A Language and System for Composing Security Policies[*]

Lujo Bauer[†] Jay Ligatti[‡] David Walker[§]

Princeton University Technical Report TR-699-04

January 2004

Abstract

We introduce a new language and system that allows security architects to develop well-structured and easy-to-maintain security policies for Java applications. In our system, policies are first-class objects. Consequently, programmers can define parameterized meta-policies that act as policy combinators and policy modifiers, so that complex security policies can be implemented by composing simple base policies. We demonstrate the effectiveness of our design by building up a library of powerful policy combinators and showing how they can be used. We also describe some issues we encountered while implementing our system and provide performance results.

1 Introduction

Security architects for large software systems face an enormous challenge: the larger and more complex their system, the more difficult it is to ensure that it obeys some security policy. Like any large software problem, the security problem can only be dealt with by breaking it down into smaller and more manageable pieces. These smaller-sized problems are easier to understand and reason about, and their solutions are simpler to implement and verify.

When decomposing the security problem into parts, it is tempting to scatter access-control checks, resource-monitoring code, and other mechanisms across

[*]This research was supported in part by DARPA award F30602-99-1-0519, ARDA Grant no. NBCHC030106, National Science Foundation CAREER grant No. CCR-0238328, and by the Army Research Office through grant no. DAAD19-02-1-0389 (*Perpetually Available and Secure Information Systems*) to the CyLab at Carnegie Mellon University; this work does not necessarily reflect the opinion or policy of the federal government and no official endorsement should be inferred.

[†]Carnegie Mellon University, `lbauer@ece.cmu.edu`

[‡]Princeton University, `jligatti@cs.princeton.edu`

[§]Princeton University, `dpw@cs.princeton.edu` the many modules that implement these components. This is especially true when the enforcement of some property involves several low-level components drawn from otherwise logically different parts of the system. For instance, in order to implement a policy concerning data privacy, it may be necessary to consider the operation of a wide variety of system components including the file system and the network, as well as printers and other forms of I/O. Unfortunately, a scattered implementation of a policy is much more difficult to understand and verify than a centralized implementation—even finding all the pieces of a distributed policy can be problematic. Moreover, the distribution of the security policy and mechanism through a large body of code can make it more difficult to react to security breaches and vulnerabilities. In the current security climate, where new viruses can spread across the Internet in minutes, speedy reaction to vulnerabilities is critical.

This paper describes Polymer, a new language and system that helps engineers enforce centralized security policies at run time by monitoring and modifying an untrusted Java application's behavior. Our design focuses on providing features that allow programmer to develop well-structured and easy-to-maintain security policies. It facilitates the decomposition of complex security policies into smaller, simpler parts, provides a convenient way to write the simple component policies, and supplies tools to combine these components into more complex and powerful policies.

The Polymer language is designed as a minimal extension to Java to make it easy for Java programmers to learn and to develop security policies for their applications. Programmers implement security policies by extending Polymer's `Policy` class, which is given a special interpretation by the underlying run-time system. Intuitively, it is best to understand `Policy` objects, which we also call *program monitors*, as state machines that operate over the sequence of actions executed by an untrusted target application. More concretely, each `Policy` object contains three elements:

- A specification of the application *actions* (i.e., method calls) relevant to security.
- Any necessary security state, which can be used to keep track of the application's activity during execution.
- A decision procedure that is invoked each time an application attempts to execute a security-sensitive action. The decision procedure returns one of a number of security *suggestions* that are interpreted by the underlying system. These suggestions include the suggestion to quietly *accept* and execute the action, to *suppress* the action and raise an exception inside the application, and to *halt* the application outright. In addition, a suggestion may contain effectful code to be executed on behalf of the application.

This structure is highly expressive and serves as an excellent starting point for an implementation. It is inspired by Schneider's automata-theoretic characterization of program monitors [21] and is derived more directly from our own previous theoretical work on *edit automata* [5, 18], which are formal state machines that transform application behavior by "editing" the sequence of application events through insertions, deletions, and truncation of the sequence. We have proven that in principle, these automata can enforce a very wide range of program properties.

In addition, since our program monitors are first-class Java objects, it is possible for simple policies to serve as arguments to policy combinators. We have carefully structured the `Policy` class and developed a programming methodology so that simple policies can be composed with each other in a variety of ways. We have demonstrated the effectiveness of our design by developing a library of combinators, including a couple different forms of "conjunctive" and "disjunctive" policies, as well as some application-specific policy combinators and policy modifiers. One of the major challenges is developing combinators that make sense in the presence of effects.

One other important aspect of our design is a pattern-matching facility that allows us to manipulate and query program actions using a convenient and concise notation. In addition to providing simple patterns that match ordinary, concrete program actions, we provide facilities reminiscent of Wadler's views [23] to match against *abstract actions*. Abstract actions summarize the security-relevant information that appears in concrete actions and present it in a standard form. Unlike views, which implement isomorphisms between two representations, our abstract actions only implement "half" a view, the injection into the abstract type. Abstract actions allow programmers to hide the irrelevant details of a complex low-level interface and to treat many methods with similar functionality (such as the dozen or so methods to open a file in the standard Java libraries) as identical for security purposes. Overall, abstract actions make it possible to modularize policy design along another axis, orthogonal to the modularization provided by our combinators.

The rest of this paper proceeds as follows. Section 2 explains the overall process involved in building and executing secure programs using Polymer. Section 3 describes the main features of the language, including actions, suggestions, and policies. It also runs through a series of examples that illustrate how to write simple policies and policy combinators. Section 4 describes some of the key elements of our implementation and measures the performance of the main system components. Finally, Section 5 discusses the relationship between our work and previous work in this area.

2 Polymer System Overview

The Polymer system is composed of two main tools. The first tool is a policy compiler that compiles program monitors defined in Polymer the language into plain Java and then into Java bytecode. The second tool is a bytecode rewriting tool (RW) that processes ordinary bytecode, inserting calls to the monitor in all the necessary places. In order to construct a secure executable using these tools, programmers must perform the following series of steps.

1. Write the *action declaration file* (ADF). This specification lists all possible program actions that might have an impact on system security. The ADF indicates to RW which methods require code insertions to transfer control from the application to the monitor.

2. Instrument the system libraries using RW. RW reads in the system libraries and inserts calls to monitoring code in all the places described by the ADF. This step may be performed before the the details of the security policy implementation are set and need not be repeated before executing a target application. The libraries must be instrumented before the JVM starts up since the default JVM security constraints prevent many libraries from being modified or reloaded once the JVM is running.

3. Write the security policy and compile it with the policy compiler. The policy compiler translates the Polymer program into ordinary Java and then invokes a Java compiler to translate it to bytecode.

4. Start the JVM with the modified libraries.

5. Load the target application. During this loading, our specialized class loader uses RW to rewrite the target code in the same way we rewrote the library code in step 2.

6. Execute the secured application.

Figure 1 contains a picture of the end result of the process. In this picture, instrumented target and library code run inside the JVM. Whenever this code is about to invoke a security-sensitive method, control is redirected through a generic policy manager, which queries the current policy. The current policy will return a suggestion that is interpreted by the policy manager. It is possible to inline the policy manager into the application code, and doing so might result in a performance improvement. However, we have chosen to maintain this level of indirection to make it possible to dynamically update policies without bringing the virtual machine down. We have not experimented extensively with dynamic policy updates yet, but in principle, this decision makes it possible to react to security vulnerabilities quickly, even in servers and other long-running applications. Moreover, the security updates can be completely invisible to users who, if threatened with interruptions in service, might choose not to comply with security recommendations.

3 Polymer Language

In this section, we describe the core features of the Polymer language. We begin with the basic concepts and show how to program simple policies. Then, we demonstrate how to create a more complete policy by applying policy combinators to our simple base policies.

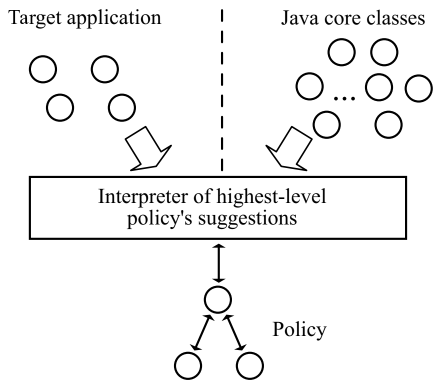

Figure 1: A secure Polymer application

3.1 Core Concepts

Polymer introduces three central new abstractions: actions, suggestions, and policies. Policies analyze actions and convey their decisions by means of suggestions.

Actions Monitors intercept and reason about how to react to security-sensitive method invocations. `Action` objects contain all of the information relevant to such invocations: static information such as the method signature, and dynamic information like the calling object and the method's parameters.

For convenient manipulation of actions, Polymer allows them to be matched against *action patterns* (Figure 2). An `Action` object matches an action pattern when the action's signature matches the one specified in the pattern. Patterns can also use wildcards: `*` matches any one constraint (e.g., any return type or any single parameter type), and `..` matches zero or more parameter types. For example, the pattern

```
<public void java.io.*.<init>(int, ..)>
```

matches all public constructors in all classes in the `java.io` package whose first parameter is an `int`. In place of `<init>`, which refers to a constructor, we could have used an identifier that refers to a particular method.

Action patterns appear in three places. First, the action declaration file (ADF) is a set of action patterns. During the instrumentation process, every action that matches an action pattern in the ADF is instrumented. Second, every monitor uses action patterns to declare the set of actions that it regulates. An individual monitor only ever has to reason about actions in this set. Third, monitors use action patterns in `aswitch` statements to determine which security-sensitive action they are dealing with. `aswitch` statements are similar to Java's `switch` statements.

```
aswitch(a) {
  case <void System.exit(int status)>: E;
  ...
}
```

If `Action a` represents an invocation of `System.exit`, this statement evaluates expression `E` with the variable `status` bound to the the value of the method's single parameter.

ActPat:	< [*Mod*] *RetTy* [*ID* =] *Name* **(** *Params* **)** >
Mod:	**public** | **protected** | **package** | **private**
	| **abs** | *WildCard*
RetTy:	**void** | *Type* | *WildCard*
Name:	((*ID* | *WildCard*) **.**)+ (<**init**> | *ID* | *WildCard*)
Params:	**..** | *Param* [**,** *Params*]
Param:	*WildCard* | *Type* [*ID*]
WildCard:	*****
ID:	Any Java identifier
Type:	Any Java type

Figure 2: Syntax of action patterns. Square brackets around an item indicate at most one occurrence of that item, while a plus indicates one or more occurrences. Categories are in an italic font; literals are bold. The abs modifier is described in Section 3.3.

Suggestions Whenever the untrusted application attempts to execute a security-relevant action, the monitor suggests a way to handle this action (which we often call a *trigger action* because it triggers the monitor into making such a suggestion). There are four main types of suggestions a monitor can generate. If the target is attempting to execute an action that could not possibly violate the monitor's security policy, the monitor suggests that this action be quietly accepted. If, on the other hand, the action causes (or may cause) a policy violation, the monitor has three basic choices. It may halt the target altogether; it may ignore the target's request but allow the target to continue executing; or it may do any of the above after executing some auxiliary code that attempts to recover from the potential violation or performs bookkeeping or other tasks.

The monitor's decision about a particular trigger action is conveyed using a `Suggestion` object. Polymer supplies a subclass of `Suggestion` for each type of suggestion listed above:

- An `OKSuggestion` suggests the trigger action should execute unconditionally.

- A `HaltSuggestion` suggests the action should not be executed and the target should be halted.

```
public abstract class Policy {
  public abstract ActionSet getActions();
  public abstract Suggestion query(Action a);
  public void finish(Suggestion sug,
    Object result, boolean wasExnThn) { };
}
```

Figure 3: The parent class of all policies

- A `SupSuggestion` suggests the action should not be executed but the target should be allowed to continue. Whenever following a `SupSuggestion`, Polymer notifies the target that its attempt at invoking the trigger action has been denied by throwing a `SuppressException` that the target can catch before continuing execution.

- An `InsSuggestion` suggests that making a final decision about the target action be deferred until after some auxiliary code is executed and its effects are evaluated.

We have found that breaking down the possible actions of a monitor into these four orthogonal categories provides great flexibility and also makes monitor semantics relatively easy to understand. In addition, this breakdown simplifies our job tremendously when it comes to controlling monitor effects and building combinators that put monitors together in sensible ways (see section 3.4).

By default, any actions declared security sensitive by a monitor will be processed by that monitor, even if these actions are executed by other collaborating monitors or the monitor itself.[1] In other words, by default, our system adheres to the principal of *complete mediation*, which states that every access to a resource should be monitored. This design helps to prevent situations in which a malicious program fools a privileged security component into executing unmonitored, yet security-sensitive, actions on its behalf. In addition, this design makes it straightforward to write monitors that monitor the activity of other monitors.

Policies Programmers encode a run-time monitor in Polymer by extending the base `Policy` class (Figure 3). A new policy must provide implementations of the `getActions` and `query` methods, and may optionally override the `finish` method.

- `getActions` returns the set of actions the policy considers security-sensitive; any action that is not an element of this set cannot trigger the monitor.

- `query` analyzes a trigger action and returns a suggestion indicating how to deal with it.

[1]If desired, it is possible to change this behavior and guard against its misuse by defining the right sort of policy combinator.

- `finish` is called after a policy's suggestion is followed. If a suggestion forces the target to be halted or a `SuppressException` to be raised, `finish` is called immediately before halting the target or raising the exception. The three arguments to `finish` are the original suggestion the policy returned, the return value of the trigger action or inserted action (this will be `null` if no action was executed or its return type was void, and will be an `Exception` value if the action completed abnormally), and a flag indicating whether the action completed abnormally.

The existence of `query` and `finish` methods in policies is fundamental to the design of Polymer. We can compose policies by writing policy combinators that `query` other policies and combine their suggestions. In combining suggestions, a combinator may choose not to follow the suggestions of some of the queried policies. Thus, `query` methods must not assume that their suggestions will be followed and should be free of effects such as state updates and I/O operations. All policies should obey the design principle that `query` methods be pure. There are two places where effects may occur: an effectful computation can be encapsulated in an insertion suggestion, or an effectful computation can execute in the `finish` method. Updates to the policy state should generally happen in the `finish` method because only at this point can a policy be sure that its suggestion has been executed successfully. This design provides fine-grained control over exactly when effects occur and makes it possible to build powerful policy combinators with precise and comprehensible semantics.

3.2 Simple Policies

To give a feel for how to write Polymer policies, we define four simple examples in this section; in Section 3.4, we will build more powerful policies by combining the basic policies presented here using a collection of policy combinators.

We begin by considering the most permissive policy possible: one that allows everything. The Polymer code for this policy is given in Figure 4. The `getActions` method in `AllPolicy` returns a set of actions constructed from a one-element array of action patterns. The single element in this array matches every action; therefore, `AllPolicy` considers every action security-relevant. Because the `query` method of `AllPolicy` always returns an `OKSuggestion`, it allows all trigger actions to execute unconditionally. To enable convenient processing of suggestions, every `Suggestion` constructor has at least one argument, the `Policy` making the `Suggestion`.

For our second example, we consider a slightly more useful policy that only allows network connections to hosts on the same local network as the monitored system. Specifically, our policy examines method calls that create network sockets. Whenever the policy is queried concerning the creation of a `Socket` object, it invokes a utility function to determine whether the application is creating a local connection or not. If it is, the policy allows the connection to proceed; otherwise, the policy considers the attempt at remote connection an intolerable security violation and halts the target's execution by returning a `HaltSuggestion`. The

```
public class AllPolicy extends Policy {
  private ActionSet actions =
    new ActionSet(new ActionPattern[ ] { <* *.*(..)> });
  public ActionSet getActions() { return actions; }
  public Suggestion query(Action a)
    { return new OKSuggestion(this); }
}
```

Figure 4: Policy that allows everything

```
public class OnlyLocalHosts extends Policy {
  private ActionSet actions =
    new ActionSet(new ActionPattern[ ] {
      <void java.net.Socket.<init>(String, ..)> });
  public ActionSet getActions() { return actions; }
  public Suggestion query(Action a) {
    aswitch(a) {
      case <void java.net.Socket.<init>(String s, ..)>:
        if(AddrUtilities.isHostLocal(s))
          return new OKSuggestion(this);
    }
    return new HaltSuggestion(this);
  }
}
```

Figure 5: Policy that allows network connections only to hosts on the local network

Polymer code to implement this policy is shown in Figure 5.

Our third policy (shown in Figure 6), attempts to prevent a simple denial-of-service attack by restricting the number of files that an untrusted target may have open at any time. This policy examines calls that open and close files while maintaining a variable that contains a count of the number of files the target has open. It allows all trigger actions to proceed normally except for the opening of a file that would send the count of open files over its limit (which is set in the policy's constructor). In this disallowed case, the monitor signals to the target that its attempted file opening creates a security violation by returning a `SupSuggestion`. After an allowed trigger action has been successfully executed, the `finish` method updates the `numFilesOpen` variable appropriately. Trigger actions are passed to the `Suggestion` constructors in the `query` method so that the `finish` method has access to them.

Finally, we consider a policy that requires a warning to be displayed before creating the first network socket after having opened a file. The Polymer code for this policy is shown in Figure 7. `WarnBeforeConnect` inserts an action to display a warning whenever a socket is created after a file has been opened. The inserted `Action` is constructed by supplying a caller (in this case,

```
public class LimitFilesOpen extends Policy {
  private int maxOpen, numFilesOpen = 0;
  private ActionSet actions =
    new ActionSet(new ActionPattern[ ] {
      <* java.io.FileInputStream.<init>(String)>,
      <* java.io.FileOutputStream.<init>(String, ..)>,
      <* java.io.FileInputStream.close()>,
      <* java.io.FileOutputStream.close()> });
  public ActionSet getActions() { return actions; }
  public LimitFilesOpen(int maxOpen)
    { this.maxOpen = maxOpen; }
  public Suggestion query(Action a) {
    aswitch(a) {
      case <* *.close()>:
        return new OKSuggestion(this, a);
    }
    if(numFilesOpen < maxOpen)
      return new OKSuggestion(this, a);
    return new SupSuggestion(this, a);
  }
  public void finish(Suggestion sug, Object result,
                     boolean wasExnThn) {
    if (wasExnThn || sug.isSuppress()) return;
    aswitch(sug.getTrigger()) {
      case <* *.close()>: numFilesOpen--; return;
      default: numFilesOpen++;
    }
  }
}
```

Figure 6: Policy that limits the number of open files

`null`, because the inserted action is static), a fully qualified method name (here, `examples.Warnings.warn`), and parameters to the inserted method packed in an array of `Object`s. After the inserted action is executed, the policy will again be asked to decide whether opening a socket is allowed; since this second attempt will occur after a warning has been displayed, the policy will simply allow the socket to be opened.

3.3 Abstract Actions

Thus far we have assumed that a target application will open files only by calling particular `FileInputStream` and `FileOutputStream` constructors. In practice, several other methods also open files; for example, other `FileInputStream` and `FileOutputStream` constructors, `RandomAccessFile` constructors, `File.createNewFile` and `File.createTempFile` methods, `java.util.zip.ZipFile` constructors, and `Runtime` methods for executing system commands. It can be

```
public class WarnBeforeConnect extends Policy {
  private boolean haveFilesBeenOpened = false;
  private boolean haveWarned = false;
  private ActionSet actions =
    new ActionSet(new ActionPattern[ ] {
      <* java.io.FileInputStream.<init>(String)>,
      <* java.io.FileOutputStream.<init>(String, ..)>,
      <* java.net.Socket.<init>(String, ..)> });
  public ActionSet getActions() { return actions; }
  public Suggestion query(Action a) {
    aswitch(a) {
      case <* java.net.Socket.<init>(String s, ..)>:
        if(haveFilesBeenOpened && !haveWarned) {
          Action insAct = new Action(null,
            "examples.Warnings.warn", new Object[ ]{
            "Connecting to "+s+" after opening files"});
          return new InsSuggestion(this, a, insAct);
        } //else fall through
      case <* java.io.FileInputStream.<init>(*)>:
      case <* java.io.FileOutputStream.<init>(..)>:
        return new OKSuggestion(this, a);
    }
    return new HaltSuggestion(this, a);
  }
  public void finish(Suggestion sug, Object result,
                     boolean wasExnThn) {
    if(wasExnThn || sug==null) return;
    if(sug.getTrigger().isInsertion()) haveWarned=true;
    aswitch(sug.getTrigger()) {
      case <* java.io.FileInputStream.<init>(*)>:
      case <* java.io.FileOutputStream.<init>(..)>:
        haveFilesBeenOpened = true;
    }
  }
}
```

Figure 7: Policy that displays a warning when a socket is created after files have been opened

cumbersome and redundant to have to enumerate all these methods in a policy, so Polymer makes it possible to group them into *abstract actions*. Abstract actions enhance the power of action patterns; for example, instead of using a series of action patterns to capture each of the possible actions that will open a file, a policy can use a single action pattern that references the abstract `FileOpen` action.

Abstract actions allow a policy to reason about security-relevant actions at a different level of granularity than is offered by the Java core API. They permit policies to focus on regulating particular behaviors, say, the opening of files,

```
public class FileOpen extends AbsAction {
  public String param1; //The name of the file to be opened
  public boolean matches(Action a) {
    aswitch(a) {
      case <boolean java.io.File.createNewFile()>:
        param1 = ((java.io.File)(a.getCaller())).getName();
        return true;
      ...
      case <void java.util.zip.ZipFile.<init>(String name)>:
        param1 = name; return true;
    }
    return false;
  }
}
```

Figure 8: FileOpen abstract action definition

rather than forcing them to individually regulate each of the actions that cause this behavior. This makes it easier to write more concise, modular policies. The use of abstract actions also makes it possible to write platform-independent policies. The set of actions that cause a file to be opened may not be the same on every system, but as long as the abstract `FileOpen` action is adjusted accordingly, the same policy for regulating file access can be used everywhere.

To illustrate the usefulness of abstract actions, we will use them to simplify the `WarnBeforeConnect` policy (Figure 7). First, however, we need to define abstract actions for opening files and network connections. An abbreviated definition of the `FileOpen` abstract action is shown in Figure 8. The `NetSockOpen` abstract action is defined similarly.

An abstract action has two main tasks: one is to determine whether it encompasses a particular concrete action; the other is to provide a consistent interface to all the concrete actions it represents.

To solve the first task, each abstract action has to implement a `matches` method. This method takes as its argument a particular concrete action and returns `true` if the action is one of the abstract action's constituents. An abstract action may not match every instance of a particular concrete action. For example, a `nonTmpFileOpen` abstract action may represent the attempt to open any file not in the `/tmp` directory. The `matches` method has access to the concrete action's run-time parameters and is thus able to see whether the file being opened is in the `/tmp` directory and make its decision accordingly.

The second task of an abstract action is to present a policy with a consistent interface to the set of concrete actions. The signature of the `FileOpen` abstract action, for example, could be written as `void FileOpen(String filename)`. Not all methods that open a file take a `String` as an argument, however. `File.createNewFile` takes no arguments; to make it fit the abstract action signature, the `matches` method must compute the name of the file being opened. Each of the abstract action's parameters is represented by a field (`param1`,

```
public class WarnBeforeConnect extends Policy {
  private ActionSet actions =
    new ActionSet(new ActionPattern[ ] {
      <abs * examples.FileOpen(String)>,
      <abs * examples.NetSockOpen(String, ..)> });
  public Suggestion query(Action a) {
    aswitch(a) {
      case <abs * examples.NetSockOpen(String s, ..)>:
        if(haveFilesBeenOpened && !haveWarned) {
          Action insAct = new Action(null,
            "examples.Warnings.warn", new Object[ ]{
            "Connecting to "+s+" after opening files"});
          return new InsSuggestion(this, a, insAct);
        } //else fall through
      case <abs * examples.FileOpen(String)>:
        return new OKSuggestion(this, a);
    }
    return new HaltSuggestion(this, a);
  }
  ...
}
```

Figure 9: Version of `WarnBeforeConnect` policy that uses abstract actions

`param2`, etc.). Once the matches method determines the value of the parameter (in this case, the name of the file) it assigns it to the appropriate field (`param1`).

Policies use abstract actions just as they do concrete actions, except that abstract action patterns are prefaced by the keyword `abs`. Figure 9 shows the `WarnBeforeConnect` policy, now modified to use abstract actions.

3.4 Policy Combinators

Polymer supports policy modularity and code reuse by allowing policies to be combined with and modified by other policies. In Polymer, a policy is a first-class Java object, so it may serve as an argument to or result of other policies. We call a policy parameterized by other policies a *policy combinator*. When referring to a complex policy with many policy parts, we call the policy parts *subpolicies* and the complex policy a *superpolicy*. We have written a library of common combinators; however, security policy architects are always free to develop new combinators to suit their own specific needs.

Conjunctive combinator It is often useful to restrict an application's behavior by applying several policies at once and enforcing the most restrictive one. For example, a policy that disallows access to files can be used in combination with a policy that disallows access to the network; the resulting policy disallows access to both files and the network. In the general case, the policies being conjoined may reason about overlapping sets of actions. When this is the case, we must consider what to do when the two subpolicies suggest different courses of action. In addition, we must define the order in which effectful computations are performed.

Our conjunctive combinator composes exactly two policies; we can generalize this to any number of subpolicies. Our combinator operates as follows.

- If either subpolicy suggest insertions, so does the combinator, with any insertions by the left-most (first) conjunct occurring prior to insertions by the right-most conjunct. Following the principle of complete mediation, the monitor will recursively examine these inserted actions if they are security-relevant.

- If neither subpolicy suggests insertions, the monitor follows the more restrictive suggestion. Halting is more restrictive than suppressing, which is in turn more restrictive than accepting.

Notice that a sequence of insertions made by one conjunct may have the effect of changing the state of a second conjunct. In fact, that is quite likely if the second conjunct considers the inserted actions security-relevant. Consequently, the second conjunct may make a different suggestion regarding how to handle an action before the insertions than it does after. For example, in the initial state, the action might have been OK, but after the intervening insertions, the monitor might suggest that the application be halted.

An abbreviated version of the conjunctive combinator is shown in Figure 10. The calls to `SugUtils.getNewSug` in the query method simply create new suggestions with the same type as the first parameter in these calls. Notice that the suggestion returned by the combinator includes the suggestions on which the combinator based its decision. This design makes it possible for the combinator's `finish` method to notify the appropriate subpolicies that their suggestions have been followed.

We can use the conjunctive combinator to instantiate a policy that both prevents foreign code from opening more than three files and displays a warning if a network connection is being opened after any files have been accessed. The parameters passed to the combinator are exactly the policies described previously (in Section 3.2).

```
Policy LimitAndWarn = new Conjunction(
  new LimitFilesOpen(3), new WarnBeforeConnect())
```

Precedence combinators We have found the conjunctive policy to be the most common combinator. However, it is useful on occasion to have a combinator that gives precedence to one subpolicy over another. One example is the `TryWith` combinator, which queries its first subpolicy, and if that subpolicy returns an `OKSuggestion` or an `InsSuggestion`, it makes the same suggestion. Otherwise, the combinator defers judgment to the second subpolicy.

```
public class Conjunction extends Policy {
  private Policy p1, p2;
  private ActionSet actions;
  public Conjunction(Policy p1, Policy p2) {
    this.p1 = p1; this.p2 = p2;
    actions = new ActionSet(p1.getActions(),
                            p2.getActions());
  }
  public ActionSet getActions() { return actions; }
  public Suggestion query(Action a) {
    Suggestion s1=p1.query(a), s2=p2.query(a);
    if(s1.isInsertion()) return SugUtils.getNewSug(
      s1, this, a, new Suggestion[ ]{s1});
    if(s2.isInsertion()) return SugUtils.getNewSug(
      s2, this, a, new Suggestion[ ]{s2});
    if(s1.isHalt() && s2.isHalt())
      return SugUtils.getNewSug(s1, this, a,
        new Suggestion[ ]{s1,s2});
    if(s1.isHalt()) return SugUtils.getNewSug(
      s1, this, a, new Suggestion[ ]{s1});
    ...
  }
  public void finish(Suggestion sug, Object result,
                     boolean wasExnThn) {
    //notify subpolicies whose advice was followed
    Suggestion[ ] sa = sug.getSuggestions();
    for(int i = 0; i < sa.length; i++) {
      sa[i].getSuggestingPolicy().finish(
        sa[i], result, wasExnThn);
    }
  }
}
```

Figure 10: Conjunctive policy combinator

Suppose, for example, that we want to exempt local hosts from the `LimitAndWarn` policy. One way to implement such a policy is to edit `LimitAndWarn` directly. If `LimitAndWarn`, however, is a useful policy in our library, we may not wish to change its semantics, as this will impact other policies. Instead, we can apply a `TryWith` combinator that combines the `OnlyLocalHosts` policy with the `LimitAndWarn` policy. The overall effect of this combination is to first determine whether the trigger action is a connection to a local host. If so, the action is accepted because `OnlyLocalHosts` accepts it. Otherwise, the `LimitAndWarn` policy takes over. Here is the code to create this policy.

```
Policy LocalOrLimitAndWarn = new TryWith(
  new OnlyLocalHosts(), LimitAndWarn)
```

A similar sort of combinator is the `Dominates` combinator, which always follows the suggestion of the first conjunct if that conjunct considers the trigger action security-relevant. Otherwise, it follows the suggestion of the second conjunct.

Selectors *Selectors* are combinators that choose to enforce exactly one of their subpolicies. There are two classes of selectors: delayed selectors, which initially consult both their subpolicies but eventually start using only one; and immediate selectors, which use the first trigger action to decide which subpolicy to enforce.

A delayed selector can be used, for example, to combine a policy that allows access only to particular file and GUI methods with a policy that allows access only to particular network and GUI methods. The resulting policy permits an application to run unhindered as long as it executes only GUI methods allowed by both subpolicies. If the application invokes an action permitted by one subpolicy but not the other, though, the combinator then chooses only to ever obey the subpolicy that allows this action.

Immediate selectors are useful when it is possible to determine without delay which of the subpolicies should be enforced. We may want to enforce the `LocalOrLimitAndWarn` policy, for example, on all applications that do not carry a trusted digital signature. We can accomplish this by applying an immediate selector to the `LocalOrLimitAndWarn` and `AllPolicy` policies. The selector's job is to verify whether the target application is signed and then enforce the appropriate policy.

```
Policy AllowSigned = new SignedSelector(
  new AllPolicy(), LocalOrLimitAndWarn)
```

Selectors are reminiscent of the Chinese Wall security policy [7]. Initially in a Chinese Wall, one can access many resources, but as some resources are accessed, others become unavailable. Here, when a selector chooses one subpolicy, the others become unavailable.

Unary combinators Unary combinators enforce a single policy while also performing some other actions. Suppose, for example, that we want to enforce the `AllowSigned` policy while logging the actions of the target application and the monitor. Figure 11 shows the hierarchy of subpolicies that is created by applying a `LogEverything` unary combinator to the `AllowSigned` policy. `LogEverything` handles trigger actions by blindly suggesting whatever `AllowSigned` suggests; `LogEverything`'s `finish` method simply performs a logging operation before invoking the `finish` method of `AllowSigned`.

Another unary combinator is one that controls a policy's ability to monitor itself and other policies. In some circumstances, self-monitoring policies can cause loops that will prevent the target program from continuing (for example, a policy might react to an action by inserting that same action, which the policy will then see and react to in the same way again). It is easy to write a unary combinator to prevent such loops.

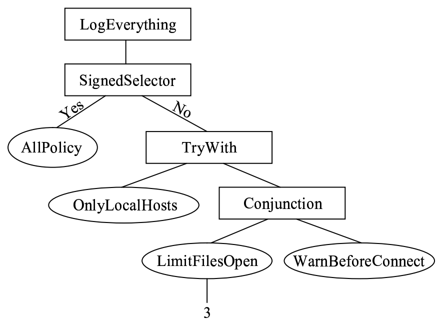

Figure 11: A foreign code policy hierarchy. Combinators are shown in rectangles and base policies in ovals.

4 Implementation

The principal requirement for enforcing the run-time policies we are interested in is that the flow of control of a running program passes to a monitor whenever a security-relevant method is about to be executed. Two strategies for diverting control flow can accomplish this: diverting at call sites, the locations where security-relevant methods are invoked; or diverting at call targets, the actual method bodies of the security-relevant methods.

Diverting the control flow at call sites at first seems the more straightforward approach. On the surface, it would seem to involve only rewriting the target application's bytecode so that security-relevant method invocations are intercepted by the monitor. The code comprising a running application, however, also includes all the class libraries that the application uses. A security-relevant method could be invoked from many places in this code, so it is potentially necessary to instrument a large number of call sites. More importantly, in order to determine which call sites to rewrite, we must perform a relatively sophisticated static analysis and, in general, need to determine the precise method that is invoked dynamically. The reason for the dynamic checks is that the decision to monitor a method should be based on the behavior of the method, but since subclasses have arbitrarily different behavior, it is not sufficient to instrument based on the static type of an object. Consequently, it is simpler and more efficient to divert control flow within the method body of each security-relevant method.

Security mechanisms that rely on rewriting must ensure that all possible ways of invoking a security-relevant method are guarded. If there is even a single way of sidestepping the added guards, no claims can be made about the security of the application. Tools that rewrite Java bytecode in order to make it more secure sometimes struggle to prevent such sidestepping in the presence of the Java reflection mechanism. Reflection allows, for example, methods to be invoked at runtime even if their names do not appear in bytecode. It is therefore very difficult to statically ensure that the application will behave correctly without disabling reflection entirely, which is rarely acceptable in practice. Polymer does not suffer from this shortfall. In fact, we can write a Polymer policy that prevents a target application from using reflection, custom class loaders, or other methods to circumvent Polymer. Since this policy can analyze these methods at runtime, it is able to disallow calls that would breach security while allowing legal ones. This policy can be viewed as a unary superpolicy that imposes restrictions on certain reflective calls but otherwise follows the suggestions of its subpolicy.

The kind of pre- and post-invocation control-flow modification to bytecode that we use to implement Polymer can be done by tools like AspectJ [15]. Accordingly, we considered using AspectJ to insert into bytecode hooks that would trigger our monitor as needed. However, we wanted to retain precise control over how and where rewriting occurs to be able to make decisions in the best interests of security, which is not the primary focus of aspect-oriented languages like AspectJ. Instead, we used the Apache BCEL API [3] to develop our own bytecode rewriting tool.

4.1 Obstacles and Limitations

Custom class loaders have often been used to modify bytecode before executing it [2, 4]; we use this technique also. Ideally, we would force untrusted applications to be loaded using a custom class loader that would modify bytecode as necessary to consult the monitor before executing security-relevant methods. Standard Java VMs (such as Sun's), however, load a number of classes (e.g., `java.io.File` and `java.io.BufferedInputStream`) before loading a client application. These classes are therefore out of reach of a custom class loader; since many of them (such as `java.io.FileOutputStream`) could be considered security relevant, this is unacceptable.

Several strategies could be employed to combat this problem: we can integrate our class loader more tightly with the Java VM, in effect adding functionality to the system class loader; we could rename all the core Java classes used by a target application to ensure that they are loaded by our class loader; or we can preprocess the core Java classes, thus ensuring that even the classes loaded by the system class loader have the hooks needed to invoke the monitor. The first approach might result in a cleaner overall system, but would be more complicated to implement and obviously would not be portable across JVMs. The second approach, renaming any core classes used by the target application, has the advantages of simplicity and portability, but unfortunately it does not work. Many classes, like `java.lang.Object` and `java.lang.Throwable`, are treated specially by standard VMs and cannot be renamed; since they are loaded by the system class loader, they also cannot be modified. Because these classes, along with any they reference (which includes, for example, `java.io.FilterOutputStream` and `java.io.PrintWriter`), cannot be renamed, this implementation strategy prevents the enforcement of many useful policies. The third approach, rewriting the core classes in a preprocessing phase, is simple, effective, portable, and hence our choice.

An additional hurdle is that some just-in-time (JIT) compilers assume that particular methods perform particular functions. The JIT compiler in Sun's Java 1.4 SDK, for example, assumes that `Math.sqrt` calculates the square root, and produces native code for calculating square roots without actually referring to the body of the `Math.sqrt` method. Since we divert control to the monitor by instrumenting the bodies of security relevant methods, this optimization interacts poorly with our implementation. To regulate such methods, we would have to instrument bytecode at the call site; fortunately, these methods are very few, and, like `Math.sqrt`, unlikely to be of concern to security policy writers.

Similar difficulties occur with a few other methods that are treated specially by Sun's, and probably other, VMs. Sun's VM will not initialize if the constructor of the `Object` class, for example, has been modified. It also resists modifications to `Object.finalize`, `Object.toString`, and `Class.newInstance`. Regulating these methods, therefore, is outside the scope of our current implementation.

4.2 Performance

It is instructive to examine the performance costs of enforcing policies using Polymer. We did not concentrate on making our implementation as efficient as possible, so there is much room for improvement here. However, the performance of our implementation does shed some light on the costs of run-time policy enforcement.

Our system impacts target applications in two phases: before and during loading, when the application and the class libraries are instrumented by the bytecode rewriter RW; and during execution. We evaluate each phase separately.[2]

Instrumentation Because Polymer instruments core Java class libraries separately from non-library code, we measured the performance of these tasks independently. The total time to instrument every method in all of the standard Java library packages (i.e., the 28742 methods in the 3948 classes in the `java` and `javax` packages of Sun's Java API v.1.4.0) was 107 s, or 3.7 ms per instrumented method. This cost is reasonable because library instrumentation only needs to be performed once (rather than every time a target application is executed). The average time to load non-library classes into the JVM with our specialized class loader, but without instrumenting any methods, was 12ms, twice as long as the VM's default class loader required. In addition to this per-class loading penalty, instrumenting every method in a target class containing 100 methods took 520 ms, or 5.2 ms per method instrumented. The reason for

[2]The tests were performed on a Dell PowerEdge 2650 with dual Intel Xeon 2.2 GHz CPUs and 1 GB of RAM, running RedHat Linux 9.0. The times represent real time at low average load. We performed each test multiple times in sets of 100. The results shown are the average for the set with the lowest average, after removing outliers.

the higher per-method cost in this test, as compared to the library instrumentation test, is that the target methods contained relatively many parameters, and the instrumenter has to loop through parameter lists to insert code that wraps arguments of primitive types into `Object`s so that they can be sent to the monitor uniformly. These instrumentation timings form a useful basis for understanding the approximate overhead of Polymer's bytecode instrumentation process; there was an approximately 6 ms penalty per class for using the Polymer class loader, in addition to about 4.5 ms per method instrumented.

Execution The run-time cost of enforcing policies with Polymer falls into three categories: (1) the cost of passing control to Polymer before and after processing the body of a security-relevant method, (2) the cost of determining whether the security-relevant action belongs to the set of actions regulated by the top-level policy, and (3) the cost of reasoning within a policy about whether a given method should be allowed or disallowed and potentially taking corrective action. While (1) is a static cost incurred even if the active policy does not reason about the current trigger action, both (2) and (3) clearly depend upon the details of the policy being enforced.

We tested the impact of regulating the ability to open a socket via the `java.net.ServerSocket(int)` constructor. The results reveal chiefly the static costs (1), since we tested with only the simplest policies. The base time to open a socket when Polymer was not involved (class libraries and the target application were not instrumented, so Polymer was never consulted) was 35.74 ms. When class libraries and the application were instrumented (so that control passed to Polymer), but the active policy was not regulating any actions, the same operation required 36.36 ms. With the class libraries instrumented and `AllPolicy` as the active policy, opening the socket took 38.64 ms.

Passing control to Polymer requires a couple of extra method calls. Evaluating the action takes longer, but this cost depends entirely upon the complexity of the policy being enforced. Compared to the relatively long execution time of the `ServerSocket(int)` constructor, the overhead of using Polymer in this test was low. In other situations, the method being instrumented might be much faster (the `java.io.FileOutputStream(String)` constructor, for example, executes in roughly a millisecond), so the proportion of time spent on enforcing the security policy would be greater.

5 Related Work

Safe language platforms, such as the Java Virtual Machine (JVM) [19] and Common Language Runtime (CLR) [20], use stack inspection as the basis of their program monitoring systems. Unfortunately, while stack inspection can be effective in many situations, it has some serious drawbacks as well. First, stack inspection is just one algorithm for implementing access control, and, as explained by several researchers [21, 11, 13, 1], this algorithm is inherently partial: A more complete security system would make decisions based on the entire history of a computation and *all* the code that has had an impact on the current system state, not just the current control stack. A second important flaw in the stack inspection model is that operations to enable privileges and perform access-control checks are scattered throughout the system libraries. Consequently, in order to understand the policy that is being enforced, one must read through arbitrary amounts of library code.

A number of researchers [12, 11, 10, 16, 17, 8] have attempted to improve software security by developing domain-specific languages for specifying security properties. The primary point of contact between our work and these others is the idea that it is possible to centralize a security policy by including specifications of *where* security checks should be placed in addition to *what* the security checks should be. Our main new contributions with respect to these other projects are the insights that (1) security monitors may have a variety of responses (accept, suppress, insert, or halt) to application events, and (2) security monitors may be elevated to first-class status and explicitly manipulated and composed. We demonstrate the utility of these ideas by showing how to develop well-structured security policies built from a library of monitor combinators.

Naccio [12], one of the domain-specific security policy languages that inspired our work, does allow policies to be broken down into smaller parts, but the mechanism by which these parts are put together does not appear to be fully specified. Since these policy parts usually have effects, the semantics of the overall policy is unclear. In his thesis [10], Erlingsson develops another security policy language that provides C-like “include statements” so that policies may be broken up into multiple files. He specifies that computational effects will occur in the order that they appear in the text of the overall policy, once the inclusions have been processed. This approach eliminates the semantic ambiguity, but does not provide the power and flexibility of our language.

One other important difference between our language and much of this previous work on domain-specific security policy languages is that rather than building a completely new language, our Polymer extensions are *conservative* over the rest of Java. This design decision has a number of advantages. First, it gives programmers familiar with Java fewer concepts to learn, speeding the adoption path for the product and diminishing the likelihood of policy errors due to unfamiliarity with the language. Second, it ensures the policy language has the full power of the application language (and more). This power may be required to effectively manipulate and constrain an unruly application. Third, it reduces the chance that there will be a dangerous semantic disconnect between the policy language and the application language. Any semantic gap between policy language and application language semantics is a potential security hole.

Our monitor language can also be viewed as an aspect-oriented programming language (AOPL) in the style of AspectJ [15]. However, it is normally impossible to use a general-purpose aspect-oriented language for security since an untrusted application can use aspects to disrupt and subvert the security code. The main high-level difference between our work and previous AOPLs is the fact that our “aspects” (the program monitors) are first-class values and that we provide mechanisms to allow programmers to explicitly control the composition of aspects. Several researchers [22, 24, 9] describe functional, as opposed to object-oriented, AOPLs with first-class aspect-oriented advice. However, they do not support aspect combinators like the ones we have developed here. In general, composing aspects is a known problem for AOPLs, and we hope the ideas presented here, particularly the idea that aspects may be first-class, composable objects, will suggest a new design strategy for general-purpose AOPLs.

We currently have no direct language-theoretic model of Polymer the programming language. However, in previous work [6], we developed an idealistic calculus of functional program monitors with a built-in set of combinators. We are currently working on defining the formal semantics of Polymer as an extension of Featherweight Java [14].

Acknowledgments

We would like to thank Úlfar Erlingsson for explaining some of the difficulties in instrumenting Java library classes. We would also like to thank Dawn Song and Brent Waters for helpful comments about earlier drafts of this paper.

References

[1] M. Abadi and C. Fournet. Access control based on execution history. In *10th Annual Network and Distributed System Security Symposium*, 2003.

[2] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object Oriented Programing: Systems, Languages, and Applications (OOPSLA)*, Oct. 1997.

[3] Apache Software Foundation. *Byte Code Engineering Library*, 2003. `http://jakarta.apache.org/bcel/`.

[4] L. Bauer, A. W. Appel, and E. W. Felten. Mechanisms for secure modular programming in Java. *Software—Practice and Experience*, 33(5):461–480, 2003.

[5] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.

[6] L. Bauer, J. Ligatti, and D. Walker. Types and effects for non-interfering program monitors. In *International Symposium on Software Security*, Tokyo, Japan, 2002.

[7] D. Brewer and M. Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, May 1989.

[8] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Twenty-Seventh ACM Symposium on Principles of Programming Languages*, pages 54–66, Boston, Jan. 2000. ACM Press.

[9] D. Dantas and D. Walker. Aspects, information hiding and modularity. Submitted for publication, Nov. 2003.

[10] Ú. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement.* PhD thesis, Cornell University, Nov. 2003.

[11] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, Oakland, CA, May 2000.

[12] D. Evans and A. Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.

[13] C. Fournet and A. Gordon. Stack inspection: Theory and variants. In *Twenty-Ninth ACM Symposium on Principles of Programming Languages*, Jan. 2002.

[14] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java. In *ACM conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 132–146, Denver, CO, Aug. 1999.

[15] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.

[16] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.

[17] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan. Run-time assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, June 1999.

[18] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. Under consideration by the *International Journal of Information Security*. Submitted Dec 2002; Revised May 2003. Available as Princeton Computer Science Department TR-681-03.

[19] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[20] E. Meijer and J. Gough. A technical overview of the Common Language Infrastructure. `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`.

[21] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, Feb. 2000.

[22] D. B. Tucker and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, pages 158–167, 2003.

[23] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *Fourteenth ACM Symposium on Principles of Programming Languages*, pages 307–313. ACM Press, Jan. 1987.

[24] D. Walker, S. Zdancewic, and J. Ligatti. A theory of aspects. In *ACM International Conference on Functional Programming*, Uppsala, Sweden, Aug. 2003.