Modeling and Analyzing Security Requirements for Java

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

In partial fulfilment of the requirements for the Degree of

Bachelor of Science

by

James Cialdea Joseph Politz

on

December 2008

Sponsored by:

Ronald Monzillo, Sun Microsystems

Advised by:

Professor Daniel J. Dougherty, Computer Science Department

Abstract

Defining access control policies for Java applications can be complex. This project investigates methods to simplify the process. First, static analysis was used to help determine permission requirements. The results of this analysis led towards the development of a new view of Java access control, called the Action-Centric Model. The Action-Centric Model provides a higher-level view of Java access control in order to simplify the process of creating policy and facilitate analysis of existing policies.

Contents

1 Introduction and Problem Description **1**
- 1.1 Introduction to Access Control in Java . . . 1
  - 1.1.1 How Java Performs Security Checks . . . 1
- 1.2 The Challenges in Working With Permissions . . . 3
- 1.3 Related Work and Motivation . . . 4
  - 1.3.1 Static Analysis and Permissions . . . 4
  - 1.3.2 A Simpler Model . . . 5
- 1.4 Contributions . . . 6

2 Static Analysis **8**
- 2.1 Methods . . . 8
  - 2.1.1 Algorithm . . . 8
  - 2.1.2 Implementation . . . 10
  - 2.1.3 Limitations of Static Analysis Data . . . 12
- 2.2 Results . . . 12
  - 2.2.1 Successes . . . 12
  - 2.2.2 Limitations . . . 13

3 Action-Centric Model **16**
- 3.1 Methods . . . 16
  - 3.1.1 Design . . . 16
  - 3.1.2 XML Specification of Definitions . . . 22
  - 3.1.3 NetBeans Plug-In . . . 22
- 3.2 Results . . . 23
  - 3.2.1 Successes . . . 23
  - 3.2.2 Limitations . . . 24

4 Evaluation and Future Work **28**
- 4.1 Permission Identification . . . 28
- 4.2 Action-Centric Model . . . 28
  - 4.2.1 Soundness of Model Translation . . . 28
- 4.3 Future Work . . . 29
  - 4.3.1 Action-Centric Model Extensions . . . 29
  - 4.3.2 NetBeans Plug-In Extensions . . . 29

Appendix A - Static Analysis Example **31**

Appendix B - NetBeans Plug-In Guide **35**

Appendix C - XML Specification of File System Actions **46**

List of Figures

1.1 Control flow of a typical security check 7

2.1 The static analysis data structure 15

3.1 Example of filled socket accept action 17
3.2 Example of SocketPermissionRule 17
3.3 Example of socket accept action definition 17
3.4 The relationship between resources, actions, and rules 18
3.5 Formal expression of parameter applications 20
3.6 Creation of filled rules from filled actions 20
3.7 Converting rules to actions, step 1 21
3.8 Converting rules to actions, step 2 26
3.9 XML Specification of Socket Accept 27

4.1 Plug-In startup screen 36
4.2 Plug-In main screen 37
4.3 Adding a simple action 38
4.4 Plug-In main screen with actions and rules 39
4.5 Grouping demonstration 40
4.6 Diff view 41
4.7 Unmatched rules demonstration - all matched 42
4.8 Unmatched rules demonstration - some unmatched 43
4.9 Policy Import 44
4.10 Policy Export 45

Chapter 1

Introduction and Problem Description

As our daily lives become more dependent upon computer applications, the risks associated with computer systems becoming compromised or acting improperly build. Every day hundreds of millions of people trust computer applications to process their personal data, execute potentially life changing transactions, provide important communication channels, and organize their lives. All this must be accomplished with high reliability and without violating the users' privacy.

Access control is one method to help achieve these requirements. Access control provides ways to place limits on the privileges of an application, typically at runtime, using a set of authorizations. Limiting applications' access to sensitive data or computing resources effectively limits the risk of compromised applications and unexpected access to resources. For example, most applications do not access to personal files such as customer billing information, but a compromised system may attempt to acquire such data. Access control can eliminate this threat by denying access to any data outside the scope of what the application requires. Access control can also be utilized to help reduce downtime by eliminating the possibility for applications to damage or overuse critical resources. For example, important system files could be protected by access control to avoid unexpected access, or important socket port ranges could be protected and reserved for specific applications.

1.1 Introduction to Access Control in Java

This section focuses on the design and usage of Java's ability to provide fine-grained access control to features such as file access, socket communication, program configuration, and security configuration.[7] A discussion of challenges in using Java security follows the practical introduction. There are many security features dealing with Java's class loaders that help assure that the code is from a trusted source as well as features for certificate and key management that will not be discussed here. These features are important to Java's overall security design, but are not important to this project.

1.1.1 How Java Performs Security Checks

Overview

When a Java application is launched, it receives a *security policy*. Usually, the policy is read in from a file when the Java Virtual Machine (JVM) starts or provided by the application server hosting the application. When a Java class is loaded, it is assigned to a *protection domain* based

on its class-path and code source (which JAR it came from). Protection domains are groups of classes which share a set of *permissions*. A permission defines a specific privledge that should be granted to a protection domain. The security policy defines what permissions each protection domain has been granted. The Java Security Manager is responsible for determining if a specific permission is granted to a piece of code by examining the minimal permission set of all protection domains in the *call stack*. The call stack is a list that tracks the path of method invocations to the current point of execution. The minimal permission set is computed by traversing the call stack to determine all the protection domains that are crossed, then taking the intersection of the permission sets of all these protection domains. The minimal set must be computed, otherwise unprivileged code could be allowed to perform privileged actions by simply calling through another protection domain. During a security check, if the required permission is granted, the program continues normal execution. If the permission is not granted, a SecurityException is thrown, which can be handled or cause the application to terminate.[7]

Detail

Before performing a security check, a permission object that represents the desired authorization must be constructed. This is done by creating an instance of Java's Permission class. The Permission class is an abstract class used to define a type of authorization. The implies(Permission) method of the Permission class is used to determine if a given permission is granted by another. The implies(Permission) method can be used to infer other permissions that should logically be granted. For instance, a permission that grants read access to the file space named "/usr/*" would imply a permission with access to "/usr/local/*". Commonly used extensions of Permission include:

- **java.io.FilePermission** - Represents a file read and/or write authorization
- **java.net.SocketPermission** - Represents authorization to connect, accept, and/or work with sockets
- **java.lang.RuntimePermision** - Represents authorization to do any one of several Java Virtual Machine related actions
- **java.lang.ReflectPermission** - Represents authorization to do any one of several Java Object Reflection related actions
- **java.security.AllPermission** - Causes all permissions to be granted, regardless of what abilities they grant

There are many other extensions of Permission included in Java that are not mentioned here. It is not uncommon for an application to have its own domain-specific extensions of Permission. All standard permissions are parameterized by two Strings. The first is called the *target*. It generally contains information about what specific objects the permission is talking about. The second is called the *action*. It contains information about what sorts of operations the permission is allowing on the target.

Once the proper Permission is constructed, it is passed to the Security Manager where it is checked against the current policy. This process is demonstrated in Figure 1.1. The Security Manager is usually accessed through the System.getSecurityManager() method. To invoke a security check, either the SecurityManager.checkPermission(Permission) method or the SecurityManager.checkPermission(Permission, AccessControlContext) method is called. The AccessControlContext encapsulates the context under which the security check should be carried out. Most notably, the AccessControlContext is responsible for determining the minimal permission set in the case that the call stack crosses more than one Protection Domain. If an AccessControlContext is not provided to checkPermission(), the Security Manager delegates to the Access Controller, where

the appropriate AccessControlContext is created. Once the AccessControlContext is determined, the Permission is passed to that AccessControlContext's checkPermission method. The AccessControlContext then checks to be sure each Protection Domain has been granted the Permission in question or one that implies it. The ProtectionDomain objects each call out to the system's current Policy object to make the individual checks. If any Protection Domain has not been granted the desired permission, an AccessControlException is thrown, causing a SecurityException to be thrown from the SecurityManager. If the security check passes, the program continues to run normally.

doPrivileged There is a special construct defined in Java called a PrivilegedAction. The AccessController.doPrivileged() method takes in a PrivilegedAction and executes its run() method with special security semantics. In particular, the ProtectionDomains in the call stack above the call to doPrivileged() are *not* inspected to see if they have the required Permissions. This allows a certain block of code to "take responsibility" for its actions, and encapsulate some particular secure computation. It could be used in library code or specific application code to provide a service needed by other applications that should not be granted certain permissions.

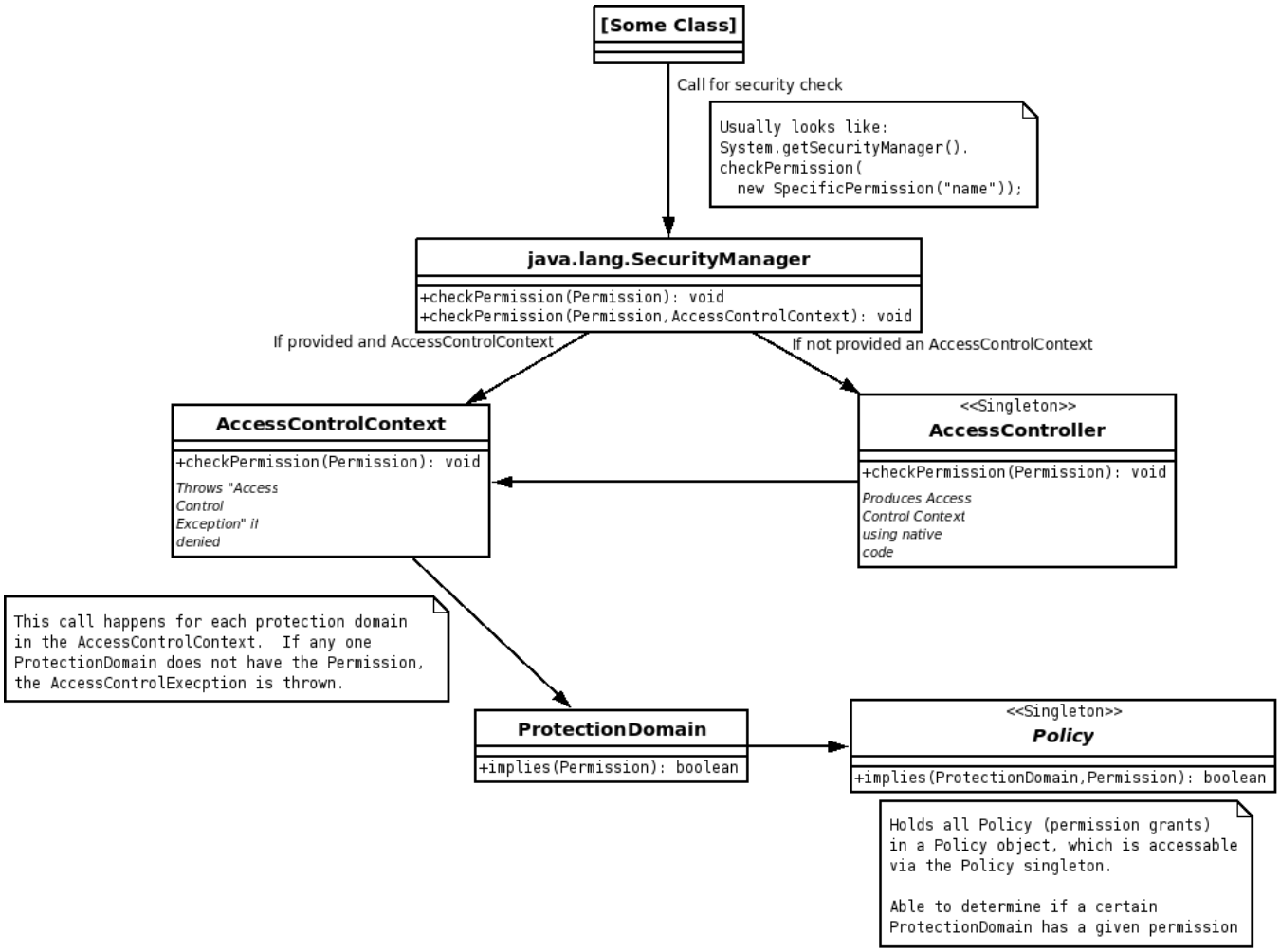

Figure 1.1: Control flow of a typical security check

The Need to Determine Permission Requirements Accurately

The access control requirements of an application are represented by a policy containing a set of permissions. In order to ensure that a given application operates as expected, the proper set of permissions must be identified. An application becomes a security hole if it is provided with permissions that grant access to resources that the application would not normally use. If this application were to be compromised, these additional resources would be placed in jeopardy needlessly. If the over-privileged application contains coding errors, these additional resources could be overused or damaged accidentally. An application becomes unusable if it is not provided with permissions that grant access to the resources it needs to operate. When this application is run, it will not function properly and possibly terminate unexpectedly. The desired policy strikes a balance between these two situations by granting the application exactly the permissions it needs to operate and no others.

1.2 The Challenges in Working With Permissions

Many developers find it hard to define security policies for several reasons.

Determining the permission requirements is not always straightforward. Some method calls and JVM actions result in unexpected permission requirements. Without a very deep understanding of how the JVM acts and a complete knowledge of Java's security systems, it is likely that application developers and maintainers will not understand where all permission requirements actually come from nor will they understand exactly what they mean. Without this knowledge, these key people may not understand the security risks and implications associated with certain sets of permissions.

There is no fail safe way of detecting permission requirements before run time. The exact behavior of any program is affected by runtime data which may not be available before the program runs. Because of the undecidable nature of the problem, it is not possible to predict the parameters of all permissions. Without complete knowledge of all parameters, it is likely that the program will either not be granted enough permissions, causing it to fail security checks unexpectedly, or will be granted too many permissions, opening security holes.

The language of permissions is verbose. Permission classes must be specified by fully qualified names (FQNs) which are long and tedious for application developers and maintainers to type out. The policy file syntax is also verbose, requiring significant amounts of repeated data entry, putting additional strain on the developer and increasing the chance of error. Permission parameter syntax is also complex in many cases, making it prone to human error. Multiple pieces of data are often grouped into one string parameter, making permission parameters hard to understand and prone to syntactic errors, which are hard to troubleshoot.

Sets of permissions often appear together. We expect that certain tasks will require more than one permission to be granted. For many common tasks, it is possible to specify exactly what permissions would be required in a standardized way. Java policy creation currently does not take advantage of this. Instead, each permission must be individually defined, even if it is part of a larger logical set.

1.3 Related Work and Motivation

1.3.1 Static Analysis and Permissions

Static analysis is one method that has met with some significant success in security analysis for Java [4] [5] [10] [11]. Some studies that include static analysis have focused on locating behavior of Java programs that might be conducted in an insecure manner[11]. Others have focused on features of the Java language in particular, specifically the way Java handles security checks through Permissions and stack inspection [4] [10] [5]. The SWORD4J application is of particular intrest to this project because it uses static analysis to infer the permission requirements of Java programs [2]. The description of the algorithm written by Koved and used in SWORD4J was the starting point for our investigation into static analysis[10].

One result that has been investigated through the use of static analysis is the inference of Permissions that are checked within a program [10], [9]. The goal of such a procedure is to produce the most accurate possible representation of the set of Permissions required by a program. This information can be used to create policy files at the level of Permissions and begin to understand the security requirements of the program. We consider how this information can be used and how other models can provide a better understanding of security.

Permissions as a Low-Level View of Security

Inferring what permissions are required by detecting permission checks in method calls can result in a specification of a set of Permissions required by a program. It should be clear that permission objects themselves are simply instances of a particular Java class. They have meaning relative to the Java security system, including SecurityManagers and AccessControllers. This understanding is important when one considers a set of "required" Permissions. Saying Permissions are required is to say: "If a runtime policy is created for this ProtectionDomain that does not contain at least this set of Permissions with these arguments, then a SecurityException will be thrown." This result can be quite useful, in terms of getting existing code to run under a SecurityManager. Depending on the understanding available about what each of these Permissions means, this could say very much or very little about the actual capabilities the program is being granted.

The identification of Permissions is an important first step in understanding the security requirements and behavior of Java programs. It provides a view of security at the level it is implemented. However, the way Permissions are implemented in the Java security framework is really just program *behavior*. A application developer or maintainer uses his or her own knowledge of Java access control to add the proper Permissions to their policies. Information about the Permission requirements of a program can provide a starting point for a more high level analysis which describes the security and resource requirements of a program without the need to use specific language constructs.

Specificity of Results

In general, perfectly describing the Permission requirements of a program is not possible. Runtime data is not available statically, and resolving all the Permissions themselves is in general undecidable. Runtime analysis can help to specify some cases of runtime data, but the general problem still remains. The simple fact is that all the information about Permissions will not be available. It is possible to get very close in many cases (and as we will show, other projects have succeeded at this to much greater degrees than this one [10] [9]). In general, however, the goal of identifying Permissions should be to get as much information as is feasible, and use it as a starting point for a further analysis. This further analysis can have the goal of providing a more complete understanding of the security requirements of the program.

1.3.2 A Simpler Model

The Action-Centric Model we will present in this report represents a particular high-level view of security. The goal of the abstraction from Java's security implementation is to allow for flexibility, extensibility, and understandability. Here we consider some of the benefits and applications of such a model.

Simplifying Policy Creation

Just defining the Permissions that go into a policy file is not necessarily an easy task. The parameters of each desired Permission must be specified, and the Permissions that "should" go into the policy file must already be known. Sun Microsystems created a program called "Policy Tool", which can be used to create individual policy specifications at the level of Permissions [1]. It has a graphical user interface (GUI) to reduce some of the complexity of dealing with permission syntax. The tool allows the user to specify what permissions should be granted. This can be used to specify commonly-found Permissions on a case-by-case basis. Another approach to defining how security should be controlled is suggested by the Security Annotation Framework (SAF), which uses Java annotations to enforce access control decisions [3]. This allows for further control over security in Java than what provided by Permissions. As far as defining policy in terms of Java Permissions, however, a tool that could simplify the specification of permissions would be desirable. We present an implementation of a high level model of security in the form of a NetBeans plug-in that demonstrates how such a model could help do this.

High-Level View of Security

This project attempted to present application developers and maintainers with a view of Java access control that focuses on larger tasks rather than fine grained authorizations. The Action-Centric Model removes much of the complexity associated with Java access control. Actions represent tasks which can imply many permissions, thus reducing the amount of items that need to be inputted into the policy. The action view also helps conceptualize the actual resource usage of an application by focusing attention on the resources being acted upon rather than the permissions required for execution. The parameters used to build actions are much simpler than Java permission parameters because they avoid confusing syntax and do not combine multiple pieces of input data into single strings. By translating actions into rules, then back to actions, the Action-Centric Model can detect implied actions. These implied actions can help find security holes that otherwise might go unnoticed. By combining all this functionality, the Action-Centric Model provides a view of Java access control that is more convenient and easier to understand than Java permissions alone.

1.4 Contributions

There are several ways in which this project contributes to solving some of the overall challenges present in using Java security. One particular theme is the usefulness of detailed permission information in the context of the problem - that is, making the task of developing an application to run under a security manager simpler. It is important to have information about permission requirements in order to create an effective policy. However, the task of creating policies permission by permission can be confusing and has a steep learning curve. This project began with an attempt to build a tool that would infer security checks using static analysis. The results of this analysis proved to be ineffective in accomplishing the main goal of simplifying the view of security. Both the inaccuracy of the analysis and the overwhelming amount of data that the analysis does present suggested that modeling security in terms of some abstraction of permission requirements would be desirable. A summary of our specific contributions:

1. An implementation of existing static analysis strategies for inferring permissions was built. This lead to mixed results, which are presented in 2.2. The results suggest that work in the direction of high-level modeling may benefit from existing work on static analysis.

2. An extensible, action-based model of access control was defined, which is used to provide some useful abstractions and models of Java Permissions. This model is detailed in 3.1.

3. The model provides for future modifications, customizations to particular security environments, and the possibility of portability to other security implementations. The extensibility of the model is discussed in 3.2.1.

4. Methods for interpreting information about existing policies or Permissions in existing code are discussed, which include the use of static analysis, and the use of its results in the model.

5. Possible further work is proposed in 4.3 for providing alternative methods of creating and using Java security policy, both at development time and runtime.

Chapter 2

Static Analysis

Attempting to statically infer permission checks is one way to help understand the security requirements of Java programs. Presenting an estimation of an application's permission requirements would give someone with knowledge of the security system the most accurate picture of what the requirements of the code actually are. This information could be used for review by the policy developer to create or inform the creation of policy files and to provide a starting point for future analysis. So, the goal of static analysis is to get as complete a picture as possible of the actual permission requirements of a Java program in terms of permission checks.

There was existing interest at Sun Microsystems in using static analysis to infer permission requirements of Java programs. Several small prototypes were available, but no complete solution. To better understand common situations that arise in Java's security context, and provide detailed information about Permissions, an implementation of static analysis for permission inference was attempted. This attempt, while not entirely successful, showed what kind of results can be expected from this kind of analysis, and what some of the challenges are in performing it.

In general, the detection of all permission requirements of a Java application is undecidable by Rice's Theorem[14], so any static analysis algorithm must decide whether to overestimate or underestimate. In our case, we will show that our analysis is conservative in that it reports a set of permissions that can contains (many) more permissions than are actually required.

2.1 Methods

2.1.1 Algorithm

Existing Algorithm

The algorithm used to perform the static analysis is a combination of call graph traversal and data flow analysis. It is based on an algorithm created by Larry Koved[10] to perform a similar analysis on Java 1.2 security. The level of correctness and termination of the algorithm are proved by Koved as well. This section briefly summarizes this algorithm to inform the description of the implementation that follows. The basic structure of the algorithm is as follows:

1. Create a complete, context-insensitive method invocation graph with edges E and nodes N.
2. Associate with each node ν a set of required permissions denoted by $P(\nu)$, initially empty.
3. For each node $\nu \in N$ that is a call to checkPermission() with permission π, let $P(\nu) = P(\nu) \cup \{\pi\}$.

4. For each node $\bar{\nu}$ which has an edge directed at ν (i.e. represents a method that calls ν), if $P(\nu) \not\subseteq P(\bar{\nu})$, then let $P(\bar{\nu}) = P(\bar{\nu}) \cup P(\nu)$, and repeat step 2 for all nodes with an edge directed into $\bar{\nu}$. Otherwise, stop.

Assuming that the permissions are checked at each call to checkPermission() are known (which is not an entirely safe assumption), this process leads to an overestimating approximation of the permissions possibly required for each method in the program. In the case of the call graph, most of the "extra" permissions come from inability to resolve virtual calls. Further approximation is made in actually determining the characteristics of these permissions (see 2.1.1).

Special Cases

There are several special cases that this algorithm needs to handle, based on particular constructs in the Java programming language and Java security specification. These mostly involve handling exceptional cases in the control flow or definitions of granted permissions in certain contexts.

Handling Threads When the start() method of a Thread object is called in the JRE, several things happen. The JVM executes some native code to create a new thread in the operating system, associates it with the Java Thread object, associates with this object *the access control context that created the* Thread *object*, and calls the run() method of the Thread in this context. This means that when a node in the call graph is encountered that represents a call to Thread.run(), the permissions required here do not necessarily reflect the permissions required in the context that called Thread.start(). Rather, they are required in the context that called Thread.<init>(). This means that if a node ν is a node representing the method Thread.run(), then in addition to traversing nodes $\bar{\nu}$ that call ν directly, the traversal should also go each node $\hat{\nu}$ which represents a constructor Thread.<init>() for this Thread object.

Handling doPrivileged() Invoking a method using doPrivileged() has special meaning for the security context below that call in the call stack. It indicates that only the security context of the caller should be taken into account in code executed within the doPrivileged() call. In terms of this algorithm, a doPrivileged() block is a stopping point for the graph traversal, since no context above the call requires having the permissions below the call. That is, when a node ν is encountered which represents doPrivileged(), the traversal should stop and not propagate the required permissions to any further nodes.

Handling SecurityExceptions In some programs, a SecurityException being thrown is expected behavior. One can imagine a situation where an application handles the exception that is thrown when a login fails, for instance, and the call to checkPermission() failing does not necessarily indicate an application failure. In terms of this algorithm, this means that a method which catches a SecurityException and handles it without re-throwing the exception is treated in the same way as a call to doPrivileged(). That is, the calls in the stack above the call to the method which catches the exception should not be marked as requiring all the permissions of that method.

Basic Data Flow Analysis

In the initialization of the above algorithm, the call graph analysis described above assumes that at each program point that calls checkPermission(), the Permission argument to the call is known. However, it is not always trivial to find the specifics of the Permission that is being checked. A simple data flow analysis algorithm was used to describe these Permission arguments in some simple cases.

Once the program point that calls SecurityManager.checkPermission() is found, it is simple to extract the expression that is passed as the Permission argument to the method. From this

program point, the list of statements in the body of the method is traversed against the direction of program flow to find an instance of the constructor call to the relevant Permission type, or assignment statements to the variable passed to checkPermission(). This analysis only considers the method that called checkPermission(), which is a simplifying assumption. If the constructor is found, then the expressions passed to the constructor can be extracted, and a similar traversal can be performed on these expressions to find their initialization. The difference in this case is that these expressions (almost always String objects), are often declared as static fields, so the static initializers of the relevant methods are also inspected to find definitions of static fields. This analysis is sufficient for many simple cases, but misses many others, most commonly when the expressions passed to the Permission constructor come into the method as parameters and their runtime values are more difficult (or impossible) to determine.

Much of the simplicity of this traversal is due to the intermediate format Jimple from Soot, which simplifies compound expressions in Java statements [6]. Soot will be discussed in more detail in the next section.

2.1.2 Implementation

A number of different strategies were considered for implementing the call graph and data flow analysis algorithms. Of primary importance was the ability to construct a Java method invocation graph, tools for traversing it, and the possibility of using the model to perform data flow analysis. In the end, Soot, a "Java Optimization Framework,"[13] was chosen. It has the benefits of being well established in other large applications and research projects, being well documented, and providing built-in call graph generation and methods for acting on generated graphs. This project only used a small subset of the features of Soot, which has many other applications.

Soot

Soot takes Java source code, byte-code, or class-files as input and converts it into different representations or models of the code. These representations simplify the process of performing many tasks, including several types of static analyses. Of particular interest to this project was Soot's ability to make call graphs and operate on them [6].

There are four different models that Soot uses for representing Java Code, each with different uses. The one that was used in the analyses in this project is called Jimple, which is Soot's default intermediate code model. It breaks complicated or compound statements down into more basic ones (for example, i = x + y + z could become i0 = x + y; i = i0 + z), to simplify statement by statement analysis. So, when using Jimple, Soot creates a number of intermediate variables, each of which represents a single argument to a method invocation, class field, or local variable (that is, the variable is never used more than once). This feature was especially relevant in simplifying data flow analysis [6].

Call Graph Generation Soot's call graph generation was used to create the method invocation graphs that were the core data structure in the static analysis algorithm. For the purposes of this project, a graph was needed that was:

1. *Context-insensitive* - When performing the call graph traversal, the specific call sequences that could lead to a method invocation were not important. Instead, the set of *all* methods that could invoke a given method was needed for each method, since all of the invoking methods *could* need the permissions checked in any method which they transitively invoked.

2. *Aware of native code functionality* - When doPrivileged() or Thread.start() are called, the JVM uses a native method to execute the run method of the PrivilegedAction or Thread in a special context or system thread, as the case may be. The call graph creation in Soot allows

for the insertion of implicit edges to PrivilegedAction.run() and Thread.run() from the native calls.

With Soot's call graph generation, a set of Java class-files or source-files are provided, along with the name of the main class and a set of entry points to the code to be analyzed. Starting from these method entry points, Soot is capable of generating call graphs in a number of different ways, including allowing for these two important features.

Type Analysis When creating a method invocation graph in a language with inheritance like Java, resolving virtual method calls statically can be a challenge. Being unable to resolve the runtime type of objects before the creation of a call graph can seriously limit accuracy. For example, if there is a given method that takes an argument of type Object, and then some method say toString() is invoked on that object, a totally naive algorithm would have to assume that that the toString() method of *any* class currently loaded could be called, since every class inherits from Object. Now, in simpler cases (like the basic Object methods), this can be avoided. But with complicated interactions between interfaces and numerous implementing classes, it is difficult (and in general undecidable) to determine which implementation of a method will be called statically.

To remediate this problem somewhat, more of Soot's built in features were used, which included the ability to do *on-the-fly* call graph generation. This technique performs the call graph construction along with a *points-to* analysis, using the information from the points-to analysis to provide a more accurate representation of possible calls. A points-to analysis is a type of data flow analysis, the goal of which is to determine the possible values of references at each program point (i.e. the possible values each reference "points to"). Using this method, the size of the call graph created by Soot was roughly cut in half.

Data Flow Analysis The algorithm used for data flow analysis described above was implemented in a straightforward manner, using the Jimple representation of Java code. A given Jimple statement in the Soot-generated code has references to the statements that come immediately before and after it. So, once the field is found that refers to the Permission argument to a checkPermission() call, this can be isolated as a Jimple identifier. From the method invocation statement, Jimple statements are traversed backward, searching for assignments statements that include the isolated identifier. If the Permission object was instantiated in the current method, eventually a method invocation to Permission.<init>() will be found (or the constructor of some specific subclass of Permission). The arguments to *this* constructor can then be found as Jimple identifiers, and a similar process can take place with them (i.e. finding where they are initialized as string constants).

As mentioned above, it is common practice to declare specific Permissions or relevant String constants as final static fields of the class in question. So, the data flow analysis also searches the <clinit>() method for initializations of the fields that are found in permission checks or Permission constructors.

This simple process is able to totally describe the checked permissions in many commonly found cases. Extending the analysis further would allow more cases to be found, but ultimately cannot succeed in all cases. If the Permission object or its arguments are passed into a method, for example, interprocedural data flow analysis could provide more results, but may only find that the information simply cannot be known until a user provides it at runtime.

Analysis Class Structure

The results of the static analysis were intended to fit into a larger framework that could be used to take information about permission requirements from different sources and aggregate them into one combined model. The structure would keep track of which permissions were required at which program points, designated in the model as CodeBlocks. A CodeBlock represents an

arbitrary segment of code of any size that has some PermissionRequirements associated with it. These requirements would have associated with them a description of the permission and the source that decided the requirement existed (this could be from the static analysis itself, or developer specified). This would make it possible to compare requirements from different analyses, like static analyses with different kinds of call graphs, or a runtime analysis. Figure 2.1 shows the structure of these classes.

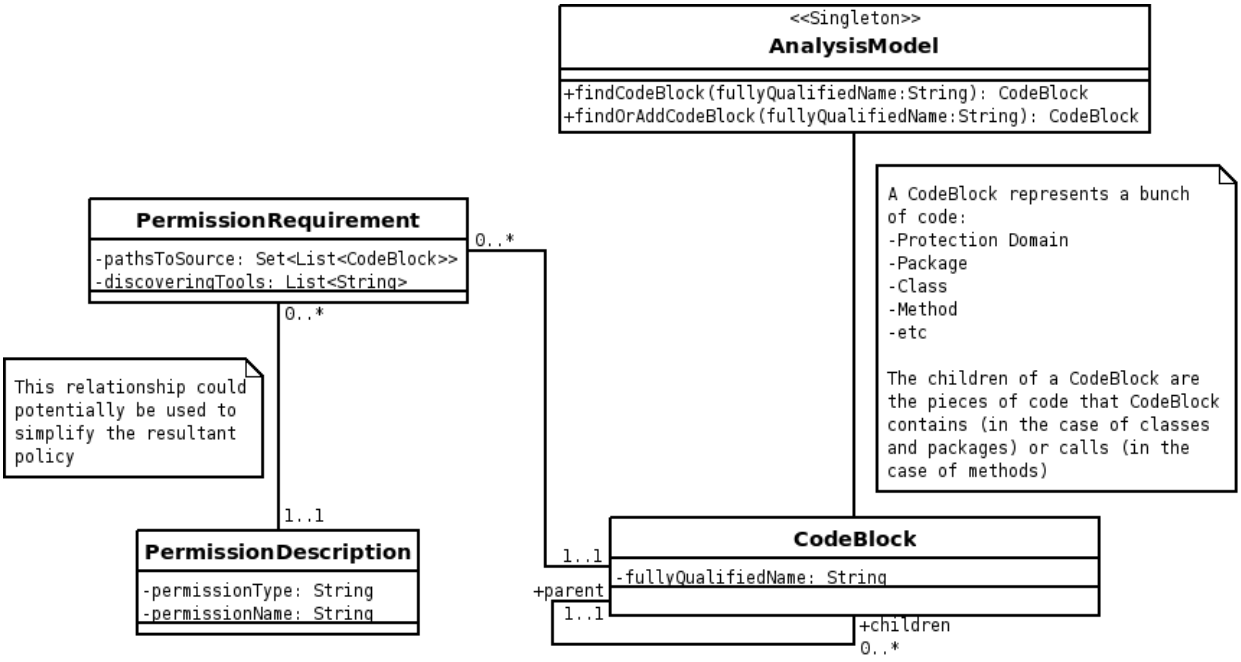

Figure 2.1: The static analysis data structure

This structure also includes ways for outside sources, like a policy file or user input, to enumerate expected or required permissions (not shown in Figure 2.1). These outside sources could be compared with the model resulting from static analysis. The model could show if the expected results were found, or add other requirements than those found by a given analysis.

2.1.3 Limitations of Static Analysis Data

To be precise, the static analysis described here provides information about which Permissions a certain set of code needs to be granted to not throw a SecurityException at runtime. If the analysis were perfect (which it cannot be, this type of analysis is undecidable), it could specify a precise policy file entry for each required Permission. This would accomplish the goal of a developer being able to run the static analysis and generate a policy file that allows the program to run under the default SecurityManager, following the principle of least privilege for the required permissions.

In practice, this information is incomplete because of the lack of runtime data, and is overly conservative due to the virtual call problem. However, even if perfect information was available, this type of analysis can only tell a programmer or developer which low-level Permission objects are required to run the code successfully. This information can be quite useful in informing decisions about policy, but says little about what the Permissions actually mean for what the program was designed to be allowed to do at a high-level. For example, allowing permissions to create reflected objects is quite powerful, as it allows sidestepping visibility modifiers like protected and private. Simply saying that a piece of code that does reflection will throw an exception if a given permission

is not granted does not necessarily mean that the code should have these capabilities - that is up to the developer and the application deployer. The results of the static analysis do not give an indication as to what the expected capabilities of the program are or should be, just what the security implementation is requiring of the program. A higher level model of security that did not speak simply in terms of the implementation in terms of Java Permissions could be valuable in helping to interpret the data resulting from such an analysis, or specifying the analysis from the top down in the first place.

2.2 Results

2.2.1 Successes

This section discusses the results of the static analysis procedure.

1. The analysis successfully performs a conservative analysis. That is, the actual permission requirements of the program are a (relatively small) subset of the results provided by the algorithm.

2. The data flow analysis captures specific permission information in several commonly found cases.

The static analysis procedure was tested on several small sample problems, most of which were designed to include the special cases of Thread objects and doPrivileged() blocks. The semantics of these cases were successfully handled, and the propagation of permission requirements between methods worked as expected. A simple test case of the static analysis procedure can be seen in Appendix A.

Summary of Permission Requirements

While the static analysis is conservative, it still provides useful results. Information about how one permission check in a given method propagates through the program is available. This is useful in the case of several interacting classes with methods that provide security-checked functionality. It is possible to see at the class and method level what permissions are required as a result of all calls that are made from that point. Since the analysis is conservative and overestimates (drastically) the number of permissions, it will report all the permissions that it finds, most specifically user-defined permissions that are inside the space of the application (as opposed to permissions that are checked in calls into JVM library methods). While the analysis will not be able to report an exact set of the required permissions, it can give hints as to where secure behavior may not be encapsulated and spreads the requirements to other areas of the program (other classes and protection domains). This can provide the start of a higher level analysis of the behavior of the application in terms of its permission requirements.

Permission Information

Due to the fact that the information specifying permissions can be found relatively “close” to permission checks, the analysis is often able to provide the specifics of a given required permission in terms of its arguments. This is mostly true in the case of code in libraries that does these static or “nearby” declarations as a common practice. The importance of finding this information is especially true in the case of permissions where the type of the permission is not as informative as the arguments to the permission. For example, RuntimePermissions represent a wide array of capabilities for a program, from creating reflected objects to getting system properties. The actual capability that is described is almost entirely dependent on the argument to the permission. The ability of the analysis to find this information can be quite valuable.

2.2.2 Limitations

There are a number of ways in which the static analysis is limited. The conservativeness of the algorithm has been mentioned, because of the overly verbose call graph. Other limiting factors are discussed here as well.

Call Graph Verbosity

The main challenge facing the algorithm is creating a call graph that is complete while not being overly verbose. The problem is mainly due to the difficulty in resolving virtual calls and performing static type identification. Even using the optimizations that Soot provides, the call graph still has many edges that are added amongst calls to JVM methods which take Object type parameters, and this is because the analyis cannot resolve the type of parameter and assumes all possible subclass calls are possible. There is a lot of existing research on creating more accurate call graphs in object-oriented languages (and handling the virtual call problem) [15], [8]. Unfortunately, further investigation of these methods was outside the scope of this project.

One solution that was considered in this project was to annotate library code with the permissions that are required at each entry point, so the analysis could start from these points instead of calls to checkPermission() inside the library itself. The idea of using annotations to perform security checks has been investigated[3], but this tends to work to *enforce* security, rather than describe or report existing information.

Weak Data Flow Analysis

The data flow analysis performed for this project is often able to find specific information about permissions, but there do still exist many cases where this is not possible. The approach that is used relies on the practice of defining permissions in certain places and ways (that is, statically or in the same method as the permission check), which are not required. For programs written in certain ways, the algorithm may be totally unable to detect information about the permissions.

Runtime Data

The most unavoidable limitation of the static analysis is that any information that is provided at runtime cannot be known by the static analysis. A common example of this is a filename supplied by the user during program execution. The algorithm may be able to determine if a "read" or "write" permission is requested, but no way to identify the name of the file statically. This could also be the case for an application that is designed to get information at runtime from other applications and use this dynamic data to construct permission objects. This is a limitation of static analysis in general, and a reason to explore other options to combine with static analysis to produce a more complete picture of security requirements.

Chapter 3

Action-Centric Model

Even if it was possible to use static analysis to determine the required policy for an application, the major access control related problems that face application developers and maintainers are not solved. First, static analysis does not explain why permissions are needed. There could be logical errors in the application that cause it to use resources that it should not. Second, the verbosity of permission syntax is not addressed at all, since all calculations are done using permissions only. Lastly, there is sill no notion of how the permissions interact with each other. This means that a security hole could be disguised within a set of permissions. This sort of error is difficult to detect even to an experienced policy writer. To address these concerns, we developed the *Action-Centric Model*.

The Action-Centric Model for Java Security was developed to provide a higher level view of Java Security. The Action-Centric Model focuses on the resources available to the developer and what actions can be performed on those resources. Its purpose is to mask the complexities of permission such as syntax concerns, syntax verbosity, unclear relationships between permissions, and unfamiliar vocabulary of permissions. It attempts to present a more expressive and extensible way to represent Java security policy than permissions.

3.1 Methods

3.1.1 Design

To describe the Action-Centric Model, we show how it models a Java SocketPermission. A SocketPermission, like many other Permissions, has two String fields that describe it. As described in the JDK Documentation, the first is a "host specification," and the second specifies the ways in which the host will be accessed [12]. An example of a host and access specification:

- host = "sun.java.com:5000-6000"
- access = "accept"

So in Java syntax, we would have:

```
SocketPermission sunSocket = new SocketPermission("sun.java.com:5000-6000", "accept");
```

The host specification contains a host specified as an address, followed by a colon, followed by a port range. The need to use this form of the syntax could be avoided if the host and the port range parts of the parameter could be specified independently. In the Action-Centric Model, the same SocketPermission is represented by an instance of the "Accept" *Action Definition* which can be found in the "Socket Connections" *Resource Definition*. A Resource Definition defines a

resource that is subject to access control and is used to group Action Definitions together. Action Definitions define tasks that can be performed on a resource along with all the parameters that are required to perform it. An instance of an Action Definition is called a *Filled Action*. In order to create a useful Filled Action, all of the Action Definition's parameters must be specified. When the "Accept" filled action is translated, an instance of the "Socket Permission" *Filled Rule*, which contains all the data needed to create the desired socket permission, will be the result. The Filled Rule that is created by the translation can be viewed as if it were a permission object. The usefulness of abstracting rules from permissions is detailed in 3.2.1, but for the purposes of most of this document, it is acceptable to view filled rules and permissions as the same thing. In the Action-Centric Model, the instance of the Filled Action with all parameters specified in would be as shown in Figure 3.1.

- Socket Connections
  - Accept
    - address=sun.java.com
    - portrange=5000-6000

Figure 3.1: Example of filled socket accept action

The filled action shown in Figure 3.1 could be translated into an instance of a Rule Definition that directly maps to the Permission shown in Figure 3.2.

- SocketPermissionRule
  - host = "sun.java.com:5000-6000"
  - access = "accept"

Figure 3.2: Example of SocketPermissionRule

In this case, it is fairly easy to see how the translation can happen - the two parameters in the filled in action, *address* and *portrange*, are inserted into the correct locations to fill in a rule that specifies a permission. The actual structure of the action definition, without any filled in parameters, would be as shown in Figure 3.3.

- Socket Connections
  - Accept
    - (address, portrange)
    - AppliedRule(SocketPermissionRule)
      - host="{address}:{portrange}"
      - access = "accept"

Figure 3.3: Example of socket accept action definition

In the general case, these parameters haven't been filled in yet, so we see the locations in the *AppliedRule* where they would be placed in the translation. host="{address}:{portrange}" is a mapping of the parameters of the action to the parameter that would be used to define a Permission.

The particular components that make up the model (shown as the mapping of "Resource-ToRule" in Figure 3.4) are:

- **Resource Definition** - *Resource Definitions* are named groups of *Action Definitions*. They are simply used to describe where a particular Action Definition "belongs" (examples include the "Socket Connections" resource, the "File System" resource, and the "Security System" resource).
- **Rule Definition** - Rule Definitions are the representation of Java Permissions in the model. Each Rule Definition in the model as defined corresponds to exactly one Permission class.
- **Applied Rule** - An Applied Rule describes how parameters of an Action Definition should be translated into parameters of a particular Rule Definition.
- **Action Definition** - An Action Definition is a specification of how to translate a security capability or behavior in the high level model to Permissions. It contains a set of parameters, denoted as its Action Parameters, and a set of Applied Rules, which describe how these parameters translate into the parameters of specific rules.
- **Filled Rule** - A Rule Definition with values for its parameters.
- **Filled Action** - An Action Definition with values for its parameters.

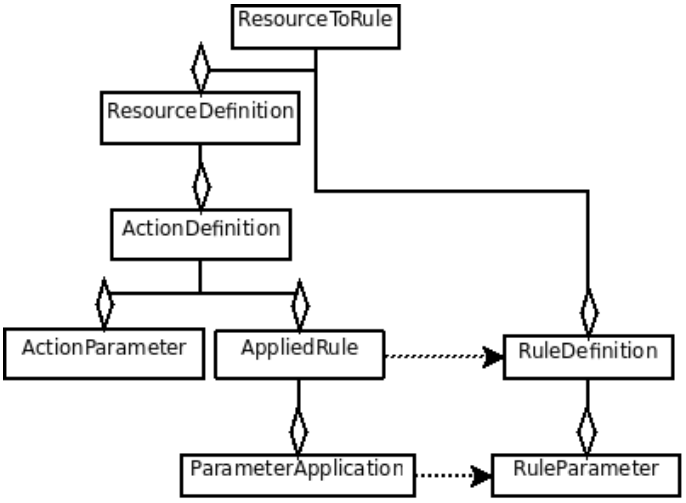

Figure 3.4: The relationship between resources, actions, and rules

Implementation of Resources Resources are represented by the ResourceDefinition class. The ResourceModel keeps track of all the ResourceDefinitions that are available in the current model and provides methods to search for specific ResourceDefinition instances.

Implementation of Actions Each ResourceDefinition contains a set of ActionDefinitions describing all possible actions that can be performed on the resource. Each ActionDefinition contains a set of ParameterDefinitions describing all the input data required to create a FilledActionDefinition. Each ParameterDefinition may supply a default value, or not. A FilledActionDefinition is instantiated with an ActionDefinition as an argument to its constructor. The parameter value mapping and examination functionalities are provided by FilledActionDefinition's super class, FilledParameterContainer. FilledParameterContainer provides the ability to map ParameterDefinitions to their appropriate values. It also provides functionalities to help the parameter mapping process, such as the ability to find unmapped parameters and the ability to resolve parameters to their defaults.

Implementation of Rules All possible rules are represented by a set of RuleDefintions contained in the singleton ResourceModel instance. Each RuleDefinition is labeled with a "rule type" attribute, allowing rules describing multiple security systems to exist in the model concurrently while still being easily distinguishable. RuleDefinitions contain a set of ParameterDefinitions defining all input data required to create a FilledRuleDefinition. FilledRuleDefiniton extends FilledParameterContainer to deal with parameter manipulations, just like FilledActionDefinition.

Conversion

An important feature of this model is the facility to translate from Permissions to Filled Actions and vice versa. While the model uses the language of Rule Definitions and Filled Rules for Java Permissions, the important result is the conversion to and from Filled Actions and Permissions. The reasons for using the layer of abstraction provided by rules is discussed later. For this discussion, it can be safely assumed that Rule Definitions have a one to one association with Permission classes, and Filled Rules have a one to one association with instances of Permission objects.

Using a fairly straightforward algorithm, it is possible to translate a set of Filled Actions into a set of Filled Rules. A more complex algorithm is used to translate a set of Filled Rules into a set of Filled actions. In this section we describe the specifics of mapping the parameters of actions to the parameters of rules, and the translation of Filled Actions to Filled Rules and back.

Parameter Mapping The mapping between actions and rules is done via the applied rule construct, which describes how some number of action parameters can be converted into the rule parameters for one particular action definition and rule definition. When filled actions or filled rules are created, applied rules can be used to translate back and forth.

To meet these requirements, the mechanism for mapping from one set of parameters to the other needed a few qualities:

1. Reversibility - The process used to convert from an action to a set of rules had to be able to be reversed to convert from a set of rules to actions. Ideally, this would mean a 1-to-1 function from sets of action parameters to rule parameters.

2. Simplicity - A simple mechanism for defining these mappings would allow for easier specification of rule and action definitions, and help with reversibility

3. Expressibility - The mapping needed to represent a number of ways of taking information expressed in terms of actions and converting to the level of detail found in rules (similar to the level of detail in permissions).

The method that was chosen was a fairly straightforward variable replacement. An applied rule would specify, using special characters, the places in a string where to substitute the value of an action parameter. To enforce reversibility, adjacent action parameters are not allowed in the parameter mapping. Any number of action parameters can be used any number of times in the parameter mapping. Formally, a parameter application takes the form:

$$(\{\rho\}\omega^+)^* + (\omega^+\{\rho\})^* + \{\rho\}(\omega^+\{\rho\})^* + \omega^+$$

Figure 3.5: Formal expression of parameter applications

With a few extra restrictions, where ρ is an action parameter and ω is any character other than "{" or "}". Note that "{" and "}" are the delimiters for where action parameters are substituted (and are removed in the substitution), and () have the usual regular expression meaning in the form shown above. The only remaining restriction is that if there is a string between two substitution clauses, the action parameters cannot have that string as a substring. For example, given action parameters A and B, the following are legal parameter applications:

- $property.\{A\}.\{B\}.*$
- $/tmp/\{A\}/\{B\}$
- $\{A\}.\{B\}.com$
- $read$

But in the first case, if A or B contained a single "." it would be impossible to reverse the process given the substituted string. The same is true of B in the third case, and if A contained a "/" in the second.

Action to Rules Conversion To convert FilledActions (that is, actions with specified parameters), to FilledRules, the process is straightforward. ActionDefinitions have AppliedRules that specify particular RuleDefinitions which they map to, and descriptions of how to insert the ActionParameters into specific RuleParameters. The mapping the parameters in this direction is implemented with a simple string substitution algorithm, here denoted as $mapToRuleParameter()$. The algorithm for conversion follows is shown in Figure 3.6.

1: **ActionToRules**(A [A $FilledAction$])
2: $P \leftarrow ActionParameters(A)$
3: $R \leftarrow \emptyset$
4: **for all** $\alpha \in AppliedRules(A)$ **do**
5: $\tau \leftarrow \emptyset$
6: **for all** $\rho \in ParameterApplications(\alpha)$ **do**
7: $\tau \leftarrow \tau \cup \{mapToRuleParameter(P, \rho)\}$
8: **end for**
9: $R \leftarrow R \cup \{createRule(RuleType(\alpha), \tau)\}$
10: **end for**
11: $returnR$

Figure 3.6: Creation of filled rules from filled actions

Rules to Actions Conversion The process of converting from a given set of FilledRules into a set of FilledActions is slightly more complicated. Since a given Rule could be applied by many Actions, it is necessary to go through all the AppliedRules for each Action Definition, find those FilledRules of the same type as the Applied Rule, and perform reverse mapping of the Rule's parameters into potential Action parameters as specified by the Applied Rule. The first part of the algorithm does the reverse variable mapping and builds sets of rules and their reverse variable mappings that correspond to particular applied rules. This is shown in Figure 3.7. The second phase of the algorithm takes these associations of applied rules, filled rules, and reverse parameter mappings, and groups them into sets that fully represent filled actions. This is shown in Figure 3.8.

Further discussion of a few points of the algorithm:

- A mapping (that is, the name of a variable to a value), in this case is referring to ActionParameters which can be inferred from the RuleParameter of a FilledRule.
- Any one of these mappings can "conflict" with the parameters in a given FilledRule. In line 10, this check is to ensure that a given set of FilledRules that compose a FilledAction will agree on the values of the parameters for the FilledAction, since a large initial set of rules may be analyzed.

1: **RulesToActions**($ActionDefs$ [*A Set of ActionDefinitions*], $Rules$ [*A Set of FilledRules*])
2: $\pi \leftarrow \emptyset$ {π will be an association from AppliedRules to tuples of FilledRules and sets of parameter mappings}
3: $RuleSets \leftarrow \emptyset$
4: **for all** $A \in ActionDefs$ **do**
5: $\quad \pi \leftarrow \emptyset$
6: $\quad$ **for all** $\alpha \in AppliedRules(A)$ **do**
7: $\quad\quad \rho \leftarrow \emptyset$
8: $\quad\quad$ **for all** $R \in Rules$ **where** $R = RuleType(\alpha)$ **do**
9: $\quad\quad\quad \rho \leftarrow \rho \cup \{\langle R, reverseMap(RuleParameters(R, \alpha))\rangle\}$
10: $\quad\quad$ **end for**
11: $\quad\quad \pi \leftarrow \pi \cup \{\langle \alpha, \rho \rangle\}$
12: $\quad$ **end for**
13: $\quad RuleSets \leftarrow RuleSets \cup$ **GetSetsOfRulesForActions**$(\pi, \emptyset)$
14: **end for**
15: return $CreateActionFromSpecificSet(RuleSets)$

Figure 3.7: Converting rules to actions, step 1

- The return value $ReturnSets$ will contain a set of sets of FilledRules, and each of these sets will fully describe a single FilledAction. The function $CreateActionFromSpecificSet$ takes one of these sets and creates the particular FilledAction.

So, given a set of Permissions, these algorithms can be used to create a set of corresponding Filled Actions defined by particular Action Definitions.

Special Cases Several interesting things happen when rules are converted. Consider two actions A and B where action A requires rule 1 and action B requires rules 1 and 2. If action B is granted, then translated into its component rules, then translated back to actions, we find that the resultant set of actions is now A and B, even though only B was granted. We call B an *implied action*. While this may seem like a logic problem at first, we can actually use this to our advantage. By implying additional actions from the set of actions granted, a more complete understanding of the security requirements is gained. For instance, imagine a more complex example using 3 to 5 actions implying 1 or 2 others.

Consider a case where an access control policy is already in existence and has been represented in filled rules. For example, filled rules 1, 2, and 3 are granted. Action A is the only action defined, and it requires rules 1 and 2. When the filled rules are translated to filled actions, only action A will be represented. If the resultant action were to be converted back to rules, only filled rules 1 and 2 would exist; filled rule 3 would be lost! Filled rule 3 is referred to as an *unmatched rule*. Because it is not acceptable to lose security requirements during translation, the Action-Centric Model has a special container for unmatched rules such that they can exist and influence the resultant security policy without being associated with a filled action. Unmatched rules can be incorporated back into actions at a later time if either more filled rules are added such that an action could be created using the unmatched rule or a filled action is added that requires the filled rule represented by the unmatched rule.

3.1.2 XML Specification of Definitions

For this project, the structure for defining resources, actions, and rules is given in an XML schema. An XML document following this schema can define an explicit set of resources, actions, and rules that represents security capabilities and references an implementation in the form of rules. In the case of this project, all of the rules described Java Permissions. It would be a just as straightforward

to make rules for any other security system that would like to describe actions in terms of low-level items or statements in the implementation.

The XML structure was decided upon for a few reasons:

- The containment structures of XML schema are appropriate for the structure of resources, actions, and rules.
- The process of creating and editing XML is well known, which supports extensibility of the definitions.
- There are tools, such as JAXB, used in this project, that enable flexible XML modifications to facilitate conversion from XML to Java objects and back again.

The structure of the model in terms of XML mirrors the form shown earlier in Figure 3.4. The root element, ResourceToRule, is an aggregation of a set of ResourceDefinitions and RuleDefinitions. ResourceDefinitions contain a set of ActionDefinitions. These ActionDefinitions contain a set of ActionParameters, and a set of AppliedRules containing ParameterApplications. These constructs specify just how the parameters specified in an action will be mapped to rules. The arrows between AppliedRule and RuleDefinition, and between ParameterApplication and RuleParameter, represent the relationship between actions and rules is found in the model. In XML these are only specified by giving the name of the referenced RuleDefinition or RuleParameter within a RuleDefinition.

One additional feature provided in the XML description of definitions is the ability to specify parameters to "group" actions by. It is possible to put a "group by" annotation on an action parameter, which is used in the tool to group filled actions of that type by the parameter specified. This is particularly useful in the case of things like FilePermissions or SocketPermissions, where several actions may reference to the same file or socket parameter.

As an example, the Socket Accept action shown in the initial example is shown in its XML specification in Figure 3.9.

3.1.3 NetBeans Plug-In

To apply the model to the development process, it was necessary to design a method by which instances of the model could be created and manipulated. The end goal would be to provide an interface to create and edit both the definitions of actions and rules, and filled instances of actions and rules. The filled instances could be turned into policy files based on the specification of the rules, and the definitions could be saved as definition files.

While all of these goals were not accomplished, some of these features were implemented as a NetBeans plug-in. It provides a toolbox for a user to load a definition file, create actions and rules, and see how the mappings between the two work. See Appendix B for a guide to using the tool. In particular, the tool as implemented allows a user to:

- Load a definition file and view the resources, actions, and rules defined.
- Create filled instances of actions and rules, and see the relationship between them.
- See a diff-type view that compares two policies in terms of actions and rules side-by-side.
- Save and load configurations into the tool

In Section 3.2, several desirable extensions to the tool are discussed.

3.2 Results

3.2.1 Successes

The Action-Centric Model provides a new way to view Java access control policy. By providing a higher-level view of Java access control, the Action-Centric Model allows application developers and maintainers to produce and analyze complex policies in a familiar vocabulary that is both simpler and more intuitive than Java permissions. The model allows for any set of action and rule definitions to be used. This makes the model very flexible in the sense that any user has the ability to redefine the entire structure of possible actions and rules. The abstraction of rules away from permissions makes the model independent of the underlying access control system, making it possible to support other access control models in the future.

Creating Definitions

To get a large set of definitions to test the functionality of the model implementation, the vast majority of the standard named Permissions in the JDK were defined using resources, actions, and rules in the Action-Centric Model. This included creating actions that referred to specific sets of named Permissions, and parameterized actions that could refer to Permissions with more complex parameters (similar to the socket example in section 3.1.1). These definitions provide an example of how the model can be used to group together Permissions which either have similar meanings or combine to mean something more.

In general, the definitions we created provided parameterized actions in obvious cases (files, sockets, and other permissions without a specified, finite set of possible choices). In other cases, we grouped named Permissions together into actions when they were obviously related to a common resource or task. We expect that security developers with more domain specific knowledge could define definitions which would be more useful in certain environments. Some examples are shown in Appendix C.

The ability to create definitions in the form of this model helps in two particular ways.

1. **Convenience** - To take an example, an applet developer may not know or care to know that three AWTPermissions, constructed with target fields “createRobot,” “accessEventQueue,” and “listenToAllAWTEvents” are required to create and manipulate certain on-screen events. If this developer is presented with a tool that shows the “Applet and Media Output” resource with the “Control AWT Events” action, it may be easier to simply select this option. This provides an acceptable approximation of the security requirements of the program, rather than having the developer learn each of these requirements. In this way, the model and tool aim to provide a simpler mechanism for uninitiated security users to specify program policy.

2. **Interpretation of existing policy** - An existing set of Permissions can be interpreted in terms of defined actions and rules. That is, as long as there is a rule defined that refers to a particular Permission, along with a set of actions grouped by resources relevant to the security context, the low-level information from the security information can be interpreted at a high level. Information about Permissions could come from existing policy or the results of a static analysis. This is useful in interpreting the permissions that express access to low-level security resources like properties, library loading, sockets, and the file system. Grouping these things by the parameters of the actions can show relationships that exist between them. For example, it could be that the ability to load a specific library and read a specific directory, in combination, is particularly interesting. The model allows for an interpretation of these cases from implementation level security information (Permissions), and also the top-down specification of such policy.

NetBeans Plug-in

A plug-in for NetBeans was created to exhibit some of the useful features of the model. More than that, it really is representative of one environment in which the model could be well used. Particularly in the case of web applications, a security policy is almost always part of the development process, and this tool is the start of a way to easily define this kind of policy at development time. In the ideal version of the tool, a user would specify actions that described what they were trying to do with their application, and the tool would take care of converting this into Permissions for their application to use.

In addition to the simpler interface to Permissions it offers, the tool provides functionality for advanced security users. A user can choose to specify an instance of the model in terms of specific rules, along with actions. This is similar to using Policy Tool[1] to create specific Permissions. The tool reports what actions are composed from the various rules in the current policy configuration. In addition, multiple policies defined by either method can be compared to one another using the policy configuration diff view. This allows two policies, in terms of actions and rules, to be compared side by side. This could be used to compare potential new policies to old ones, or simply see how changing certain actions and rules alters the policy. Defining the policy in terms of rules and using the diff view to compare configurations would be methods used by someone knowledgable in the security domain using a definition model to better understand policy decisions.

Extensibility

One of the primary desirable attributes of the Action-Centric Model is its extensibility. The definitions file can be edited to include new kinds of resources, actions, and rules, that refer to one another or existing ones. This flexibility allows for different interpretations of the meaning of actions to be expressed without any changes to the underlying Action-Centric Model. Certain systems may define their own Permissions. The definitions file allows actions to be added to existing or new resources that map to new rules representing these domain-specific Permissions.

The implementation of the model in XML definitions is one feature that allows for this extensibility. The model is quite portable - one could imagine bundling a definitions file and an instance of a policy configuration with some code to specify the policy in this format.

Independence from Security Implementation

It is theoretically possible to export to or import from almost any sort of policy. The policy converters developed by this project were designed specifically for Java Permissions, however, the model was designed with a much broader view. These other policies can be supported simply by developing some simple converter classes and adding rule conversion definitions.

Once new policy types are supported, the Action-Centric Model could act as a sort of universal security description language, by which policies written in any supported form could be described in the context of actions and exported to any supported policy type.

3.2.2 Limitations

The current design and implementation of the Action-Centric Model leaves some things to be desired. Extensions are discussed in detail in 4.3.1, but a few of the prominent limitations are noted in particular here.

Parameters in Lists

Each action parameter is always specified as a discrete named field. There are some permissions in Java that have more advanced parameter syntax than the Action-Centric Model can describe using this approach. It would be desirable to, in some cases, allow a certain action parameter to

act like a collection. For instance, DelegationPermissions have one string argument which contains a list of paired things (in this case "Principals" and "Subjects"). This permission is intended to allow certain capabilities between these pairs of things in this list. In the current definition of actions, there is no way to have an arbitrarily sized list as a parameter. This means there is no way to define an action to rule translation for DelegationPermission that would allow a series of principal and subject pairs to be defined. For this reason, the action defined in our definitions is forced to expose the permission syntax to the action parameter. A single parameter must be filled with the textual representation of the list of principal and subject pairs.

Self-Reference of Actions

Currently in the model, resources contain actions and actions contain parameters and applied rules, which reference specific rule definitions. There is no case where a resource could contain a sub-resource, or an action could contain a sub-action. This limits the definitions to rigid categories. More expressiveness could be gained by allowing some type of recursive nesting, especially within actions. This would make definitions more reusable, and make extensions to the model easier to manage.

Testing and Use of Definitions and Tool

The tool as implemented is primarily a proof of concept of the model design. It shows the relevant features of the model using definitions of resources, actions, and rules that were created from common Java Permissions. For further improvements to the model and the tool, as well as an understanding of what kinds of definitions are most effective, using the tool in the actual development of several applications would be desired. It would be valuable to see which types of definitions are deemed the most intuitive and provide the most flexibility, and how much the model is judged to actually help inexperienced (and experienced) security users define policy. Further feedback from developers used to the Java security environment would also help to create new definitions and understand how the tool and model could be improved.

```
1: GetSetsOfRulesForActions(π [A set of tuples as described above], RuleSetAcc [An
   accumulator])
2: if π = ∅ then
3:   return ∅
4: end if
5: ρ ← ϵ ∈ π
6: π ← π \ {ϵ}
7: ReturnSets ← ∅
8: for all ⟨Rule, Mappings⟩ ∈ ρ do
9:   {Recall that ρ contains tuples of rules and sets of reverse parameter mappings}
10:  if Any mapping from RuleSetAcc conflicts with Rule then
11:    return null
12:  end if
13:  RuleSetAcc ← RuleSetAcc ∪ {⟨Rule, Mappings⟩}
14:  RestOfSets ← GetSetsOfRulesForActions(π, RuleSetAcc)
15:  if RestOfSets = null then
16:    return null
17:  end if
18:  if RestOfSets = ∅ then
19:    ReturnSets ← ReturnSets ∪ {{⟨Rule, Mappings⟩}}
20:  else
21:    for all RuleSet ∈ ReturnSets do
22:      RuleSet ← RuleSet ∪ {⟨Rule, Mappings⟩}
23:      ReturnSets ← ReturnSets ∪ RuleSet
24:    end for
25:  end if
26: end for
27: return ReturnSets
```

Figure 3.8: Converting rules to actions, step 2

```
<action name="Accept">
    <description>
Represents the action of accepting a socket connection.
    </description>
    <actionParameters>
        <actionParameter name="address" />
        <actionParameter name="portrange" />
    </actionParameters>
    <appliedRules>
        <appliedRule type="Permission"
        name="SocketPermission">
            <parameterApplications>
                <parameterApplication name="host"
                        value="{hostname}:{portrange}" />
                <parameterApplication name="access"
                        value="accept" />
            </parameterApplications>
        </appliedRule>
    </appliedRules>
</action>
```

Figure 3.9: XML Specification of Socket Accept

Chapter 4

Evaluation and Future Work

4.1 Permission Identification

As this project progressed, it became apparent that static analysis would be unable to solve the problems at hand. Because of the undecidable nature of the problem, it will never be possible to accurately determine all permission requirements statically, nor will it be possible to precisely determine the all parameters required to instantiate Permission objects. Because the algorithm used by this project favored overestimation, the output is overwhelming. The compute time needed to run the analysis is also fairly high, which would discourage its use. Even if all these issues were solved, the data is only useful to make the analyzed application run. No data is provided about why permissions are required or how they interact with each other to produce an access control policy.

4.2 Action-Centric Model

The Action-Centric Model combines several important features to deliver a new perspective to Java access control. The high-level view of access control provided by the Action-Centric Model allows application developers and maintainers to develop policy in a vocabulary that is more familiar than permissions. A policy created using the Action-Centric Model can still be used by Java's current access control system by simply translating the filled actions into a set of permissions. Assuming the action definitions, rule definitions, and translation templates are defined correctly, the policy that results from the permissions will be identical to the one modeled using filled actions. This assumption highlights one of the key topics for extension. The definition file defines all resources, action definitions, rule definitions, and translations between filled actions and filled rules. By changing the definition file, new sets of action definitions can be introduced providing new ways to implement policy. Translations can be changed to make filled actions represent different sets of filled rules, which will represent different sets of permissions. The use of rules instead of regular permissions gives the model the ability to be independent of the underlying security implementation. Because of this abstraction, it is theoretically possible for the Action-Centric Model to support almost any access control system, though this project focused solely on Java's Permission-based system.

4.2.1 Soundness of Model Translation

A few features of the translation functions in the Action-Centric Model are worth mentioning. First, if a given set of filled rules is translated into filled actions and back again, the exact same set of filled rules will result. If a given set of filled actions is translated into filled rules, there may

be more filled actions produced by the translation back to filled actions, as mentioned in 3.1.1. Repeated translations will be stable from this point forward. More formally, given the translation function α from filled actions to filled rules, and the translation function β from filled rules to filled actions:

- $\beta : \alpha = id_{rules}$
- $\alpha : \beta : \alpha : \beta = \alpha : id_{rules} : \beta = \alpha : \beta$

4.3 Future Work

Extensions to this project can be divided into two main categories: extensions to the Action-Centric Model, and extensions to the NetBeans plug-in. A few possible extensions of each type are presented here.

4.3.1 Action-Centric Model Extensions

Parameterized Resources

The main reason the Action-Centric Model is called "Action Centric" and not "Resource Centric" is because there are no meaningful parameters on the Resource class, and the notion of an "instance of a unique resource" is not modeled. Creating instances of unique resources with attached actions may allow more flexibility in the model. For instance, with unique resources, it would be easier to find all actions associated with a unique resource.

For example, to model a file resource with a read and a write action under the current version of the model requires two actions be created, one for read and one for write. The file parameter is the same on both actions. To model the same authorization under a Resource Centric model, one resource would be created using the appropriate file path, then the read and write actions would be associated with that unique resource. The benefit in the resource centric view is that unique resources are identified instead of having many actions that are not directly related. There is a small workaround built into the model that allows action definitions to specify a grouping parameter. For example, file actions are grouped by file path. In this way, the plug-in's user interface knows to group all files with the same path under one node. This makes the file resource more apparent to the user without changing the focus of the model to resources.

Actions Containing Actions

There is currently no support in the model to have actions defined as a group of other actions. Only rules may be used when defining an action's contents. If actions could be composed of other actions, the action definitions would be much simpler to maintain, and provide much more information about applications' security requirements.

4.3.2 NetBeans Plug-In Extensions

Integration with Static Analysis

Due to time constraints, this project was not able to integrate static analysis results into the plug-in's user interface. In order to use the results within NetBeans, the following steps would be taken:

1. Run static analysis on project code within NetBeans. This is actually harder than it sounds. There is a significant amount of class-path configuration that needs to be handed to Soot before the static analysis will run properly.

2. Use the Action-Centric Model to translate static analysis results (a collection of PermissionDescriptions) into a CodeGroup object. This step is trivial; all the code to do it is in place and working in the import functionality.

3. Add the CodeGroup to the plug-in's repository so that it can be seen in all the standard views.

Use During Application Deployment

Ideally, the plug-in should be extended to generate a working policy file and inject it into the deliverable when the application is being packaged. Currently, the policy file that is generated is not a standard Java policy file, though it would not take significant effort to make it. There is no facility to package files generated by the plug-in into the application deliverable.

Storage of Policies

In order to further simplify the process of developing policy, it may be helpful to store a set of standard policies, which could be packaged with the plug-in or could be created by the user. These policies could then be quickly attached to NetBeans projects. This approach would make the task of maintaining several projects that share a common policy much simpler and encourage reuse of previously written policies.

User Interface Improvements

The user interface for the NetBeans plug-in was still under heavy development at the conclusion of the project. Many refinements are outstanding, including:

- Friendly parameter input. For example, when defining a new action on a file, the "file name" field should not be a text field; it should be a file chooser or some other form of easy entry.
- Difference highlighting in comparison view
- Ability to hide the rules pane

Create a Definition Editor

An intuitive definition editor could provide a powerful mechanism for developers to specify different kinds of policy for particular domains. Currently, any changes to the definitions of resources, actions, and rules must be done manually in XML. This is tedious, and discourages use of the model.

Appendix A - Static Analysis Example

This appendix describes a sample run of the static analysis procedure.

The test classes under analysis are shown here:

```
public class CallGraphs {
  //Entry point
  public static void main(String[] args) {
    test();
  }

  public static void test() {
    TestClass a = new TestClass();
    a.foo();
  }
}

class TestClass {

  public void foo() {
    FilePermission fp = new FilePermission("/File1", "read");

    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      sm.checkPermission(fp);
    }
    bar();
  }

  public void bar() {
    anotherPermission();
    File f = new File("/File2");

    f.exists();
  }

  public void anotherPermission() {
    RuntimePermission rp = new RuntimePermission("ARuntimePermission", null);

    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
```

```
      sm.checkPermission(rp);
    }

    MyThread mt = new MyThread();

    threadPasser1(mt);
  }

  public void threadPasser1(MyThread mt) {
    threadPasser2(mt);
  }

  public void threadPasser2(MyThread mt) {
    mt.start();
  }
}

class MyThread extends Thread {
  public void run() {
    FilePermission f = new FilePermission("/File3", "read,write");

    SecurityManager sm = System.getSecurityManager();
    if (sm != null) {
      sm.checkPermission(f);
    }
  }
}
```

A lot of results are produced, mostly because of the unaddressed virtual call problem in library code causing many permissions to be found at that level. A sample of some output:

```
Permissions required for method with signature:
<test.CallGraphs: void main(java.lang.String[])>
  java.lang.RuntimePermission([charsetProvider])
  java.lang.RuntimePermission([accessDeclaredMembers])
  java.security.SecurityPermission([???])
  java.security.SecurityPermission([getPolicy])
  java.lang.RuntimePermission([???])
  java.lang.RuntimePermission([setContextClassLoader])
  java.lang.RuntimePermission([getClassLoader])
  java.security.Permission([])
  java.lang.RuntimePermission([modifyThreadGroup])
  java.lang.RuntimePermission([exitVM])
  java.lang.RuntimePermission([setSecurityManager])
  java.util.PropertyPermission([???, read])
  java.lang.RuntimePermission([createSecurityManager])
  java.lang.RuntimePermission([modifyThread])
  java.lang.RuntimePermission([enableContextClassLoaderOverride])
  javax.security.auth.PrivateCredentialPermission([???, ???])
  java.lang.RuntimePermission([getClassLoader])
  java.util.PropertyPermission([user.timezone, write])
  java.io.FilePermission([/File3, read,write])
  java.security.Permission([suppressAccessChecks])
```

```
java.net.SocketPermission([???, connect])
java.lang.RuntimePermission([getProtectionDomain])
java.security.SecurityPermission([???])
java.util.PropertyPermission([*, read,write])
java.util.PropertyPermission([???, write])
java.lang.RuntimePermission([getClassLoader])
java.lang.RuntimePermission([getClassLoader])
java.io.FilePermission([???, read])
java.io.FilePermission([/File1, read])
java.lang.RuntimePermission([ARuntimePermission, ???])
javax.security.auth.PrivateCredentialPermission([???, ???])
java.security.SecurityPermission([getPolicy])
java.net.SocketPermission([???, resolve])
java.lang.RuntimePermission([getClassLoader])
java.net.NetPermission([specifyStreamHandler])
```

So the analysis returns far more permissions than are actually required and explicitly checked by the classes listed. In particular, our analysis was only interested in 4 out of 35 of the Permissions that were listed. The exact cause of this in the call graph seems to come from the combined effects of calls through the static initializer methods of classes like Object, System, and SecurityManager. Further analysis of exactly where all the permissions come from would be the best way to improve the accuracy of this analysis.

The relevant ones for the classes shown are:

1. java.io.FilePermission([/File1, read])
2. java.io.FilePermission([???, read])
3. java.io.FilePermission([/File3, read,write])
4. java.lang.RuntimePermission([ARuntimePermission, ???])

So the analysis reported that main() requires all of the permissions created in the test classes.

For these permissions, the following permissions result for each method (an X denotes that the permission as numbered above is required):

Method	1	2	3	4
main()	X	X	X	X
test()	X	X	X	X
foo()	X	X	X	X
bar()	-	X	X	X
anotherPermission()	-	-	X	X
threadPasser1()	-	-	-	-
threadPasser2()	-	-	-	-
run()	-	-	X	-

There are a few important things to notice here.

1. Permissions propagate upward as expected, from bar() to foo() to test(), etc. This shows that permissions found at certain levels are in fact added to the methods "above" them in the call graph as the traversal works its way backward.
2. The FilePermission with the unknown (???) first argument, but known "read" parameter, is comes from within the File.exists() method. File.exists() calls SecurityManager.checkRead(), which looks like this:

```
public void checkRead(String file) {
  checkPermission(new FilePermission(file,
    SecurityConstants.FILE_READ_ACTION));
}
```

This illustrates the parameter problem with the data flow analysis, since File.exists() extracts the name of the file and passes it to checkRead. The analysis does go and find the actual String constant represented by the constant and report it, however. A more in depth interprocedural data flow analysis would likely be able to determine the value of this string in the case shown here.

3. The two threadPasser methods do *not* report those found required by MyThread.run(). This is because of how Threads work in a security context. The context that created the Thread object (anotherPermission()) is the context above that of MyThread.run() as far as the AccessController is concerned.

Appendix B - NetBeans Plug-In Guide

This section provides an introduction to the NetBeans Plug-In developed to showcase the Action-Centric Model. Installation instructions can be found in the tool distribution archives.

The plug-in is started by selecting "Security Analysis Tool" from NetBeans' Tools menu. After that, the user is presented with screen shown in Figure 4.1. A definition file should be selected that conforms to the XML standards outlined in 3.1.2. A "policy configuration" is a set of actions and rules that are grouped together into a unified policy. An existing policy configuration can be loaded, or a new one can be created. The name of the policy configuration can also be specified.

Once the starting dialog is accepted, the user is presented with the main pane, seen in Figure 4.2. This screen presents all the major functionalities of the plug-in, including: adding and removing rules and actions, importing and exporting policies, and comparing policies.

To add an action, first expand the appropriate resource. Then select the appropriate action and fill in all the parameters. An example is shown in Figure 4.3.

Once the action is added, the main screen is updated as shown in 4.4. Notice that the action has been added, and also all the rules that would be needed to create the action in Java permissions. Adding rules works similarly to this process.

If actions are added that can be grouped by parameters (as described in 3.2.1 and 3.1.2) they are grouped properly under the appropriate resource type, as shown in Figure 4.5.

Using the "Diff" button on the main screen, we can compare two policies, as demonstrated in Figure 4.6. Using the "Add" buttons, actions and rules can be copied between policies.

The plug-in also demonstrates how unmatched rules (see 3.1.1) work. Take this example - in Figure 4.7, we see a single action with all the appropriate rules to support it. By removing one of the rules, we expect no actions, and 2 unmatched rules. This is demonstrated in Figure 4.8. Notice the unmatched rules are highlighted in red.

The import and export buttons work as expected, as demonstrated in Figures 4.9 and 4.10, respectively. At the time of this writing, the policies are imported and exported from a simple text file that is not directly readable by any other application.

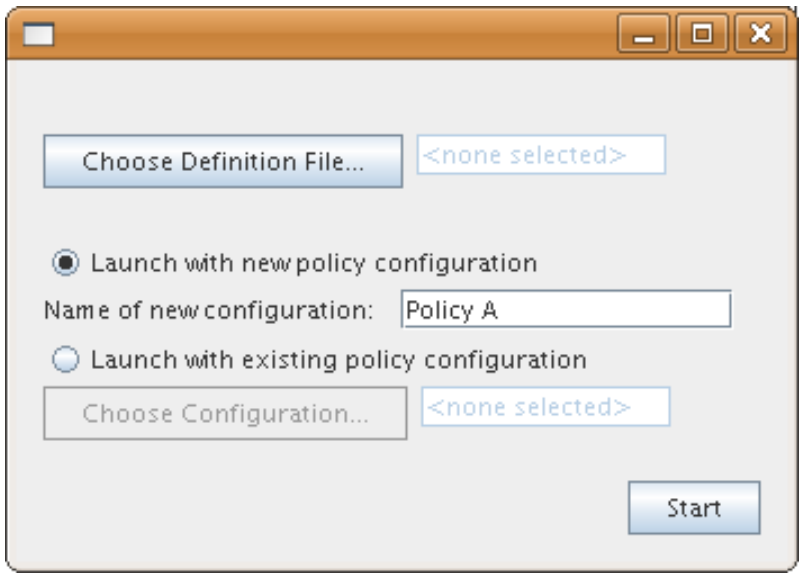

Figure 4.1: Plug-In startup screen

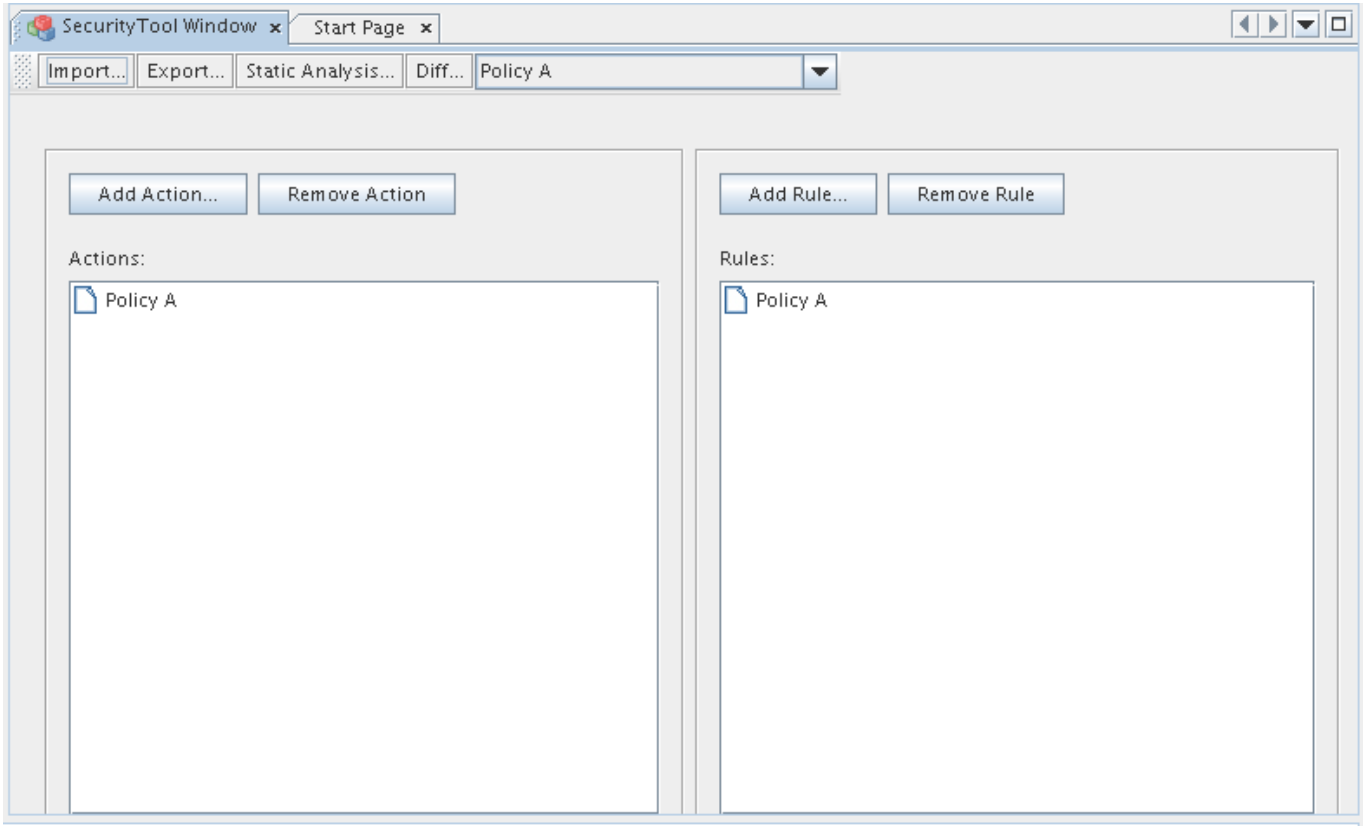

Figure 4.2: Plug-In main screen

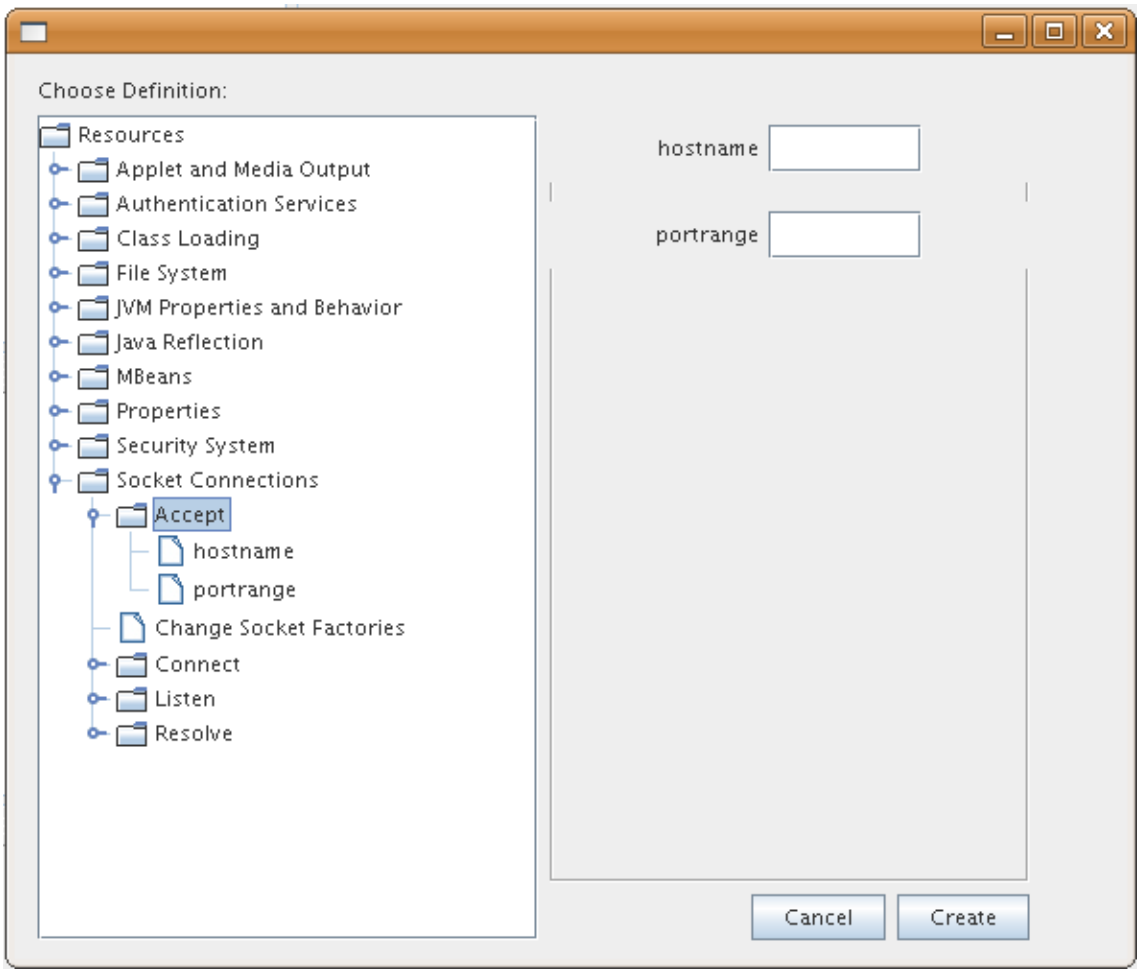

Figure 4.3: Adding a simple action

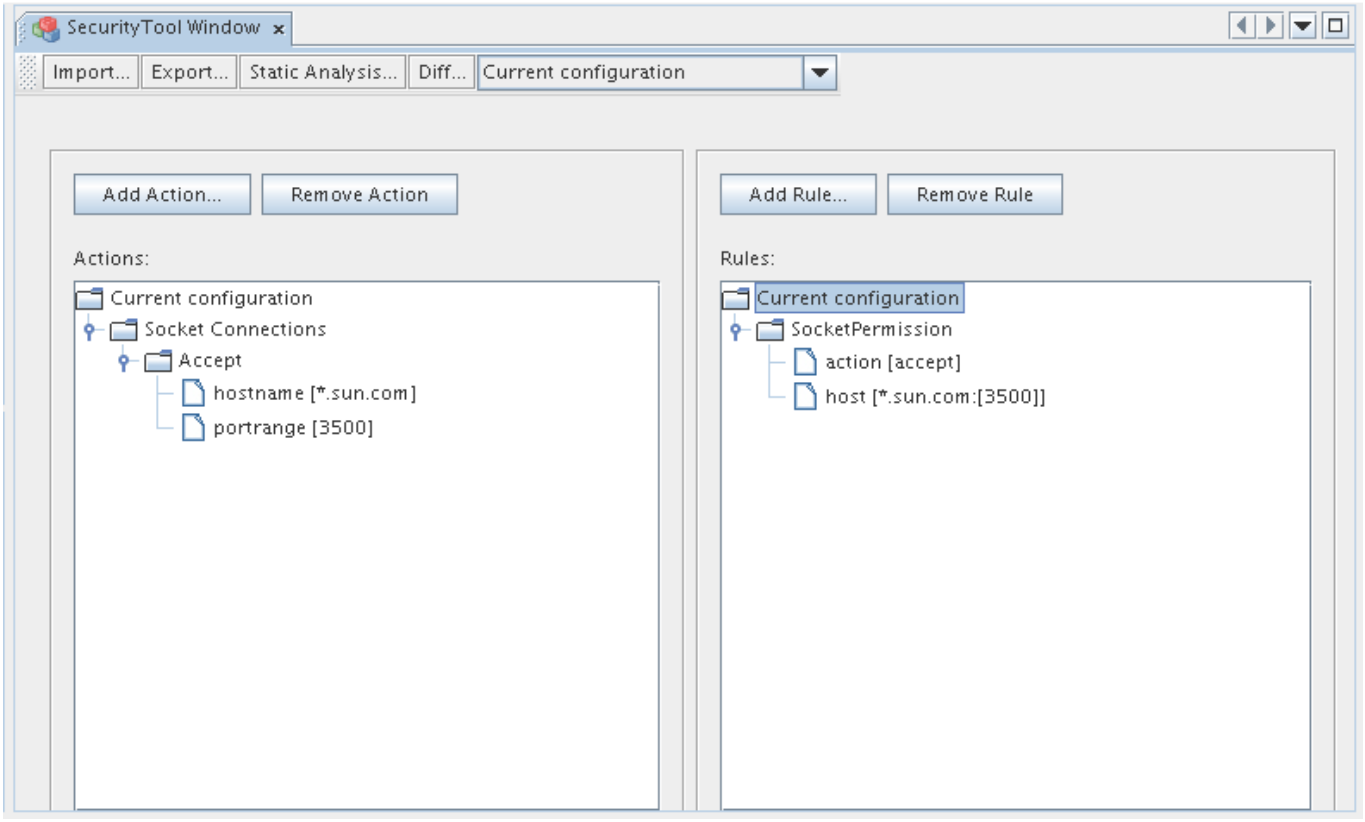

Figure 4.4: Plug-In main screen with actions and rules

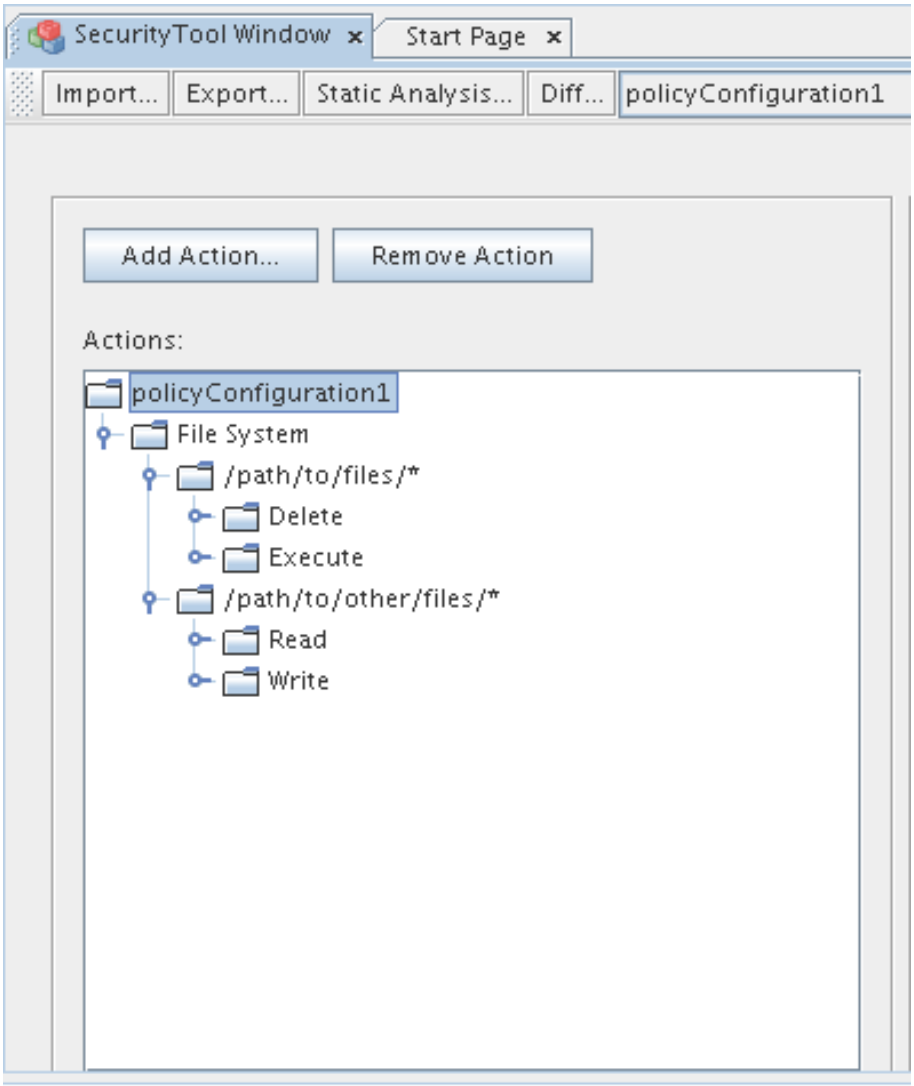

Figure 4.5: Grouping demonstration

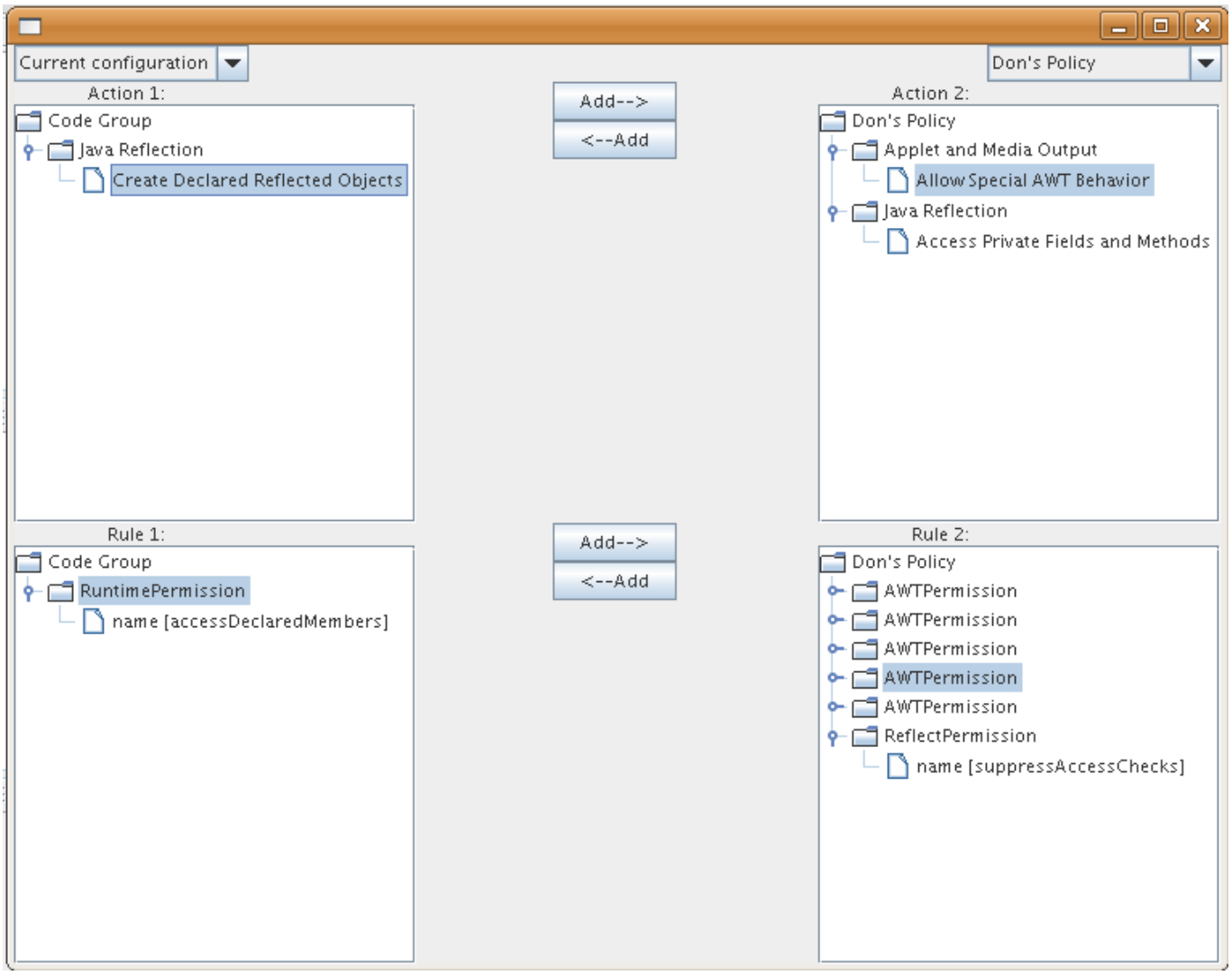

Figure 4.6: Diff view

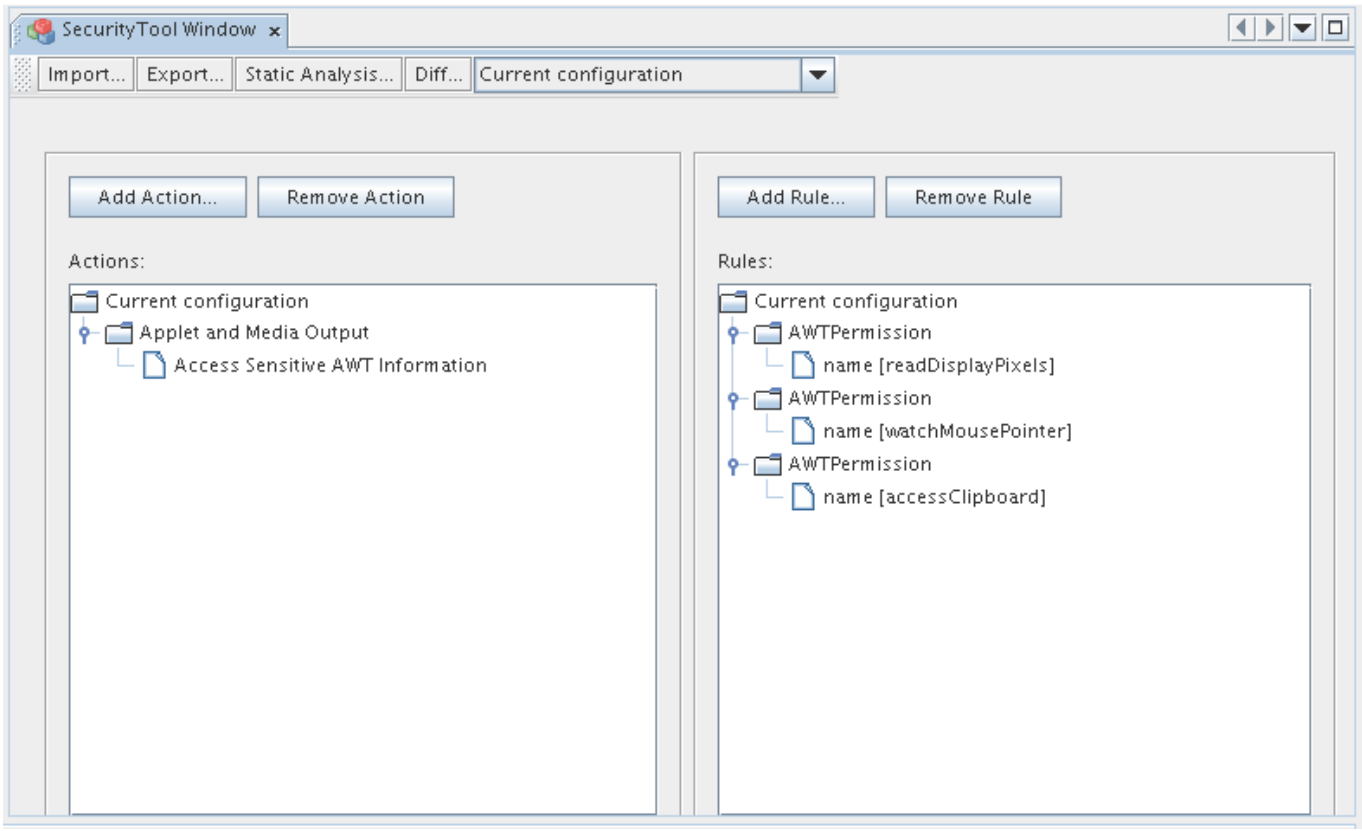

Figure 4.7: Unmatched rules demonstration - all matched

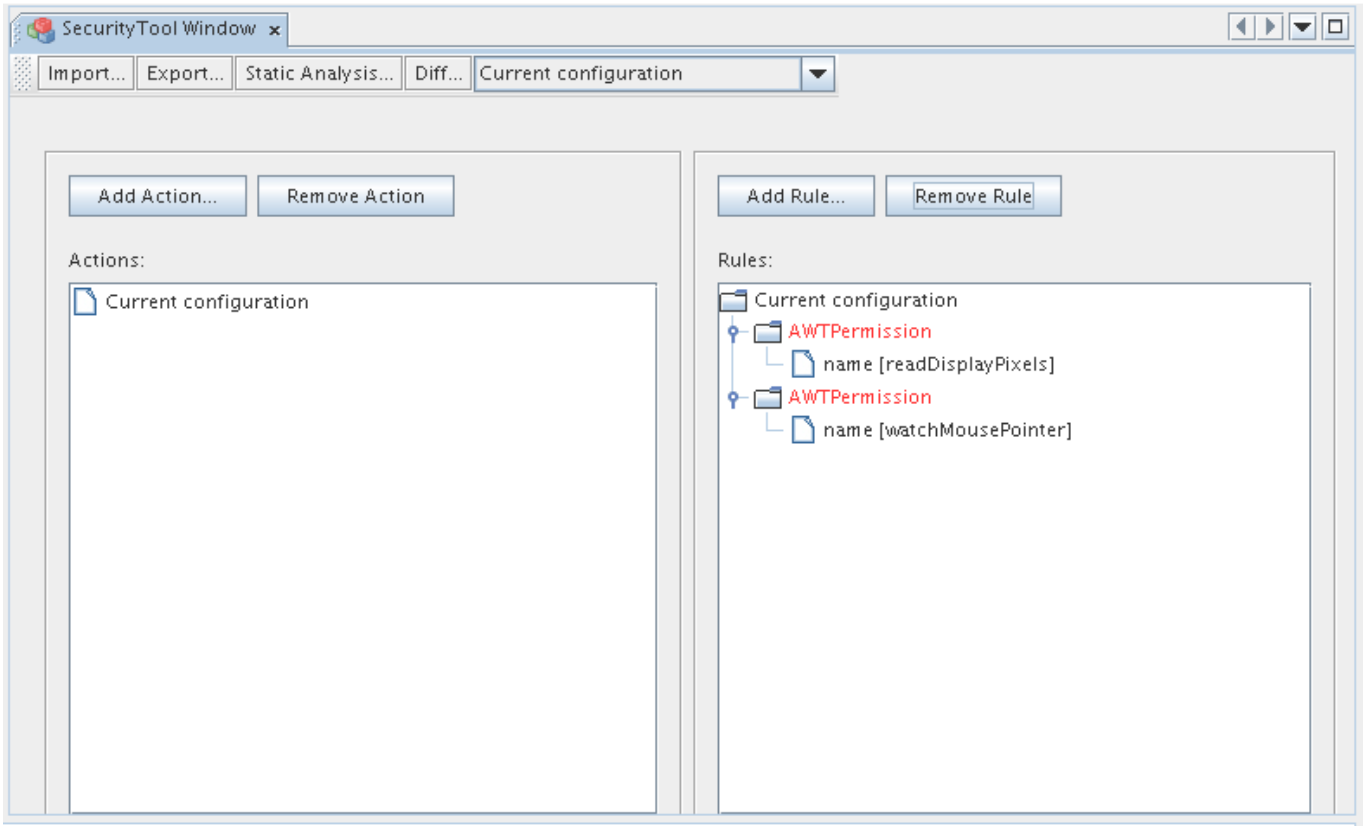

Figure 4.8: Unmatched rules demonstration - some unmatched

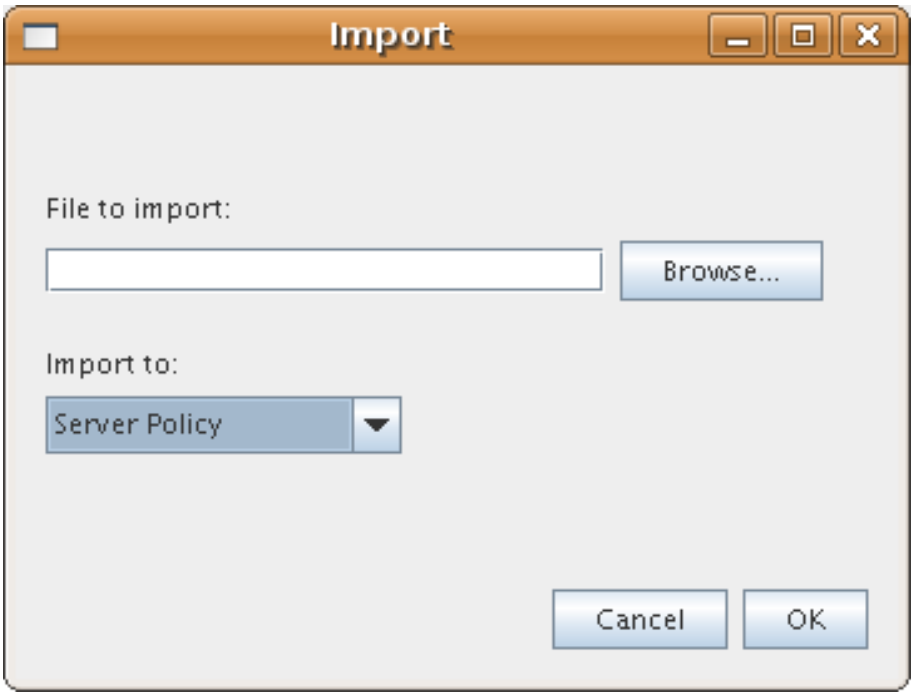

Figure 4.9: Policy Import

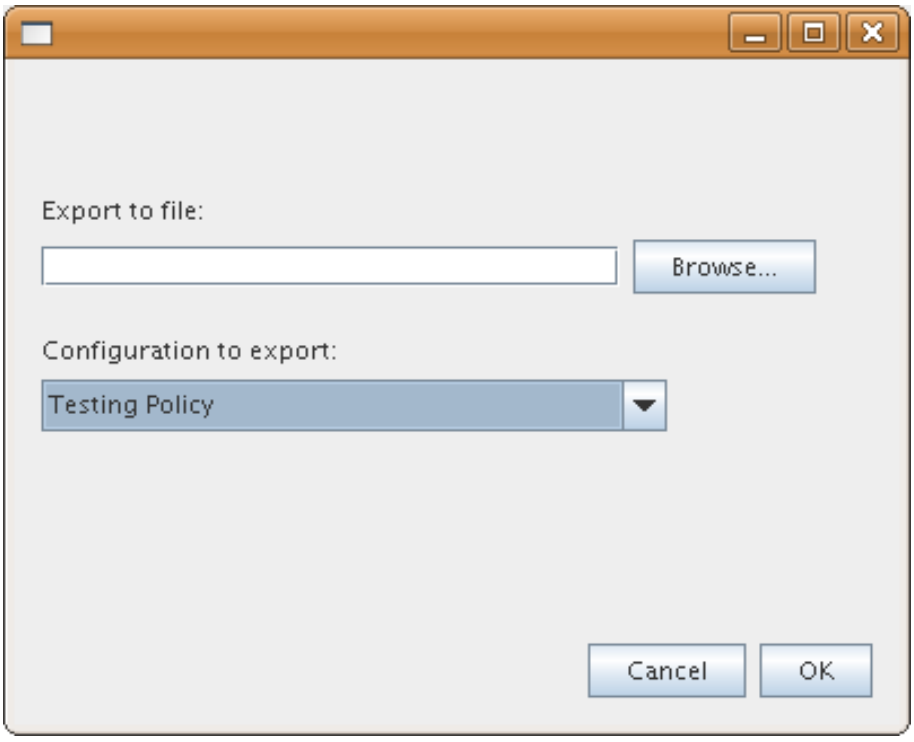

Figure 4.10: Policy Export

Appendix C - XML Specification of File System Actions

Here we show how the File System resource is defined in the Action-Centric Model.

A few things to note:

- The "groupBy" tag allows these actions to be grouped by that path (so all File Read, Write, etc. actions could be shown that share the path field).
- There are no parameters specified for the Resource itself. Parameterizing an instance of File System with a path would allow the actions to be specified without the path field if they inherited it from the Resource.
- The "Read File and Load Library" action, which was created mostly for testing. It allowed us to test a fairly simple case of needing to translate back into multiple action parameters

```
<resource name="File System">
  <action name="Read">
      <actionParameters>
          <actionParameter name="path" groupBy="true">
              <description>
                        The path to the desired file or directory.
              </description>
          </actionParameter>
      </actionParameters>
      <appliedRules>
          <appliedRule type="Permission" name="FilePermission">
              <parameterApplications>
                  <parameterApplication name="path"
                                value="{path}" />
                  <parameterApplication name="action"
                                value="read" />
              </parameterApplications>
          </appliedRule>
      </appliedRules>
  </action>
  <action name="Read and Load Library">
      <actionParameters>
          <actionParameter name="path" groupBy="true">
              <description>
                        The path to the desired file or directory.
              </description>
```

```
            </actionParameter>
            <actionParameter name="libraryName">
            </actionParameter>
        </actionParameters>
        <appliedRules>
            <appliedRule type="Permission" name="FilePermission">
                <parameterApplications>
                    <parameterApplication name="path"
                                value="{path}" />
                    <parameterApplication name="action"
                                value="read" />
                </parameterApplications>
            </appliedRule>
            <appliedRule type="Permission" name="RuntimePermission">
                <parameterApplications>
                    <parameterApplication name="name"
                                value="loadLibrary.{libraryName}" />
                </parameterApplications>
            </appliedRule>
        </appliedRules>
    </action>
    <action name="Write">
        <actionParameters>
            <actionParameter name="path" groupBy="true">
                <description>
                        The path to the desired file or directory.
                </description>
            </actionParameter>
        </actionParameters>
        <appliedRules>
            <appliedRule type="Permission" name="FilePermission">
                <parameterApplications>
                    <parameterApplication name="path"
                                value="{path}" />
                    <parameterApplication name="action"
                                value="write" />
                </parameterApplications>
            </appliedRule>
        </appliedRules>
    </action>
    <action name="Execute">
        <actionParameters>
            <actionParameter name="path" groupBy="true">
                <description>
                        The path to the desired file or directory.
                </description>
            </actionParameter>
        </actionParameters>
        <appliedRules>
            <appliedRule type="Permission" name="FilePermission">
                <parameterApplications>
                    <parameterApplication name="path"
```

```
                                value="{path}" />
                    <parameterApplication name="action"
                                value="execute" />
                </parameterApplications>
            </appliedRule>
        </appliedRules>
    </action>
    <action name="Delete">
        <actionParameters>
            <actionParameter name="path" groupBy="true">
                <description>
                        The path to the desired file or directory.
                </description>
            </actionParameter>
        </actionParameters>
        <appliedRules>
            <appliedRule type="Permission" name="FilePermission">
                <parameterApplications>
                    <parameterApplication name="path"
                                value="{path}" />
                    <parameterApplication name="action"
                                value="delete" />
                </parameterApplications>
            </appliedRule>
        </appliedRules>
    </action>
</resource>
```

Bibliography

[1] Policy tool -policy file creation and management tool. http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/policytool.html, 2002.

[2] Security workbench development environment for java. http://www.alphaworks.ibm.com/tech/sword4j, 2005.

[3] Maven - security annotation framework. http://safr.sourceforge.net/index.html, 2008.

[4] Anindya Banerjee, David A. Naumann, Anindya Banerjee A, and David A. Naumann B. A simple semantics and static analysis for java security. Technical report, 2001.

[5] Massimo Bartoletti, Pierpaolo Degano, and Gianluigi Ferrari. Static analysis for stack inspection. In *In ConCoord: International Workshop on Concurrency and Coordination, volume 54 of ENTCS*, page 2001. Elsevier, 2001.

[6] Arni Einarsson and Janus Dam Nielson. *A Survivor's Guide to Java Program Analysis with Soot*. BRICS, Department of Computer Science. University of Aarhus, Denmark., July 2008.

[7] Li Gong. Java 2 platform security architecture. http://java.sun.com/javase/6/docs/technotes/guides/security/spec/spec.doc.html, 2002.

[8] David Grove and Craig Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6):685–746, 2001.

[9] Mika Koganeyama, Naoshi Tabuchi, and Takaaki Tateishi. Reducing unnecessary conservativeness in access rights analysis with string analysis. In *APSEC '07: Proceedings of the 14th Asia-Pacific Software Engineering Conference*, pages 438–445, Washington, DC, USA, 2007. IEEE Computer Society.

[10] Larry Koved, Marco Pistoia, and Aaron Kershenbaum. Access rights analysis for java. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 359–372, New York, NY, USA, 2002. ACM.

[11] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *SSYM'05: Proceedings of the 14th conference on USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association.

[12] Marianne Mueller and Roland Schemers. Socketpermission javadoc. 2003.

[13] Vijay Sundaresan Patrick Lam Etienne Gagnon Raja Vallée-Rai, Laurie Hendren and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[14] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[15] Vijay Sundaresan. Practical techniques for virtual call resolution in java. Technical report, McGill University, 1999.