Combining Static and Dynamic Analysis for Automatic Identification of Precise Access-Control Policies

Paolina Centonze
IBM Watson Research Center
Hawthorne, New York, USA
paolina@us.ibm.com

Robert J. Flynn
Polytechnic University
Brooklyn, New York, USA
flynn@poly.edu

Marco Pistoia
IBM Watson Research Center
Hawthorne, New York, USA
pistoia@us.ibm.com

Abstract

Given a large component-based program, it may be very complex to identify an optimal access-control policy, allowing the program to execute with no authorization failures and no violations of the Principle of Least Privilege. This paper presents a novel combination of static and dynamic analysis for automatic determination of precise access-control policies for programs that will be executed on Stack-Based Access Control systems, such as Java and the Common Language Runtime (CLR). The static analysis soundly models the execution of the program taking into account native methods, reflection, and multi-threaded code. The dynamic analysis interactively refines the potentially conservative results of the static analysis, with no need for writing or generating test cases or for restarting the system if an authorization failure occurs during testing, and no risk of corrupting the underlying system on which the analysis is performed.

We implemented the analysis framework presented by this paper in an analysis tool for Java programs, called Access-Control Explorer (ACE). ACE allows for automatic, safe, and precise identification of access-right requirements and library-code locations that should be made privilege-asserting to prevent client code from requiring unnecessary access rights. This paper presents experimental results obtained on large production-level applications.

1 Introduction

Defining the security policy of a program is a challenging activity, which becomes particularly difficult when the program is large and complex and consists of multiple components.[1] Ideally, the security policy should be just sufficient for the program to run without authorization failures. Any access right unnecessarily granted to the program or its users is a violation of the Principle of Least Privilege [31].

Modern component-based software systems, such as Java [27] and Microsoft .NET Common Language Runtime (CLR) [13], have adopted a form of authorization checking called Stack-Based Access Control (SBAC); when access to a restricted resource is attempted, a *stack inspection* ensures that all the code on the call stack is sufficiently authorized.

In SBAC systems, when library code is developed, an important decision that needs to be made is whether the portion of library code requesting an access right should be made *privilege-asserting*. This causes the run-time stack inspection to stop at the library level. As a result, client code is exempted from the requirement to exhibit the access right requested by the library. For example, a library providing socket connections to its clients may have been programmed to log the network operations to a file. While it is reasonable to impose that the client code be granted the necessary `SocketPermission`, it is not reasonable to impose that any client code be granted the `FilePermission` to write to the log file. If client code had to be granted that `FilePermission`, a violation of the Principle of Least Privilege would arise; a malicious client could misuse that permission to log false information. Clearly, privilege-asserting code should be inserted cautiously because when client code is above privilege-asserting code on the stack, its access rights are not verified.

Traditionally, security policies are defined using a combination of source-code inspection and testing. However, for large and complex programs, manual code inspections may be impractical, tedious, time consuming, and error prone. This type of analysis may even be infeasible if source code is unavailable, which is the case if the program was machine generated or written by a third party. On the other hand, testing requires writing or generating test cases, exe-

[1] As an example, the Eclipse community [9] is currently undergoing a very expensive process to enable Java security on the Rich Client Platform (RCP) [10]. This requires determining the authorization and privilege-assertion requirements of the entire RCP. Part of the static analysis work described in this paper has been used to facilitate that goal.

cuting them, and logging all the authorization failures. For each failure, it is necessary to identify the missing access right, and decide whether that access right should be granted or not. Since every authorization failure can cause a program crash, for each failure occurring during testing it may be necessary to grant the missing authorization and restart the program. Furthermore, testing is *unsound* since some paths of execution, along with the access rights required to execute those paths, may remain undiscovered until the code is deployed. Another problem is that the system on which the tests are performed can become corrupted if the program being tested is malicious or behaves incorrectly.

An interesting alternative is static analysis. With static analysis, a model of the execution of the program is created, and the access rights necessary to execute the program are obtained from the model of the execution rather than from the execution itself. In general, given a program, the execution model built by the static analyzer is a *conservative* overapproximation of any run-time program execution. While it guarantees soundness, this also means that a static analyzer may include in the model infeasible events (*false alarms*). Thus, blindly granting to a program all the authorization requirements reported by a static analyzer may easily generate violations of the Principle of Least Privilege.

While theoretically sound, in practice a static analyzer may be unsound for several reasons:

- **Multi-language Code.** Today's large and complex applications are likely to comprise components written in different languages. For example, a Java program is almost always going to trigger the execution of several *native methods*, written in C and executed directly on the underlying operating system [33]. A static analyzer purely for Java will not be able to model the control and data flow generated by those methods. As a result, the analysis will be incomplete and unsound, failing to report actual authorization requirements.
- **Reflection.** *Reflection* is a mechanism that enables code to dynamically discover and manipulate fields and methods of loaded classes [33]. Modeling reflection through static analysis is potentially unsound since the type of the objects obtained through reflection is often known and available only at run time [21].
- **Callbacks.** A *library callback* is a method invocation whose receiver is a parameter passed to the library by a client. Libraries are typically analyzed as *incomplete programs* [30] since clients are likely to be available only at run time. For example, let `foo` be a library method taking a parameter of type `T` and invoking method `bar` on that parameter. A client program can create an object `t` of type `T` and pass it to `foo` by calling `foo(t)`. As a result, `foo` will execute callback `t.bar`. If neither `T` nor `T.bar` are final, a client program could subclass `T` and override method `bar`, causing `bar` to arbitrary access restricted resources that are unknown at static-analysis time.

In essence, neither dynamic analysis nor static analysis can independently guarantee the identification of a policy sufficient to execute a program and yet not too permissive. This paper presents a novel combination of static and dynamic analysis extending previous work that was completely based on static analysis [20, 26, 37]. The contribution of this paper is to achieve precise identification of authorization and privilege-assertion requirements in SBAC systems. The main characteristics of the static analyzer that contribute to its precision are the following:

1. A scalable but precise context- sensitivity level [30]
2. An automatic model generator for native methods based on language transformation and sound with respect to the security analysis described in this paper
3. A sound models for reflection and callbacks
4. A complete model of the stack inspection mechanism, which includes privilege-asserting and multi-threaded code

The dynamic analysis framework described in this paper is completely novel and includes the following components:

1. A sandboxing environment that protects the underlying system on which the analysis is performed from malicious or bad-performing programs
2. A configurable framework allowing the dynamic analyzer to automatically detect authorization and privilege-assertion requirements in the presence of different security subsystems and to automatically form the security policy of a component-based program
3. An interactive system for authorization decisions, allowing the user to precisely identify the code location responsible for each authorization and privilege-assertion requirement
4. A mechanism for immediate identification of the security side effects of any method invocation
5. An automatic mechanism for policy minimization that eliminates redundant policy requirements

The static and dynamic analysis described in this paper has been implemented in a tool for Java programs called Access-Control Explorer (ACE). This paper presents the results obtained by running ACE on production-level code.

2 Motivating Examples

This section motivates the importance of combining static and dynamic analysis for precise identification of authorization and privilege-assertion requirements.

2.1 Authorization Analysis

In Java, access rights are implemented as objects of class `Permission` or one of its subclasses. A

`Permission` object is completely characterized by the fully-qualified name of the `Permission` subclass, along with the `String` parameters representing the *target* and the *mode* of access, as in `java.io.FilePermission "C:/log.txt", "read"`. Stack inspection is performed by function `checkPermission` in Java and `Demand` in the CLR.

```
import java.io.*;
import java.net.*;

public class SecurityLibrary {
  private String dir = "C:";
  private String file = "log.txt";
  private String name;
  public FileInputStream readLogFile1()
      throws Exception {
    return new FileInputStream
        (dir + File.separator + file);
  }
  public FileInputStream readLogFile2()
      throws Exception {
    file = "audit.txt";
    return new FileInputStream
        (dir + File.separator + file);
  }
  public FileInputStream readLogFile3
        (String fileName)throws Exception {
    return new FileInputStream(fileName);
  }
  public String getSystemProperty1() {
    if (Math.sqrt(4) > 0)
      return System.getProperty("user.home");
    else
      return System.getProperty("user.dir");
  }
  private void changeName() {
    name = "user.dir";
  }
  public String getSystemProperty2() {
    name = "user.name";
    changeName();
    return System.getProperty(name);
  }
  public void changeClassLoader()
      throws Exception {
    Thread t = Thread.currentThread();
    URL[] urlArray = new URL[] {
        new URL("http://abc.xyz.com") };
    URLClassLoader loader =
        new URLClassLoader(urlArray);
    t.setContextClassLoader(loader);
  }
}
```

Figure 1. SecurityLibrary.java

Figure 1 shows the example of a library requesting authorizations. The static analysis framework presented in this paper can help a developer and system administrator detect a superset of the library's authorization requirements but, as observed in Section 1, a purely static analysis approach is limited due to conservativeness and unsoundness. For the example of Figure 1, such limitations would manifest themselves as follows:

- The permissions required by `readLogFile1` and `readLogFile2` cannot be easily evaluated with just static analysis; even a static analysis engine that keeps track of `String` constants is unable to evaluate the expression `dir + File.separator + file`, unless it performs `String` computations.
- For method `getSystemProperty1`, a *path insensitive* [30] static analyzer is unable to compute that `java.util.PropertyPermission "user.home", "read"` is the only permission required to execute `getSystemProperty1`. Reporting a `java.util.PropertyPermission "user.dir", "read"` requirement is a false alarm.
- To account for multi-threaded applications, static analyzers are typically *interprocedurally flow insensitive* [30] and do not perform interprocedural strong updates [22]. As a consequence, although `java.util.PropertyPermission "user.dir", "read"` is the only permission required to execute `getSystemProperty2`, an interprocedurally flow-insensitive static analyzer will also report an unnecessary requirement for `java.util.PropertyPermission "user.name", "read"`.
- A static analyzer for Java will typically not detect the `java.lang.RuntimePermission "setClassLoader"` requirement of method `changeClassLoader` since the receiver of the `setContextClassLoader` method call, `t`, is obtained as the return value of the call to method `Thread.currentThread`, which is native and, as such, not modeled by the analyzer. This *false negative* leads to unsound results.

A purely dynamic analysis approach is often limited too for the unsoundness generated by the absence of a complete suite of test cases. For example:

- The `java.io.FilePermission` needed by `readLogFile3` has the special `<<ALL FILES>>` target, indicating the entire file system. The reason for this broad requirement is that the exact file to be read depends from the parameter passed to `readLogFile3` by the client code. While a static analyzer can correctly model this requirement, a dynamic analyzer will show a different target every time `readLogFile3` is invoked with a different parameter.
- Using dynamic analysis to detect the authorization requirement for `readLogFile1` leads to test-case-dependent analysis results. In fact, a test case invoking `readLogFile1` *before* `readLogFile2` shows a

`java.io.FilePermission "C:/log.txt", "read"` requirement. Conversely, a test case invoking `readLogFile1` *after* `readLogFile2` shows a requirement for `java.io.FilePermission "C:/audit.txt", "read"`, the reason being that `readLogFile2`, as a side effect, changes the value of `file` from `log.txt` to `audit.txt`. This shows that any dynamic analysis for authorization-requirement identification needs to take side effects into account.

While it is always possible to improve a static or dynamic analyzer to identify authorization requirements with higher precision, static and dynamic analysis are always going to present limitations leading to imprecise policy definitions. The solution presented in this paper eliminates such imprecisions by combining the two approaches.

2.2 Privilege-Asserting-Code Analysis

A Java developer can make a portion of library code privilege-asserting by wrapping it into a call to `doPrivileged`. Method `createSocket` in Figure 2 opens a socket on behalf of its clients, but upon doing so, it logs the operation to a file. To prevent its clients (which will be on the stack) from requiring the `FilePermission` to write to the log file, the logging operation is wrapped into a call to `doPrivileged`. For large and complex libraries, it is very difficult to detect manually which portions of code should be made privilege-asserting, and to identify the sets of permissions that library code implicitly grants to its clients by virtue of calling `doPrivileged`. This paper shows how a synergy of static and dynamic analysis can be used to make this process both automatic and precise.

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

public class SecurityLibrary2 {
  private final String logFileName = "C:\\log.txt";
  public Socket createSocket(String host, int port)
      throws Exception {
    FileOutputStream f = (FileOutputStream)
        AccessController.doPrivileged
        (new PrivilegedExceptionAction() {
      public Object run() throws Exception {
        return new FileOutputStream(logFileName);
      }
    });
    PrintStream ps = new PrintStream(f);
    ps.print("Socket opened...");
    return new Socket(host, port);
  }
}
```

Figure 2. SecurityLibrary2.java

In the CLR, privilege-asserting code is achieved by calling the `Assert` function, which can be seen as a parameterized version of `doPrivileged`. In fact, unlike `doPrivileged`, `Assert` takes an `IPermission` object as a parameter. The effect is that the stack inspection is truncated only if the `IPermission` object being checked is the same as, or weaker than, the one passed to `Assert`. From this point of view, `Assert` is more secure than `doPrivileged`, which truncates the stack inspection indiscriminately. The static and dynamic analysis algorithms described in this paper can be used to make `doPrivileged` equivalent to `Assert` by precisely controlling the `Permission` checks from which client code is shielded and by only recommending insertions of privilege-asserting code as close to the authorization checks as possible.

3 Static Analyzer

ACE effectively models the run-time stack inspection mechanism using a *context-sensitive* [30] static analyzer. The context-sensitivity policy adopted by ACE disambiguates each method invocation based on the *invocation context*—the receiver and parameters used in that invocation. Moreover, ACE disambiguates objects based on their allocation sites, as in Andersen's analysis [2]; each allocation site represents an equivalence class of dynamically allocated objects. Unlike other, less precise approaches [16], these context- and object-sensitivity policies allow distinguishing calls to `checkPermission` based on the `Permission` object being checked. Given the bytecode of a Java program and library, along with all the libraries upon which the input program or library depends, the analysis engine produces a call graph $G = (N, E)$. Each node $n \in N$ represents the invocation of a method m along with the invocation context. Each edge $e = (n_1, n_2) \in E$ models the invocation of the method m_2 in the context represented by n_2, performed by the method m_1 in the context represented by n_1.

Statically computing authorization and privilege-assertion requirements can be done by appropriately modeling the dynamic stack-inspection mechanism. The static model of stack inspection is based on the observation that each path in the call graph represents a potential run-time call stack. Since stack inspection dynamically propagates authorization *requirements* backwards across a call stack starting from a `checkPermission` *method*, the static model propagates authorization-requirement *abstractions* backwards in the call graph starting from a `checkPermission` *node*. Authorization requirements are statically represented as `Permission` allocation sites. If P is the set of the `Permission` allocation sites in a given program, its powerset, 2^P, is naturally endowed with

the structure of a lattice, $\mathcal{L} := (2^P, \cup, \cap)$, where $\cup$ and $\cap$ are the usual set-union and set-intersection operators [15]. The static authorization analysis can then be cast to a standard data-flow problem [1], in which elements of 2^P are propagated backwards in G and set unions are performed at the merge points.

More precisely, if $n \in N$ is a `checkPermission` node, then n represents a call to the `checkPermission` method, which takes a `Permission` object as a parameter. Due to its inherent conservativeness (primarily due to path insensitivity and interprocedural flow insensitivity, as explained in Section 2.1), for each `checkPermission` node n the static analysis performed by ACE may overapproximate the `Permission` parameter for n as a set $Q(n)$ containing more than one `Permission` allocation site. We initialize the static analysis by defining function Gen : $N \rightarrow 2^P$ as follows: If n is a `checkPermission` node, then Gen$(n) := Q(n)$, else Gen$(n) := \varnothing$. The same considerations can be applied to a CLR program by simply replacing `checkPermission` with `Demand`.

To represent the stack-inspection mechanism precisely, it is necessary to model the property that privilege-asserting code, when encountered on a run-time stack, stops the stack walk. This behavior can be statically represented by function Kill : $N \rightarrow 2^P$, which is defined as follows: If n is a `doPrivileged` node, then Kill$(n) := P$, else Kill$(n) := \varnothing$.

In the CLR, as we observed, privilege-asserting code is parameterized with permissions. Therefore, for the CLR, Kill : $N \rightarrow 2^P$ is defined in a different way, as follows: If n is an `Assert` node and $A(n) \subseteq P$ is the set of `IPermission` allocation sites that could flow to the parameter of `Assert` in the context of n, then Kill$(n) := A(n)$, else Kill$(n) := \varnothing$.

The data-flow equations modeling the stack-inspection mechanism in Java or the CLR are defined as follows:

$$\text{Out}(n) := (\text{In}(n) \setminus \text{Kill}(n)) \cup \text{Gen}(n) \quad (1)$$

$$\text{In}(n) := \bigcup_{m \in \Gamma^+(n)} \text{Out}(m) \quad (2)$$

for each $n \in N$, where $\Gamma^+ : N \rightarrow 2^N$ is the *successor function* in G, defined by $\Gamma^+(n) := \{n' \in N \mid (n, n') \in E\}$. Clearly, $\mathcal{L}$ has finite height, $|P|$, and the data-flow functions from $\mathcal{L}$ in $\mathcal{L}$ induced by Equations (1) and (2) on each node $n \in N$ are monotonic with respect to the partial order $\supseteq$ of $\mathcal{L}$. Therefore, Tarski's Theorem guarantees that the recursive computation of the solutions of Equations (1) and (2) converges to a fix point in $\mathcal{O}(|E||P|)$ time [15].

According to the stack-inspection semantics, the code performing a privilege-asserting call needs to be granted the permissions it shields [28]. This behavior can be modeled with a one-step, non-recursive backward propagation of permission requirements, to be performed upon convergence of the recursive computation of the solutions of Equations (1) and (2), as described by Equation (3):

$$\text{In}(n) := \text{In}(n) \cup \bigcup_{d \in \Gamma^+(n) \cap D} \text{In}(d) \quad (3)$$

for each $n \in N$, where $D \subseteq N$ is the set of the call-graph nodes representing calls to `doPrivileged` in Java and `Assert` in the CLR. Solving Equation (3) has a time complexity of $\mathcal{O}(|E|)$.

Upon convergence, In(n) overapproximates the access rights necessary to perform the invocation represented by n at run time. It should be observed that every single method can be represented by multiple nodes in the call graph, depending on the different calling contexts with which that method can be invoked. If method m is represented by nodes $n_1, n_2, \ldots, n_k \in N$, then ACE overapproximates the permissions necessary to invoke m at run time as the set

$$\Pi(m) := \bigcup_{i=1}^{k} \text{In}(n_i) \quad (4)$$

While statically modeling the stack-inspection mechanism, ACE allows identifying, for each permission q, which code is responsible for making the call to the security-sensitive function requiring q. This allows a developer to identify statically, without executing the program, the portions of the program under analysis that are candidate for being made privilege-asserting.

As observed in Section 1, static analysis, which is supposed to be sound, can become unsound due to improper modeling of callbacks, multi-threaded code, native methods, and reflection. The following sections explain how ACE ensures static-analysis soundness.

3.1 Callbacks

When analyzing a library as an incomplete program, ACE detects all the library entry points that take parameters of non-final types or that have fields of non-final types. Every time a method is invoked on one of those parameters or fields, if that method is itself non-final, ACE flags that invocation as potentially requiring the special `AllPermission` authorization requirement since the implementation of that method is unknown during the static analysis.

3.2 Multi-threaded Code

To prevent the freshly created stack of a child thread from having potentially more permissions than the stack of the parent thread, SBAC systems attach the parent thread's stack to the child thread's stack. When

checkPermission traverses the stack, all the callers in both the child and the parent threads will have to show possession of the permission being checked. ACE models this run-time behavior statically by adding an edge from every constructor node instantiating a Thread t to the node where t.start is invoked.

3.3 Native Methods

Without a sound model for native methods, any security analysis performed by ACE would be unsound since several security-sensitive functions trigger the invocation of native methods, such as doPrivileged and the implementation of Thread.start. For this reason, ACE statically represents 161 native methods (those that have been established to affect authorization) with control- and data-flow-equivalent stubs. While in the past these stubs needed to be manually constructed, the most interesting aspect is that now the process of creating these stubs has been automated with a tool [37] that converts C code into Java code that is control- and data-flow-equivalent to its C counterpart. The Java bytecode representing the automatically translated native methods is then seamlessly integrated in the analysis scope. The process of automatic translation prevents errors in the stub creation and facilitates the creation of new stubs when new versions of the Java libraries become available.

3.4 Reflection

ACE models reflection by identifying the type to which the result of a call to newInstance is cast and by using that type for disambiguation. When the type cannot be inferred, java.lang.Object is conservatively assumed. The dynamic analysis performed by ACE can be used to refine conservative results.

4 Dynamic Analyzer

A major novelty of ACE is its dynamic-analysis component and the static- and dynamic-analysis synergy. For every method m in the program under analysis, the static analyzer of ACE reports not only the set $\Pi(m)$, overapproximating the set of permissions required to invoke m, as defined by Equation (4), but also, for each permission in $\Pi(m)$, the execution path that may lead to that authorization requirement at run time. This information is then used to dynamically recreate the invocation of m in the context reported by the static analysis.

The dynamic analyzer uses reflection to load classes, create objects, and invoke methods on those classes and objects. This way, the process of creating a test case is completely automated. A major concern when performing dynamic analysis of untrusted code is the potential ability of that code to compromise the integrity of the underlying system. To prevent this, ACE acts as a layer between the program p under analysis and the underlying system. Upon startup, ACE enforces a SecurityManager and confines p into a *sandbox* in which p is initially granted no access rights. Any attempt by p to escape from its sandbox is intercepted by the SecurityManager, which raises an AccessControlException. However, rather than letting the system stop working, ACE catches the Exception and interactively prompts the user to make a security decision about it. In particular, ACE extracts the following information from the AccessControlException:

1. Precise information about the attempted security-sensitive operation (for example, opening file passwords.txt in read mode)
2. The fully-qualified name of the Permission class guarding the attempted security-sensitive action, along with the explicit String parameters (target and modes), if any, of the Permission object (for example, java.io.FilePermission "passwords.txt", "read")
3. All the callers currently on the stack at the time the security-sensitive operation was attempted
4. The Java ARchive (JAR) file of the class attempting to escape from the sandbox, along with the Uniform Resource Locator (URL) of that JAR file and the certificates of the principals who digitally signed that JAR file
5. Precise information (source file name, class name, method signature, and line number) on the portion of code requesting the privilege, *which is also the portion of code in p that should be made privilege-asserting to prevent client code from requiring unnecessary authorizations*
6. The set of permissions already granted to the code, retrievable from the ProtectionDomain of the class attempting the security-sensitive action [14]

The user can judge, based on this information whether p should be granted the right to access the security-sensitive resource. In that case, the ACE layer automatically updates and refreshes the security policy, without the need for restarting the program or manually editing the policy. The invocation of m is automatically repeated and, if no other permission is required, it will now succeed; otherwise, the user will be prompted with a new security decision.

4.1 Elimination of Static False Alarms

The process described is iterated for every method m that, according to the static analysis, may attempt a security-sensitive action. It m does not cause any AccessControlException when ACE executes it *in*

the context reported by the static analysis, that means that the static analysis had reported a false alarm, and that the corresponding access right should `not` be granted to p. This process enables ACE to eliminate statically-detected false alarms and to build a policy just sufficient to execute the methods that were dynamically analyzed. Furthermore, the dynamic analyzer of ACE allows filtering out those static false alarms caused by the inability of a static analyzer to perform `String` computations, as observed in Section 1.

4.2 Configurable Security Subsystem

In the Java type system, any object implementing a class that extends `java.lang.SecurityManager` can be a valid `SecurityManager`. ACE enables the enforcement of `SecurityManager` objects of different subtypes. This is an important feature because while the standard `SecurityManager` enforces the stack inspection mechanism by calling `checkPermission`, a different implementation of the SecurityManager may enforce access control in a different way [24, 9]. This feature is especially desirable since it has been proved that the stack inspection mechanism is unsound [25]. A static analyzer that was built to model the stack inspection mechanism typically does not have the flexibility to automatically model a different access-control mechanism. Therefore, the dynamic analysis component of ACE becomes a useful feature when the behavior of the run-time `SecurityManager` deviates from the standard one.

4.3 Security Side Effects

Unlike other dynamic analyzers, ACE has a framework for dynamically detecting the security side effects of each method invocation, such as the side effect caused by `readLogFile2` on the permission requirements of `readLogFile1` in Figure 1, as explained in Section 2. Specifically, for any policy change, ACE retests all the paths of execution previously traversed, and detects whether new authorizations are necessary. By dynamically analyzing the results of the previously executed static analysis, ACE achieves complete coverage of the methods requesting permissions and identifies a policy compliant with the Principle of Least Privilege.

4.4 Privilege-Asserting-Code Analysis

As highlighted in Point 5 above, the dynamic-analysis component of ACE can be used to identify the portions of library code candidate to be made privileged to prevent client code from requiring unnecessary access rights. The recommended location is *optimal* in the sense that it is always the closest to the authorization check.

4.5 Dynamic Policy Minimization

Another novel contribution of ACE is its ability to minimize access-control policies. As an example, `java.io.FilePermission "dir/*", "write"` is *stronger* than `java.io.FilePermission "dir/log.txt", "write"`. If a program p requires both these permissions, it is sufficient to list in the policy for p only the stronger one, which makes the policy easier to maintain. ACE performs minimization automatically by instantiating all the `Permission` objects detected, and by then executing the `implies` method of each `Permission` object against all the other `Permission` objects. If `p` and `q` are `Permission` objects required by p, and `p.implies(q)` returns `true`, then only `p` needs to be added to the policy for p. Unlike previous, unsafe policy-minimization approaches [20], ACE prevents potentially-malicious code embedded into `implies` methods from harming the system by executing `implies` only under the system `SecurityManager`.

5 Experimental Results

This section presents the experimental results on production level code. The applications analyzed, listed in both Tables 1 and 2, are all from SourceForge [32], except `Crypto`, which is obtained by combining in one project all the cryptography and Transport Layer Security (TLS) examples from two Java security books [28, 27].

Table 1 describes the general characteristics of the applications along with statistics from executing the static analyzer of ACE for authorization- and privilege-assertion-requirement detection. For each application, the size of the application itself and the sum of the sizes of its supporting libraries (including the Java core libraries) are displayed. The time taken by the static analyzer is reported. These results were obtained on an IBM ThinkPad T23 with 1 GB of RAM and a processor of 1.3 GHz. The operating system was Microsoft Windows XP SP2. The static analyzer was itself implemented as a Java program running on a Sun Microsystems' Java Runtime Environment (JRE) V1.4.2.

Name	Size (MB)		Static Analysis	
	Applications	Libraries	Time	Memory
`Aamfetch`	0.106	36.921	489 sec	251 MB
`Crypto`	0.760	37.724	321 sec	324 MB
`Ganymed`	0.336	36.921	567 sec	374 MB
`Gnu`	1.867	52.408	872 sec	622 MB
`JavaSign`	0.174	38.441	639 sec	675 MB
`JPassword`	0.678	36.921	634 sec	397 MB

Table 1. Static-Analysis Statistics

This section does not detail the time required to perform the dynamic analysis part since that is an interactive pro-

Name	Authorization Requirements					Privilege-Assertion Requirements				
	Static	Dynamic Refinement				Static	Dynamic Refinement			
		False	Actual				False	Actual		
			Imprecise	Precise	Minimized			Imprecise	Precise	Total
Aamfetch	49	5	40	4	6	28	20	5	3	8
Crypto	24	7	2	15	12	78	14	40	24	64
Ganymed	45	6	38	1	3	37	9	16	12	28
Gnu	51	15	17	19	11	187	43	78	66	144
JavaSign	19	4	10	5	6	32	0	25	7	32
JPassword	27	3	19	5	6	113	37	55	21	76

Table 2. ACE Authorization and Privileged-Code Requirement Findings

cess, and its duration strictly depends on the experience of the analyst in using ACE. In our experience, it takes in average 1 minute to evaluate each statically-detected authorization requirement.

Table 2 focuses on the security results. Specifically, for each application, it shows the following:

1. The number s of authorization requirements detected using only *static analysis*.
2. The number f of *false alarms* among those authorization requirements. f is detected by refining the static analysis results using the dynamic analyzer of ACE.
3. The number a of the *actual* requirements found—meaning that these requirements are real. This category includes the following:
   (a) The number i of the actual, but *imprecise*, authorization requirements (for example, an actual `FilePermission` requirement for which the static analysis was not able to disambiguate the file name due to its inability to perform `String` computations, as observed in Section 1).
   (b) The number p of the actual and *precise* authorization requirements.
   (c) The number m of the actual permission requirements after policy minimization, as discussed in Section 4.5.

   Notice that $s = f+i+p$ since an authorization requirement found by the static analysis can only be false or actual, and if it is actual it can only be imprecise or precise. Also, in general, $m \leq s$. Table 2 shows the usefulness of this automatic combined approach since it allows detecting authorization requirements (though in a conservative manner) and then using the dynamic analyzer to significantly reduce the requirements found.

Next, Table 2 shows the results obtained by using ACE to detect the privilege-assertion requirements, as follows:

1. The number s of privilege-assertion requirements detected using only *static analysis*.
2. The number f of *false alarms* among those privilege-assertion requirements.
3. The number a of the *actual* requirements found. This category includes:
   (a) The number i of the actual privilege-assertion requirements that are, however, *imprecise* since the `Permission` objects associated with the privilege-assertion requirement could not be uniquely disambiguated.
   (b) The number p of the actual and *precise* privilege-assertion requirements.
   (c) The *total* number t of the actual privilege-assertion requirements.

   In this case, $s = f + i + p$ and also $t = i + p$ since, unlike authorization requirements, privilege-assertion requirements cannot be minimized.

As shown, the combined static and dynamic analysis approach dramatically improves a process that, if done manually, would become impractical. With the exhaustive model for native methods included in the static analyzer of ACE, no false negative was observed.

6 Conclusion and Future Work

This paper described a combination of static and dynamic analysis for precise identification of access-control policies. The algorithms presented in this paper have been implemented as part of a tool called Access-Control Explorer (ACE), which has been used on a number of production-level applications.

Future areas of investigation will involve improving the precision of the static analyzer of ACE to reduce the number of false alarms it generates and simplify the dynamic analysis. Currently, the static analyzer of ACE employs a very expensive context-sensitivity policy, which adds precision indiscriminately, even where it is not needed, thereby reducing scalability. It would be desirable to automatically increase the precision of the static analyzer when modeling security-sensitive calls, while reducing the precision of the analyzer where security is not needed. In particular, the context-sensitivity policy used by ACE disambiguates different calls to the same method based on the receiver and parameters. If a method is static and takes no parameters, then all the calls to it will be represented by only one node in the call graph, which creates a pollution point when model-

ing stack inspection through that node. A research direction will involve a more precise and selective context-sensitivity policy that eliminates this problem. Furthermore, integrating static authorization analysis with an analysis for string computations will help in eliminating many false alarms.

7 Related Work

The need for integration of static and dynamic analysis has been known for a long time. Orso, et al. [23] have combined static and dynamic analysis for networking modeling, discovery, and analysis. Our work distinguishes itself from that work because it precisely addresses the need for a precise authorization and privilege-assertion analysis. Ernst [12] discusses synergies and similarities of static and dynamic analysis, and how these two approaches can be integrated to achieve optimal results in a wide range of problems.

Felten, et al. study a number of security problems related to mobile code [34, 8, 36, 5, 35, 7, 6] and present a formalization of stack introspection that examines authorization based on the principals currently active in a thread stack at run time (*security state*). An authorization optimization technique, called *Security-Passing Style* (SPS), encodes the security state of an application while the application is executing [36]. Each method is modified to pass a security token as part of each invocation. The token represents an encoding of the security state at each stack frame, as well as the result of any authorization test encountered. With this mechanism, the SPS explores subgraphs of the comparable invocation graph and discovers the associated security states and authorizations. Their goal is to optimize the authorization performance, while one of the purposes of this paper is 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. Pottier, et al. [29] extend and formalize the SPS via type theory using a λ-calculus, called λ_{sec}. However, their work does not address incomplete-program analysis [30]. Jensen, et al. [18] 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. Bartoletti, et al. [3] 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.

Rather than analyzing security policies as embodied by existing code, Erlingsson and Schneider [11] 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. Through static analysis, the mathematical framework of this paper can be used to discover how the security policy needs to be modified or updated to enable the code to execute.

Hajime and Forrest [17] present a dynamic permission analysis. Their solution is not interactive, does not deal with security side effects, is not integrated with a static-analysis solution, and does not deal with privilege-asserting code, and does not prevent a malicious program from harming the underlying system; all the permissions requested by a program are automatically granted, with the risk of compromising the underlying system.

Privileged code has historic roots in the 1970's. The Digitial Equipment Corporation (DEC) Virtual Address eXtension/Virtual Memory System (VAX/VMS) operating system had a feature similar to privilege assertion, called *privileged images*. Those images were similar to UNIX `setuid` programs [4], except that they ran in the same process as all the user's other unprivileged programs. As such, they were considerably easier to attack than UNIX `setuid` programs because they lacked the usual separate process/separate address space protections, as shown by Koegel, et al. [19]. Koved, et al. [20] and Pistoia, et al. [26] automate static security analysis for Java authorization and privilege assertion. Zhang, et al. [37] enhance those works with an automated native-code model generator to reduce the number of false negatives. This paper extends those works with a more precise static analysis and a novel dynamic analysis for elimination of false alarms.

8 Acknowledgments

The authors would like to thank the reviewers of the Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007) for their precious suggestions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, Jan. 1986.

[2] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994.

[3] M. Bartoletti, P. Degano, and G. L. Ferrari. Static Analysis for Stack Inspection. In *Proceedings of International Workshop on Concurrency and Coordination, Electronic Notes in Theoretical Computer Science*, volume 54, Amsterdam, The Netherlands, 2001. Elsevier.

[4] H. Chen, D. Wagner, and D. Dean. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, Berkeley, CA, USA, August 2002. USENIX Association.

[5] D. Dean. The Security of Static Typing with Dynamic Linking. In *Proceedings of the 4th ACM conference on Computer and Communications Security*, pages 18–27, Zurich, Switzerland, 1997. ACM Press.

[6] D. Dean, E. W. Felten, and D. S. Wallach. Java Security: From HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 190–200, Silver Spring, MD, USA, 1996. IEEE Computer Society Press.

[7] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz. Java Security: Web Browsers and Beyond. Technical Report 566-597, Princeton University, Princeton, NJ, USA, February 1997.

[8] R. D. Dean. *Formal Aspects of Mobile Code Security*. PhD thesis, Princeton University, Princeton, NJ, USA, Jan. 1999.

[9] Eclipse Project, `http://www.eclipse.org`.

[10] Equinox Project, `http://www.eclipse.org/equinox`.

[11] U. Erlingsson and F. B. Schneider. IRM Enforcement of Java Stack Inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, CA, USA, May 2000. IEEE Computer Society.

[12] M. D. Ernst. Static and Dynamic Analysis: Synergy and Duality. In *Proceedings of the Program Analysis for Software Tools and Engineering (PASTE 2004) Workshop*, pages 24–27, June 2004.

[13] A. Freeman and A. Jones. *Programming .NET Security*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, June 2003.

[14] L. Gong, G. Ellison, and M. Dageforde. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*. Addison-Wesley, Reading, MA, USA, second edition, May 2003.

[15] G. Grätzer. *General Lattice Theory*. Birkhäuser, Boston, MA, USA, second edition, January 2003.

[16] D. Grove and C. Chambers. A Framework for Call Graph Construction Algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, November 2001.

[17] H. Inoue and S. Forrest. Inferring Java Security Policies Through Dynamic Sandboxing. In *International Conference on Programming Languages and Compilers*, Las Vegas, NE, USA, June 2005.

[18] T. P. Jensen, D. L. Métayer, and T. Thorn. Verification of Control Flow Based Security Properties. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 89–103, Oakland, CA, USA, May 1999.

[19] J. F. Koegel, R. M. Koegel, Z. Li, and D. T. Miruke. A Security Analysis of VAX VMS. In *ACM '85: Proceedings of the 1985 ACM Annual Conference on the Range of Computing: Mid-80's Perspective*, pages 381–386. ACM Press, 1985.

[20] L. Koved, M. Pistoia, and A. Kershenbaum. Access Rights Analysis for Java. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–372, Seattle, WA, USA, November 2002. ACM Press.

[21] B. Livshits, J. Whaley, and M. S. Lam. Reflection Analysis for Java. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems*, pages 139–160, Nov. 2005.

[22] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, June 1997.

[23] A. Orso, M. J. Harrold, and G. Vigna. MASSA: Mobile Agents Security through Static/Dynamic Analysis. In *Proceedings of the First ICSE Workshop on Software Engineering and Mobility (WSEM 2001)*, Toronto, Canada, April 2001.

[24] OSGi Specification, `http://www.osgi.org`.

[25] M. Pistoia, A. Banerjee, and D. A. Naumann. Beyond Stack Inspection: A Unified Access Control and Information Flow Security Model. In *28th IEEE Symposium on Security and Privacy*, pages 149–163, Oakland, CA, USA, May 2007.

[26] M. Pistoia, R. J. Flynn, L. Koved, and V. C. Sreedhar. Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, pages 362–386, Glasgow, Scotland, UK, July 2005. Springer-Verlag.

[27] M. Pistoia, N. Nagaratnam, L. Koved, and A. Nadalin. *Enterprise Java Security*. Addison-Wesley, Reading, MA, USA, February 2004.

[28] M. Pistoia, D. Reller, D. Gupta, M. Nagnur, and A. K. Ramani. *Java 2 Network Security*. Prentice Hall PTR, Upper Saddle River, NJ, USA, second edition, August 1999.

[29] F. Pottier, C. Skalka, and S. F. Smith. A Systematic Approach to Static Access Control. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 30–45. Springer-Verlag, 2001.

[30] B. G. Ryder. Dimensions of Precision in Reference Analysis of Object-Oriented Languages. In *Proceedings of the 12th International Conference on Compiler Construction*, pages 126–137, Warsaw, Poland, April 2003. Invited Paper.

[31] J. H. Saltzer and M. D. Schroeder. The Protection of Information in Computer Systems. In *Proceedings of the IEEE*, volume 63, pages 1278–1308, Sept. 1975.

[32] SourceForge.net, `http://www.sourceforge.net`.

[33] Sun Microsystems, Java™ Technology, `http://java.sun.com`.

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

[35] D. S. Wallach, D. Balfanz, D. Dean, and E. W. Felten. Extensible Security Architectures for Java. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 116–128, Saint Malo, France, 1997. ACM Press.

[36] D. S. Wallach and E. W. Felten. Understanding Java Stack Inspection. In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 52–63, Oakland, CA, USA, May 1998.

[37] X. Zhang, L. Koved, M. Pistoia, S. Weber, T. Jaeger, G. Marceau, and L. Zeng. The Case for Analysis Preserving Language Transformation. In *Proceedings of the ACM SIGSOFT 2006 International Symposium on Software Testing and Analysis (ISSTA)*. ACM Press, July 2006.