ldots, j\}$, ψ_S can be applied to $\mathtt{s}_r$, and the result will be of the form $\psi_S(\mathtt{s}_r) = \{\mathtt{u}_{r1}, \mathtt{u}_{r2}, \ldots, \mathtt{u}_{ri_r}\} \in \mathcal{P}(U)$. This allows computing $\Sigma(n)$, the subset of $\overline{P_\equiv}$ representing the subject-granted access rights under which the `doAs` or `doAsPrivileged` method represented by n, in the context specified by n, will be executed at run time. First of all, to detect whether or not the access rights granted by g apply to n, it is sufficient to proceed as follows. Let $\psi_{\mathtt{s},g} : \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h\} \to \mathcal{P}(\{1, 2, \ldots, i_r\})$ be defined by:

$$\psi_{\mathtt{s},g}(\mathtt{c}_l) = \{x \in \{1, 2, \ldots, i_r\} : \upsilon_C(\mathtt{u}_{rx}) = c_l\}, \forall l \in \{1, 2, \ldots, h\}$$

Computing $\psi_{\mathtt{s},g}$ can be accomplished with one iteration over the elements of $\psi_S(\mathtt{s}_r)$. It should be observed also that $\psi_{\mathtt{s},g}$ partitions the set of indices $\{1, 2, \ldots, i_r\}$ in equivalence classes. Specifically, two indices $x, y \in \{1, 2, \ldots, i_r\}$ are in the same equivalence class if and only if either $\exists l \in \{1, 2, \ldots, h\} : \upsilon_C(\mathtt{u}_{rx}) = \upsilon_C(\mathtt{u}_{ry}) = \mathtt{c}_l$ or $\upsilon_C(\mathtt{u}_{rx}), \upsilon_C(\mathtt{u}_{ry}) \notin \{\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h\}$.

Furthermore, let $\Sigma|_{S\times\mathcal{D}} : S \times \mathcal{D} \to \mathcal{P}(\overline{P_\equiv})$ be the function mapping a pair $(\mathtt{s}, g) \in S \times \mathcal{D}$ to the subset of $\overline{P_\equiv}$ representing the access rights granted by g to the `Principals` encapsulated in `s`. For any given $r \in \{1, 2, \ldots, j\}$, $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g)$ can be computed using the algorithm in Figure 3, where, in the notation introduced in Section 4.1.2, $p_1, p_2, \ldots, p_k$ represent the permission lattice elements corresponding to the `Permission` objects $\mathtt{p}_1, \mathtt{p}_2, \ldots, \mathtt{p}_k$, respectively.

Lines 1–3 of the algorithm presented in Figure 3 simply verify that for each `Principal` class name $\mathtt{c}_l$ specified in g, there is at least one `Principal` encapsulated in $\mathtt{s}_r$ whose class has that name. If this is not the case, then none of the `Permissions` listed in g can be applied to `s`, and $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g) = \varnothing$.

Lines 4–11 are used to verify that for every `Principal` class name $\mathtt{c}_l$ specified in g, there exists at least one `Principal`, among those encapsulated in $\mathtt{s}_r$, with name $\mathtt{t}_l$ and class name $\mathtt{c}_l$. If this is not the case, then, once again, none of the `Permissions` listed in g can be applied to `s`, and $\Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g) = \varnothing$. The presence of the **return** statements in Lines 3 and 11 guarantees that Line 12 is executed iff $\mathtt{s}_r$ encapsulates at least h `Principal` objects with fully qualified class names $\mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_h \in C$ and names $\mathtt{t}_1, \mathtt{t}_2, \ldots, \mathtt{t}_h \in T$, respectively, in which case $\mathtt{s}_r$ is granted all the `Permissions` listed in g, as desired.

If multiple `grant` statements in $\mathcal{D}$ apply to $\mathtt{s}_r$, the permission-set lattice elements from all those `grant` statements must be joined, and the resulting permission-set lattice element, $\Sigma|_S(\mathtt{s}_r)$, represents the access rights granted to $\mathtt{s}_r$, as follows:

$$\Sigma|_S(\mathtt{s}_r) = \bigsqcup_{g\in\mathcal{D}} \Sigma|_{S\times\mathcal{D}}(\mathtt{s}_r, g)$$

This defines function $\Sigma|_S : S \to \mathcal{P}(\overline{P_{\equiv}})$. SARA cannot statically determine which `Subject` allocation site in the parameter set $\psi_D(n) = \{\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j\}$ will be passed as a parameter to the `doAs` or `doAsPrivileged` method represented by n at run time. Therefore, when computing the subject-granted access rights under which the method call represented by n is executed, the safest albeit most conservative assumption is to apply the meet operation to all the permission-set lattice elements associated with the `Subject` allocation sites $\mathtt{s}_1, \mathtt{s}_2, \ldots, \mathtt{s}_j$ as $\Sigma(n) = \sqcap_{r=1}^{j} \Sigma(\mathtt{s}_r)$.

5.3 Fixed-point Iteration

When `doAs` or `doAsPrivileged` is invoked with a `Subject` parameter `s`, the `Permissions` granted to the `Principals` associated with `s` are added to the stack frame corresponding to `doAs` or `doAsPrivileged`, and to those corresponding to all the downstream calls. A static model of this aspect requires propagating $\Sigma|_S(\mathtt{s})$ to all the descendants of the `doAs` or `doAsPrivileged` node in the PCG.

For any node $n \in N$, let $\mathrm{Gen}_{\mathrm{S}}(n)$ and $\mathrm{Kill}_{\mathrm{S}}(n)$ indicate the subsets of $\overline{P_{\equiv}}$ whose elements correspond to the subject-granted access rights *generated* and *killed*, respectively, by node n. If $n \in D$, then $\mathrm{Gen}_{\mathrm{S}}(n) = \Sigma(n)$, otherwise $\mathrm{Gen}_{\mathrm{S}}(n) = \varnothing$. To compute $\mathrm{Kill}_{\mathrm{S}}(n)$, it should be noted that:

1. It is possible to call `doAs` multiple times within a thread, but only one `Subject` may be active at a time. If `doAs` is called with `Subject` parameter `s`, and later in the same thread there is another call to `doAs` with `Subject` parameter `t`, any subsequent authorization check will be performed with the authorizations granted to `t`, not those granted to `s`. The effect of the second call to `doAs` is to overwrite the authorizations granted to `s` until the second `doAs` call completes and returns to its caller. Similar considerations apply to `doAsPrivileged`.
2. In the absence of `Subjects` associated with a thread, a call to `doPrivileged` causes `checkPermission` to stop the stack walk at the stack frame corresponding to the method that called `doPrivileged`. The same principle applies when a `Subject` is present. If a `doPrivileged` call is made after invoking `doAs` or `doAsPrivileged` with `Subject` parameter `s`, `s` is not visible when an authorization check is performed by a `checkPermission` call. The effect of calling `doPrivileged` is to overwrite the authorizations granted to `s`.

Therefore, for any `doAs`, `doAsPrivileged`, or `doPrivileged` node $n \in N$, $\mathrm{Kill}_\mathrm{S}(n)$ is the universe $\overline{P_\equiv}$. For all other nodes $n \in N$, $\mathrm{Kill}_\mathrm{S}(n) = \varnothing$. This defines the two functions $\mathrm{Gen}_\mathrm{A}, \mathrm{Kill}_\mathrm{A} : N \to \mathcal{P}(\overline{P_\equiv})$.

SARA's *data-flow equation system*, for each node $n \in N$, is defined as follows:

$$\begin{aligned} \mathrm{Out}_\mathrm{S}(n) &= \mathrm{Gen}_\mathrm{S}(n) \sqcup (\mathrm{In}_\mathrm{S}(n) - \mathrm{Kill}_\mathrm{S}(n)) && (1)\\ \mathrm{In}_\mathrm{S}(n) &= \bigsqcap_{m \in \Gamma^-(n)} \mathrm{Out}_\mathrm{S}(m) && (2) \end{aligned}$$

where $\mathrm{Out}_\mathrm{S}(n)$ and $\mathrm{In}_\mathrm{S}(n)$ are the subsets of $\overline{P_\equiv}$ corresponding to the subject-granted access rights *propagated* from and *reaching* n, respectively, and $\Gamma^- : N \to \mathcal{P}(N)$ is the function that maps each node in the PCG to the set of its predecessors. The recursive computation of functions $\mathrm{In}_\mathrm{S}, \mathrm{Out}_\mathrm{S} : N \to \mathcal{P}(\overline{P_\equiv})$ performed by resolving System (1,2) converges to a fixed point with a worst-case complexity of $\mathcal{O}(|E| \cdot \mathcal{H}(\mathcal{P}(\overline{P_\equiv})))$ [17]. In particular, convergence of System (1,2) is guaranteed by the fact that $\forall n \in N$, the *data-flow function* $\beta_n : \mathcal{P}(\overline{P_\equiv}) \to \mathcal{P}(\overline{P_\equiv})$ that transforms $\mathrm{In}_\mathrm{S}(n)$ into $\mathrm{Out}_\mathrm{S}(n)$ (i) is defined on a complete lattice, $(\mathcal{P}(\overline{P_\equiv}), \sqcap, \sqcup)$, with finite height; and (ii) is monotonic with respect to the lattice's partial order, $\sqsupseteq$. The worst-case complexity is $\mathcal{O}(|E| \cdot \mathcal{H}(\mathcal{P}(\overline{P_\equiv})))$ because each edge of the PCG will need to be traversed $\mathcal{H}(\mathcal{P}(\overline{P_\equiv}))$ times in the worst-case scenario. Given that $\mathcal{H}(\mathcal{P}(\overline{P_\equiv})) \in \mathcal{O}(|P|)$, the worst-case complexity can be expressed as $\mathcal{O}(|E||P|)$.

The fact that the forward propagation of permission-set lattice elements is initialized *only* with the nodes in D significantly reduces the time complexity in typical cases, since large portions of the PCG are likely not to contain any `doAs` or `doAsPrivileged` call and will never be affected by the fixed-point iteration generated by System (1,2).

5.4 PCG Annotation

When the data-flow algorithm just described terminates, each node $n \in N$ can be labelled with the subset $\Sigma(n)$ of $\overline{P_\equiv}$ defined by $\Sigma(n) = \mathrm{Out}_\mathrm{S}(n)$, which overapproximates the subject-granted access rights under which the method represented by n, in the calling context specified by n, will be executed at run time. Figure 4 shows a PCG annotated with the values of function Σ. The PCG contains three calls to `doAs`, one to `doPrivileged`, and three to `checkPermission`. The three calls to `doAs` take parameter pairs $(\mathtt{s}, \mathtt{a})$, $(\mathtt{t}, \mathtt{b})$, and $(\mathtt{s}, \mathtt{c})$, respectively, where `s` and `t` are `Subject` instances, and `a`, `b`, and `c` are `PrivilegedAction` instances. The three calls to `checkPermission` take arguments `p`, `q`, and `r` respectively, all of type `Permission`. The PCG annotation shows how the call to `doPrivileged` eliminates from the `AccessControlContext` the access rights granted to `s`.

It may be desirable to compute the subject-granted access rights under which a method can be executed assuming all the calling contexts that are realizable in the program under analysis. A function $\mu_S : M \to \mathcal{P}(\overline{P_\equiv})$ can be defined that maps each method `n` reachable from an entry point of the program to the subset of $\overline{P_\equiv}$ representing the subject-granted access rights under which `n` can be executed. Specifically, $\mu_S(\mathtt{n})$ is defined as $\mu_S(\mathtt{n}) = \sqcap_{i=1}^{k} \Sigma(n_i)$, where $\{n_1, n_2, \ldots, n_k\} = \nu(\mathtt{n})$. The safest though most conservative approach is to compute the meet of the subsets $\Sigma(n_1), \Sigma(n_2), \ldots, \Sigma(n_k)$ of $\overline{P_\equiv}$, rather than their join. The reason is that, during the analysis, it is not known which calling context, among those of $n_1, n_2, \ldots, n_k$, `n` will be invoked with. As a result, it is not known which `Subject` will execute `n`.

5.5 Detection of PLP Violations

Unnecessary or redundant `doAs` and `doAsPrivileged` calls, as defined in Section 2.2, may lead to PLP violations and performance bottlenecks. As such, they should be removed.

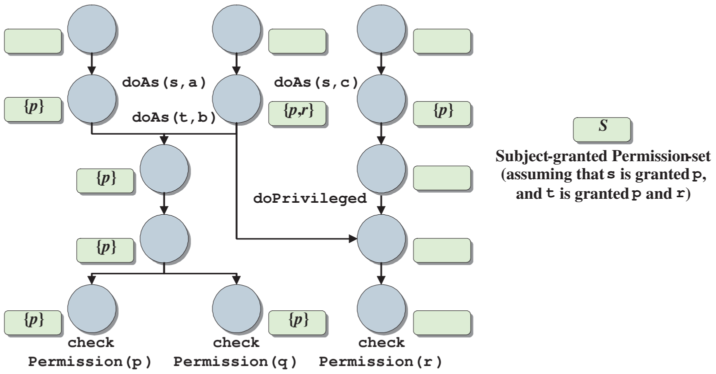

Figure 4 An Annotated PCG after SARA Completes.

SARA allows for detecting and flagging all such calls. To achieve this goal, SARA performs a basic access-rights analysis to detect the sets of permissions that each single method call will require at run time [9]. Any `doAs` or `doAsPrivileged` node that is mapped to ∅ represents a PLP violation.

6 Domain-specific Characteristics

This section describes the domain-specific characteristics implemented in the construction of a PCG in order to faithfully represent the subject-based permission model.

6.1 Modeling of Security-related Native Methods

Since SARA only analyzes Java bytecode, all *native* methods – those implemented in languages other than Java – would be incorrectly represented in the PCG as leaves, or nodes with no successors. However, many security analyses would not be sound without a model for those native methods that perform security-sensitive operations. Thus, the PCG is augmented with *automatically constructed* control- and data-flow-equivalent synthetic models [39] for a total of 162 native methods that are relevant to security analyses, such as:

- `Thread.start`, which executes a call to `Thread.run`
- The four forms of `AccessController.doPrivileged`
- `AccessController.getStackAccessControlContext`, which instantiates an object of type `AccessControlContext`, containing the `ProtectionDomains` of the classes whose methods are currently on the call stack

6.2 String Analysis

SARA is integrated with Path and Index-sensitive String Analysis (PISA) [34]. PISA can compute an over-approximation of the set of values that a variable of type `String` can take at run time. PISA is thus able to compute the set of `String` values that can flow to the target and action parameters of a `Permission` object. Without a string analysis capable of tracking `String` values and modeling operations on `String` objects, SARA would be unable to compute the target of the `Permission` object `p` in the following code snippet, and would conservatively have to overapproximate it as `"<<ALL_FILES>>"`, as explained in Section 3.2:

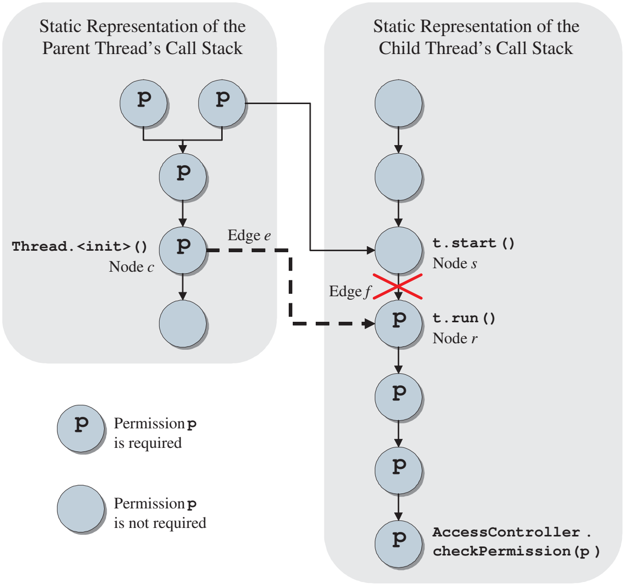

Figure 5 Domain-Specific PCG to Model Multi-threading.

```
String dir = "C:";
String file = "log.txt";
Permission p = new FilePermission(dir + File.separator + file, "read");
```

6.3 Modeling Multi-Threaded Code

In Java, to prevent confused-deputy attacks [20]. when access to a restricted resource is attempted from within a child thread, all the code in the child thread and 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 the PCG $G = (N, E)$ whose receiver is a `Thread` object. For each such node 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 $e = (c, r)$ is added to E. At the same time, edge $f = (s, r)$, where s represents the invocation of `start()` on `t`, is removed from E, as shown in Figure 5.

6.4 Extra Context for `Permission` Objects

The `Permission` parameter passed to `AccessController.checkPermission` is frequently instantiated by the `SecurityManager`. For example, when it is invoked by a library routine, the `SecurityManager`'s `checkWrite` method instantiates a `FilePermission` object and passes it to the `SecurityManager`'s `checkPermission` method, which finally passes it to the `checkPermission` method in the `AccessController` class. 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, SARA would represent them as if they were the same object. As a result, different calls to `AccessController.checkPermission` would appear in the PCG as one node, yielding overly conservative results, as shown in Figure 6. Other `SecurityManager` access-control methods,

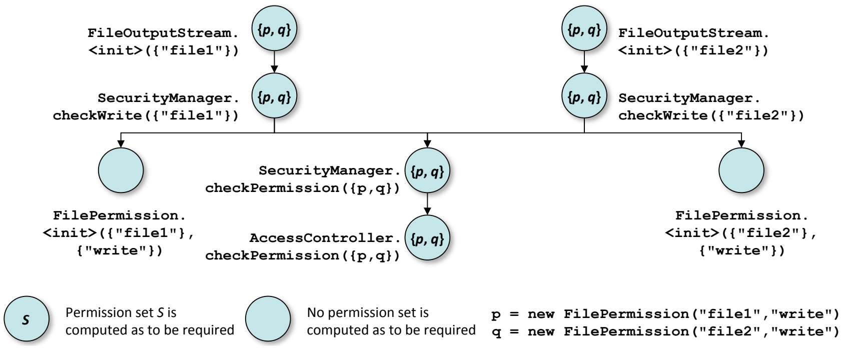

Figure 6 Conservativeness Introduced by the `SecurityManager`.

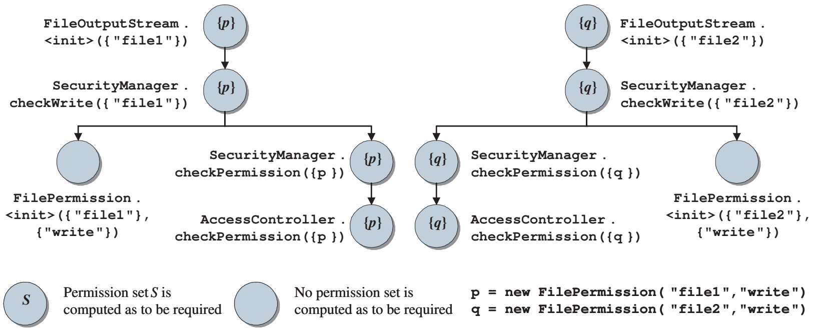

Figure 7 Extra Context for `Permission` Objects to Reduce Conservativeness.

such as `checkRead` and `checkConnect`, exhibit the same problem when represented in the PCG.

The solution is to distinguish `Permission` objects allocated in the `SecurityManager` access-control methods based not only on their types and allocation sites, but also on the nodes that contain those allocation sites. Therefore, if $m, n \in N$ are, for example, two `checkWrite` nodes in the PCG such that the parameters to the method calls they represent are the `Strings` `file1` and `file2`, respectively, the `FilePermission` allocated in m will be distinguished from the one allocated in n – even though both share the same type and allocation site – because $m \neq n$. Different `AccessController.checkPermission` calls resulting from `checkWrite` calls with different parameters will not appear in the PCG as one node. This allows a more precise propagation of permission-set lattice elements along different PCG paths, as shown in Figure 7.

To avoid building an unnecessarily large PCG, this specialization, which distinguishes `Permission` objects based on allocation sites with an additional level of calling-context sensitivity, is selectively applied only to `Permission` objects allocated in the `SecurityManager`.

6.5 Modeling `doAsPrivileged` Method Calls

In Java, the `AccessControlContext` passed to `doAsPrivileged` is obtained through a call to `getContext`. This method obtains the `AccessControlContext` associated with the current thread stack by calling `getStackAccessControlContext`. In order to faithfully propagate permission requirements along PCG paths containing a `doAsPrivileged` node d, it is sufficient to do the following:

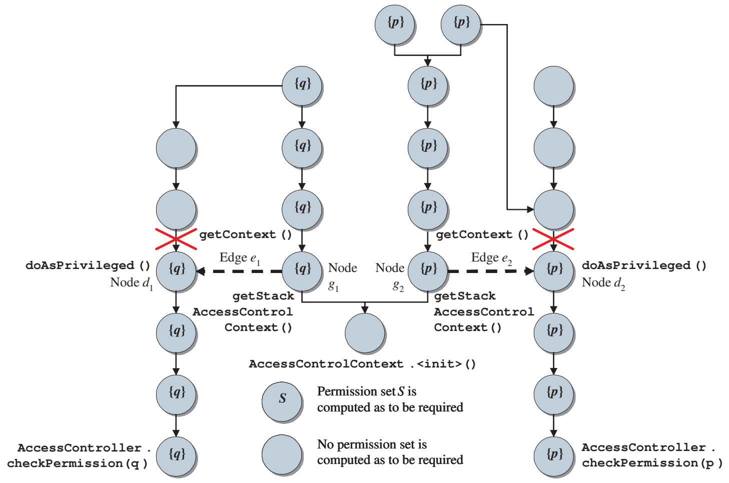
{p}
{p}
{q}
{p}
{q}
{p}
{q}
{p}
getContext ()
getContext ()
Edge e_1
Edge e_2
doAsPrivileged ()
Node d_1
{q}
{q}
Node g_1
Node g_2
{p}
{p}
doAsPrivileged ()
Node d_2
getStack AccessControl Context ()
getStack AccessControl Context ()
{q}
{p}
AccessControlContext .<init> ()
{q}
S
Permission set S is computed as to be required
{p}
No permission set is computed as to be required
AccessController . checkPermission (q)
{q}
{p}
AccessController . checkPermission (p)

Figure 8 Modeling `doAsPrivileged` Method Calls.

1. Identify the `AccessControlContext` allocation sites $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k$ that may flow to the `AccessControlContext` parameter of the `doAsPrivileged` call represented by d
2. Locate the `getStackAccessControlContext` PCG nodes $g_1, g_2, \ldots, g_k \in N$ allocating $\mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_k$, respectively
3. Remove d's preexisting predecessor edges from E
4. Add edges $e_1 = (g_1, d), e_2 = (g_2, d), \ldots, e_k = (g_k, d)$ to E

We observe that all the `AccessControlContext` instances obtained by calling `getContext` share the same allocation site, `a`, in method `getStackAccessControlContext`. Thus, in the analysis model, $\mathtt{a}_1 = \mathtt{a}_2 = \ldots = \mathtt{a}_k = \mathtt{a}$. Let d_1, d_2 be two PCG nodes representing calls to `doAsPrivileged`, and let $e_1 = (g, d_1), e_2 = (g, d_2)$ be two edges added to the PCG to model the propagation of permission requirements along the PCG paths. Since e_1, e_2 share the same tail node, g, when elements of $(\mathcal{P}(\overline{P_{\equiv}}), \sqcap, \sqcup)$ are propagated to g from d_1 and d_2, they are conservatively joined, yielding imprecise results. The solution consists of the following:

1. Adding one level of calling-context sensitivity to all the PCG nodes representing calls to `getStackAccessControlContext` and `getContext`
2. Distinguishing different `AccessControlContext` objects allocated through calls to method `getStackAccessControlContext` based not only on their allocation sites, but also on the nodes containing those allocation sites

Now, the PCG contains two `getStackAccessControlContext` nodes, g_1 and g_2, and two `getContext` nodes. Furthermore, SARA distinguishes between `AccessControlContext` objects $\mathtt{a}_1$ and $\mathtt{a}_2$, allocated in g_1 and g_2, respectively, as shown in Figure 8.

7 Evaluation

In this section, we describe an extensive experimental study.

7.1 Scope and Methodology

Subject-executed code is typically embedded in the business logic of enterprise application servers. This is because they service Web users, and at the same time they execute sensitive code and have access to critical organizational resources.

Therefore, our evaluation consisted of performing access-rights analysis of IBM WebSphere Application Server toward certifying it according to the CC EAL 4 standard. The application server contains over 2 MLOC, partitioned into 348 library components, which serve as our benchmarks. In order to pass the CC EAL 4 certification, it was necessary to correctly define access-control policies for each of these 348 libraries. For this to happen, each of the libraries was analyzed in isolation. As for the client code used for the analysis of each library, we utilized an existing testing harness in the form of a thorough unit-test suite, written by the application-server developers, which created a comprehensive client environment.

7.2 Experimental Results

Table 1 presents general information about the 348 benchmark libraries:

- 105 of the libraries were not originally associated with any policy at all (0 permissions granted to code and 0 to subjects).
- 141 had an associated access-control policy that only accounted for permissions granted to code (1,021), and ignored the permissions to be granted to subjects executing the code (0).
- The remaining 102 libraries had policies that accounted for both code (743) and subject permissions (132). For those 102 components, the subject-related permissions had been defined by developers based solely on manual code inspection and unit testing.

SARA achieved the following results:

Table 1 General Information about the Application Server Benchmarks.

Annotations	None	Code	Code & Subjects	Total
Library Count	105	141	102	348
`doAs` **Calls**	325	531	301	1,157
`doAsPriviledged` **Calls**	72	81	79	232
Original Code Permisssions	0	1,021	743	1,764
Original Subject Permissions	0	0	132	132
Permissions via SARA	143	191	151	485

- For the first 105 components, which required access-control policies for both the code and the subjects executing it, SARA computed 143 subject permissions.
- For the second set, of 141 components, SARA determined 191 subject permissions. Note that these permissions are subsequently subtracted from the sets of permissions granted to the code in order to prevent PLP violations.
- For the third and final category, SARA verified the policies that were constructed manually, and modified the permissions granted to the code in certain places in order to prevent permissions from being granted to both subject and code.

Table 1 also reports the number of `doAs` and `doAsPrivileged` calls. In total, there were 1,157 `doAs` and 232 `doAsPrivileged` calls, confirming that subject-executed code cannot be ignored when defining the access-control policy for an enterprise application server.

Table 2 highlights the specific violations detected by SARA on the 348 benchmarks:

- SARA detected 263 PLP violations, occurring when the same rights were granted to both code and subjects. This type of violation is only applicable to the third category of libraries – those that came with a developer-defined access-control policy accounting for both code and subjects. For the other two categories, no policy had been defined for the subjects executing the code, and so the number of PLP violations for those libaries was 0.

Table 2 Access-control Violations Detected by SARA.

Annotations	None	Code	Code & Subjects	Total
PLP Violations	0	0	263	263
Insufficient Policies	105	72	42	219
Unnecessary Calls	5	2	7	14
Redundant Calls	7	3	5	15

- SARA also detected 219 violations due to insufficient policies. This means that the policies coming with the library do not grant enough permissions to the subjects executing the code. It is interesting to note that the set of permissions that SARA computed for the subjects executing the code in the third library category was neither a superset nor a subset of the permissions originally computed by the developers. In some cases, permissions had been redundantly granted to both code and subjects, and so those permissions had to be removed. In other cases, the permissions were not listed at all, which would have caused authorization failures at run time (as confirmed during testing), and those permissions had to be added.

From a precision and performance point of view, we observe the following:

- The results produced by SARA were validated by dozens of developers, who confirmed them via a large set of dynamic test cases. None of the issues that were dynamically found was missed by SARA. False negatives may arise from not modeling native code and reflective calls, but SARA is equipped with (i) automatically generated analyzable synthetic artifacts for all the security-related methods in the application server and the underlying Java Runtime, and (ii) a mechanism for reflection resolution based on type cast information [26]. These two components significantly mitigate the risk of false negatives.
- SARA exhibited a high signal-to-noise ratio with 5% of the total number of reported violations being false positives.
- SARA was able to scale to, and analyze successfully, the entire application server, taking on average 103.45 seconds per library.

7.3 Discussion

Overall, SARA was able to detect a total of 511 access-rights flaws across the set of 348 benchmarks with an average scanning time of 103 seconds. We attribute SARA's ability to analyze each benchmark in under 2 minutes on average to its relatively inexpensive context-sensitivity policy. We could not compare SARA against other tools, unfortunately, as existing static-analysis algorithms can only identify the permissions required by a given component, but not those required by the subjects that will execute the code [23].

According to a thorough evaluation, which included a large set of dynamic test cases, SARA exhibited 0 false negatives. Having 0 false negatives was an important requirement for SARA because in access-rights analysis, a false negative corresponds to a missing permission, which at run time can cause unexpected authorization failures. Although we cannot claim that SARA is sound, our efforts to reduce false negatives by synthetically modeling native methods and disambiguating reflective calls through type cast information allow us to say that SARA is a *soundy* analysis [25].

SARA's false-positive rate was only 5%. This was established via careful scrutinization of the results, one by one, by the team developing the application server. A false positive corresponds to an unnecessary permission, which constitutes a PLP violation, and so it was pertinent to identify all false positives before deploying the server.

PLP violations and vulnerabilities due to insufficient policies were corrected by simply modifying the policy files associated with the libraries. However, the only way to correct the remaining two vulnerabilities detected by SARA – unnecessary and redundant `doAs` and `doAsPrivileged` calls – was to change the source code of those libraries and remove such calls, which constitute both a PLP violation and a performance bottleneck. Our experience when communicating these vulnerabilities to the developers was that the interactions between the `doAs`, `doAsPrivileged`, `doPrivileged` and `checkPermission` APIs are very complicated and hard to explain to non-security experts, which is what caused these vulnerabilities to be present in the code in the first place. Having a tool for automatic detection of such unnecessary or redundant calls proved to be a crucial instrument to improve the quality of the code and teach developers how to use these APIs correctly in future code.

Based on these results, SARA proved to be accurate and useful. In our evaluation, it detected a large number of flaws, and scaled to all the libraries comprising the application server (2 MLOC in total), with an average analysis time of 103 seconds per library.

8 Related Work

There is no work on static analysis of subject-based authorization, particularly with regard to subject-granted rights analysis. Most of the work in the area of program analysis for access control has focused on computing permission requirements based on stack inspection, eliminating or minimizing redundant authorization tests, and defining alternatives to the current approach.

Pottier *et al.* [31] extend and formalize Wallach's security passing style [38] via type theory using a λ-calculus, called λ_{sec}. However, their work focuses only on basic authorization issues and is unable to perform incomplete-program analyses [32].

Jensen *et al.* [21] focus on proving that code is secure with respect to a global security policy. Their model uses operational semantics to prove the properties, via 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. Bartoletti *et al.* [6] are interested in optimizing performance of run-time authorization tests by eliminating redundant tests and relocating others as needed. Similarly, Banerjee and Naumann [5] 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. These analyses are limited to a single thread and require the whole program.

Felten *et al.* have studied a number of security problems related to mobile code [36, 12, 38, 10, 37, 11]. In particular, they present a formalization of stack introspection. An authorization optimization technique, called *security passing style*, encodes the security state of an application while the application is executing [38]. The goal is to optimize authorization performance. Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [13] 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. Conversely, this paper examines the authorization issue from the perspective of an existing system containing authorization test points.

Koved *et al.* [23] describe an algorithm and an actual implementation for correctly identifying the authorizations needed by a Java program, but do not deal with subject-executed code.

9 Conclusion and Future Work

In this paper, we have investigated the problem of performing comprehensive access-rights analysis. Such an analysis must account for subjects, which none of the existing permission analyses supports. We have validated the significance of this dimension experimentally via SARA, the first static permission analysis featuring subject sensitivity and policy correction/synthesis. The SARA algorithm features novel mechanisms to (i) represent the program, (ii) recover the permission hierarchy, and (iii) associate `Subjects` with `Permissions`. In the future we plan to enable SARA as an IDE tool. This requires real-time performance and incremental analysis capabilities, which we are currently developing.

References

1 M. Abadi and C. Fournet. Access Control Based on Execution History. In *NDSS*, 2003.

2 O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *ECOOP*, 1995.

3 P. Anderson, T. Reps, and T. Teitelbaum. Design and Implementation of a Fine-Grained Software Inspection Tool. *TSE*, 29(8), 2003.

4 K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *CCS*, 2012.

5 A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *JFP*, 15(2), 2005.

6 M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. In *ConCoord*, volume 54, 2001.

7 F. Besson, T. Blanc, C. Fournet, and A. D. Gordon. From Stack Inspection to Access Control: A Security Analysis for Libraries. In *CSFW*, 2004.

8 P. Centonze. *An Algebra for Access Control*. PhD thesis, New York University, Polytechnic School of Engineering, Brooklyn, NY, USA, 2008.

9 P. Centonze, R. J. Flynn, and M. Pistoia. Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-control Policies. In *ACSAC*, 2007.

10 D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM Conference on Computer and Communications Security (CCS)*, 1997.

11 D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and Beyond. In *S&P*, 1996.

12 R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, 1999.

13 Ú. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *S&P*, 2000.

14 A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *CCS*, 2011.

15 C. Fournet and A. D. Gordon. Stack Inspection: Theory and Variants. In *POPL*, 2002.

16 E. Geay, M. Pistoia, T. Tateishi, B. G. Ryder, and J. Dolby. Modular String-sensitive Permission Analysis with Demand-driven Precision. In *ICSE*, 2009.

17 G. Grätzer. *General Lattice Theory*. Birkhäuser, second edition, 2003.

18 D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *TOPLAS*, 23(6), 2001.

19 S. Gulwani and G. C. Necula. Path-sensitive Analysis for Linear Arithmetic and Uninterpreted Functions. In *SAS*, 2004.

20 N. Hardy. The Confused Deputy (or Why Capabilities Might Have Been Invented). *OSR*, 22(4), 1988.

21 T. P. Jensen, D. Le Métayer, and T. Thorn. Verification of Control Flow Based Security Properties. In *S&P*, 1999.

22 G. A. Kildall. A Unified Approach to Global Program Optimization. In *POPL*, 1973.

23 L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *OOPSLA*, 2002.

24 C. Lai, L. Gong, L. Koved, A. J. Nadalin, and R. Schemers. User Authentication and Authorization in the Java™ Platform. In *ACSAC*, 1999.

25 B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In Defense of Soundiness: A Manifesto. *CACM*, 58(2), 2015.

26 B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *APLAS*, 2005.

27 S. S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers, Inc., 1997.

28 G. Naumovich. A Conservative Algorithm for Computing the Flow of Permissions in Java Programs. In *ISSTA*, 2002.

29 M. Pistoia. *A Unified Mathematical Model for Stack- and Role-Based Authorization Systems.* PhD thesis, New York University, Polytechnic School of Engineering, Brooklyn, NY, USA, 2005.

30 M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond Stack Inspection: A Unified Access-control and Information-flow Security Model. In *S&P*, 2007.

31 F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *ESOP*, 2001.

32 B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *CC*, 2003. Invited Paper.

33 J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, 1975.

34 T. Tateishi, M. Pistoia, and O. Tripp. Path- and Index-sensitive String Analysis Based on Monadic Second-order Logic. *TOSEM*, 22(4), 2013.

35 O. Tripp, M. Pistoia, P. Cousot, R. Cousot, and S. Guarnieri. Andromeda: Accurate and Scalable Security Analysis of Web Applications. In *FASE*, 2013.

36 D. S. Wallach. *A New Approach to Mobile-Code Security.* PhD thesis, Princeton University, Princeton, NJ, USA, 1999.

37 D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *SOSP*, 1997.

38 D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *S&P*, 1998.

39 X. Zhang, L. Koved, M. Pistoia, S. Weber, T. Jaeger, G. Marceau, and L. Zeng. The Case for Analysis Preserving Language Transformation. In *ISSTA*, 2006.