Composing Security Policies with Polymer

Lujo Bauer
Carnegie Mellon University
lbauer@cmu.edu

Jay Ligatti
Princeton University
jligatti@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

Abstract

We introduce a language and system that supports definition and composition of complex run-time security policies for Java applications. Our policies are comprised of two sorts of methods. The first is *query* methods that are called whenever an untrusted application tries to execute a security-sensitive action. A query method returns a *suggestion* indicating how the security-sensitive action should be handled. The second sort of methods are those that perform state updates as the policy's suggestions are followed.

The structure of our policies facilitates composition, as policies can query other policies for suggestions. In order to give programmers control over policy composition, we have designed the system so that policies, suggestions, and application events are all first-class objects that a higher-order policy may manipulate. We show how to use these programming features by developing a library of policy combinators.

Our system is fully implemented, and we have defined a formal semantics for an idealized subset of the language containing all of the key features. We demonstrate the effectiveness of our system by implementing a large-scale security policy for an email client.

Categories and Subject Descriptors D.3.2 [*Language Classification*]: Specialized application languages; D.2.1 [*Requirements/ Specifications*]: Languages; D.2.4 [*Software/Program Verification*]: Formal methods; D.3.1 [*Formal Definitions and Theory*]: Semantics, syntax; D.2.5 [*Testing and Debugging*]: Monitors

General Terms Languages, Security

Keywords Program monitors, run-time enforcement, composable security policies, edit automata, security automata

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 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 update a policy in response 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 on untrusted Java applicatons by monitoring and modifying the applications' behavior at run time. Programmers implement security policies by extending Polymer's `Policy` class, which is given a special interpretation by the underlying run-time system. Intuitively, each `Policy` object contains three main elements: (1) an effect-free decision procedure that determines how to react to security-sensitive application *actions* (i.e., method calls), (2) security state, which can be used to keep track of the application's activity during execution, and (3) methods to update the policy's security state.

We call the decision procedure mentioned above a *query* method. This method returns one of six *suggestions* indicating that: the action is *irrelevant* to the policy; the action is *OK*; the action should be reconsidered after some other code is *inserted*; the return value of the action should be *replaced* by a precomputed value; a security *exception* should be thrown instead of executing the action; or, the application should be *halted*. These objects are referred to as suggestions because there is no guarantee that the policy's desired reaction will occur when it is composed with other policies. Also for this reason, the query method should not have effects. State updates occur in other policy methods, which are invoked only when a policy's suggestion is followed.

In order to further support flexible but modular security policy programming, we treat all policies, suggestions, and application actions as first-class objects. Consequently, it is possible to define higher-order security policies that query one or more subordinate policies for their suggestions and then combine these suggestions in a semantically meaningful way, returning the overall result to the system, or other policies higher in the hierarchy. We facilitate programming with suggestions and application events by introducing pattern-matching facilities and mechanisms that allow programmers to summarize a collection of application events as an *abstract action*.

We have demonstrated the effectiveness of our design by developing a library of the most useful combinators, including a "con-

PLDI'05 June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006…$5.00.

junctive" policy that returns the most restrictive suggestion made by any subpolicy and a "dominates" policy that tries one policy first and, if that policy considers the application action irrelevant, then passes the application event on to the next policy. One of the major challenges here is developing a strategy that makes combining policies in the presence of effects semantically reasonable. In addition to our general-purpose policy combinators, we have developed a collection of application-specific policy combinators and policy modifiers, including a higher-order policy that dynamically checks for policy updates to load into the virtual machine and an audit policy that logs all actions of an untrusted application and all suggestions made by another policy acting on that application.

To test our language in a realistic setting, we have written a large-scale security policy, composed of smaller modular policies, for email clients that use the JavaMail interfaces. We have extensively tested this policy with the Pooka email client [16] and found that we can use Polymer to correctly enforce sophisticated security constraints.

1.1 Related Work

Safe language platforms, such as the Java Virtual Machine (JVM) [14] and Common Language Runtime (CLR) [15], use stack inspection as the basis of their security 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 [9, 18], this algorithm is inherently partial. More recent systems 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 [1, 6, 7, 8, 9, 11, 12]. 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.

Our current language and system are directly inspired by earlier theoretical research on automata-theoretic characterizations of security policies. Schneider [18] developed the notion of *security automata*, which are a form of Büchi automata that can recognize safety properties. We generalized this idea by defining *edit automata* [13], which are formal machines that transform a sequence of program actions via the operations of sequence truncation, insertion of new actions, or suppression of actions. The current research may be viewed as an implementation of edit automata with a practical set of "editing" capabilities and support for composition of automata.

The design and implementation of Polymer is most closely related to Evans and Twyman's Naccio [8] and to Erlingsson and Schneider's PoET/Pslang [7]. One of the crucial observations they make is that the entire security policy, including the set of security-relevant program points, should be defined separately from the main application. This architecture makes it is easier to understand, verify, and modify the security policy. The new contributions of our work include the following.

1. We have designed a new programming methodology that permits policies to be composed in meaningful and productive ways. A key innovation is the separation of a policy into an effectless method that generates suggestions (OK, halt, raise exception, etc.) and is safe to execute at any time, and effectful methods that update security state only under certain conditions.
2. We have written a library of first-class, higher-order policies and used them to build a large-scale, practical security policy that enforces a sophisticated set of constraints on untrusted email clients.
3. We have developed a formal semantics for an idealized version of our language that includes all of the key features of our implementation including first-class policies, suggestions, and application events. A formal semantics helps nail down corner cases and provides an unambiguous specification of how security policies execute—a crucial feature of any security mechanism, but particularly important as our security policies have imperative effects. We prove our language is type safe, a necessary property for protecting the program monitor from untrusted applications.

We also make a number of smaller contributions. For instance, unlike Naccio and PoET/Pslang, we allow a monitor to replace an entire invocation of a security-relevant action with a provided return value via a replace suggestion. Some policies, such as the `IncomingMail` policy in Section 3.2, require this capability. In addition, we faithfully implement the principle of *complete mediation* [17]. In other words, once a policy is put in place, every security-sensitive method is monitored by the policy every time it is executed, even if the method is called from another policy component. This has a performance cost, but it guarantees that every policy sees all method calls that are relevant to its decision. The details of our language, including its pattern-matching facilities and our complementary notion of an *abstract program action*, which allows grouping of related security functions, also differ from what appears in previous work.

Our monitor language can be viewed as an aspect-oriented programming language (AOPL) in the style of AspectJ [10]. The main high-level difference between our work and previous AOPLs is 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 [19, 20] 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 will suggest a new design strategy for general-purpose AOPLs.

2. Polymer System Overview

Similarly to the designs of Naccio and PoET/Pslang, the Polymer system is composed of two main tools. The first is a policy compiler that compiles program monitors defined in the Polymer language into plain Java and then into Java bytecode. The second tool is a bytecode rewriter that processes ordinary Java 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*, which lists all program methods that might have an impact on system security.
2. Instrument the system libraries specified in the action declaration file. This step may be performed independently of the specification of the security policy. The libraries must be instrumented before the Java Virtual Machine (JVM) starts up since the default JVM security constraints prevent many libraries from being modified or reloaded once the JVM is running.
3. Write and compile the security policy. The policy compiler translates the Polymer policy into ordinary Java and then invokes a Java compiler to translate it to bytecode. Polymer's policy language is described in Section 3; its formal semantics appear in Section 5.

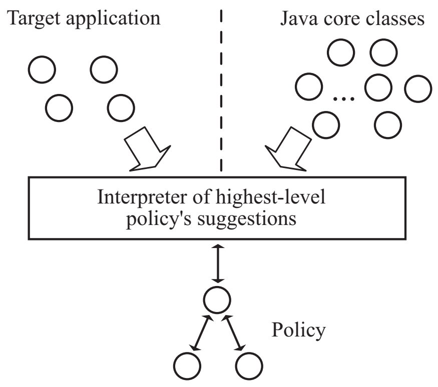


Figure 1. A secure Polymer application

4. Start the JVM with the modified libraries.
5. Load the target application. During this loading, our specialized class loader rewrites the target code in the same way we rewrote the library code in step 2.
6. Execute the secured application.

Figure 1 shows the end result of the process. The 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.

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 more complete policies by composing simpler ones.

3.1 Core Concepts

Polymer is based on three central 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*. An `Action` object matches an action pattern when the action's signature matches the one specified in the pattern. Patterns can 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 two places. First, the action declaration file is a set of action patterns. During the instrumentation process, every action that matches an action pattern in the action declaration file is instrumented. Second, policies 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, as the following example shows.

```
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 value of the method's single parameter.

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).

The monitor's decision about a particular trigger action is conveyed using a `Sug` object. Polymer supplies a subclass of `Sug` for each type of suggestion mentioned in Section 1:

- An `IrrSug` suggests that the trigger action execute unconditionally because the policy does not reason about it.
- An `OKSug` suggests that the trigger action execute even though the action is of interest to the policy.
- An `InsSug` suggests that making a final decision about the target action be deferred until after some auxiliary code is executed and its effects are evaluated.
- A `ReplSug` suggests replacing the trigger action, which computes some return value, with a return value supplied by the policy. The policy may use `InsSug`s to compute the suggested return value.
- An `ExnSug` suggests that the trigger action not be allowed to execute, but also that the target be allowed to continue running. Whenever following an `ExnSug`, Polymer notifies the target that its attempt at invoking the trigger action has been denied by throwing a `SecurityException` that the target can catch before continuing execution.
- A `HaltSug` suggests that the trigger action not be allowed to execute and that the target be halted.

Breaking down the possible interventions of monitors into these categories provides great flexibility. In addition, this breakdown, which was refined by experience with writing security policies in Polymer, simplifies our job tremendously when it comes to controlling monitor effects and building combinators that compose monitors in sensible ways (see Section 3.3).

Policies Programmers encode a run-time monitor in Polymer by extending the base `Policy` class (Figure 2). A new policy must provide an implementation of the `query` method and may optionally override the `accept` and `result` methods.

- `query` analyzes a trigger action and returns a suggestion indicating how to deal with it.
- `accept` is called to indicate to a policy that its suggestion is about to be followed. This gives the policy a chance to perform any bookkeeping needed before the the suggestion is carried out.
- `result` gives the policy access to the return value produced by following its `InsSug` or `OKSug`. The three arguments to `result` are the original suggestion the policy returned, the return value of the trigger action or inserted action (`null` if the return type was `void` and an `Exception` value if the action completed abnormally), and a flag indicating whether the action completed abnormally.

The `accept` method is called before following any suggestion except an `IrrSug`; the `result` method is only called after following an `OKSug` or `InsSug`. After `result` is called with the result of an `InsSug`, the policy is queried again with the original trig-

```
public abstract class Policy {
  public abstract Sug query(Action a);
  public void accept(Sug s) { };
  public void result(Sug s, Object result,
                     boolean wasExnThn) { };
}
```

Figure 2. The parent class of all policies

```
public class Trivial extends Policy {
  public Sug query(Action a)
    { return new IrrSug(this); }
}
```

Figure 3. Policy that allows all actions

ger action (in response to which the policy had just suggested an `InsSug`). Thus, `InsSug`s allow a policy to delay making a decision about a trigger action until after executing another action.

A policy interface consisting of `query`, `accept`, and `result` methods 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.

3.2 Simple Policies

To give a feel for how to write Polymer policies, we define several simple examples in this section; in Sections 3.3 and 4.2 we will build more powerful policies by composing 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 shown in Figure 3. Because the `query` method of `Trivial` always returns an `IrrSug`, it allows all trigger actions to execute unconditionally. To enable convenient processing of suggestions, every `Sug` constructor has at least one argument, the `Policy` making the `Sug`.

For our second example, we consider a more useful policy that disallows executing external code, such as OS system calls, via `java.lang.Runtime.exec(..)` methods. This policy, shown in Figure 4, simply halts the target when it calls `java.lang.Runtime.exec`. The `accept` method notifies the user of the security violation. Notice that this notification does not appear in the `query` method because it is an effectful computation; the notification should not occur if the policy's suggestion is not followed.

In practice, there can be many methods that correspond to a single action that a policy considers security relevant. For example, a policy that logs incoming email may need to observe all actions that can open a message. It can be 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 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, opening email, 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. Abstract actions also make it possible to write platform-independent policies. For example, the set of actions that fetch email may not be the same

```
public class DisSysCalls extends Policy {
   public Sug query(Action a) {
     aswitch(a) {
       case <* java.lang.Runtime.exec(..)>:
         return new HaltSug(this, a);
     }
     return new IrrSug(this);
   }

   public void accept(Sug s) {
     System.out.println("Illegal method called: " +
       s.getTrigger());
   }
}
```

Figure 4. Policy that disallows `Runtime.exec` methods

```
public class GetMail extends AbsAction {
  public boolean matches(Action a) {
    aswitch(a) {
      case <Message IMAPFolder.getMessage(int)> :
      case <void IMAPFolder.fetch(Message[], *)> :
      ...
        return true;
    }
    return false;
  }
  public static Object convertResult(Action a,
                                     Object res) {
    aswitch(a) {
      case <Message IMAPFolder.getMessage(int)> :
        return new Message[] {(Message)res};
      case <void IMAPFolder.fetch(Message[] ma, *)> :
        return ma;
      ...
      default:
        return res;
    }
  }
}
```

Figure 5. Abbreviated abstract action for receiving email messages; the abstract action's signature is `Message[] GetMail()`

on every system, but as long as the implementation of the abstract `GetMail` action is adjusted accordingly, the same policy for regulating email access can be used everywhere.

Figure 5 shows an abstract action for fetching email messages. The `matches` method of an abstract action returns `true` when a provided concrete action is one of the abstract action's constituents. The method has access to the concrete action's run-time parameters and can use this information in making its decision. All constituent concrete actions may not have the same parameter and return types, so one of the abstract action's tasks is to export a consistent interface to policies. This is accomplished via `convertParameter` and `convertResult` methods. The `convertResult` method in Figure 5 allows the `GetMail` abstract action to export a return type of `Message[]`, even though one of its constituents has a `void` return type.

Naccio [8] implements an alternative notion, called platform interfaces, that supports a similar sort of separation between concrete and abstract actions. It appears that our design is slightly more general, as our abstract actions allow programmers to define many-many relationships, rather than many-one relationships, between

```
public class IncomingMail extends Policy {
  ...
  public Sug query(Action a) {
    aswitch(a) {
      case <abs * examples.mail.GetMail()>:
        return new OKSug(this, a);
      case <* MimeMessage.getSubject()>:
      case <* IMAPMessage.getSubject()>:
        String subj = spamifySubject(a.getCaller());
        return new ReplSug(this, a, subj);
      case <done>:
        if(!isClosed(logFile))
          return new InsSug(this, a, new Action(
            logFile, "java.io.PrintStream.close()"));
    }
    return new IrrSug(this, a);
  }
  public void result(Sug sugg, Object res,
                     boolean wasExnThn) {
    if(!sugg.isOK() || wasExnThn) return;
    log(GetMail.convertResult(sugg.getTrigger(), result));
  }
}
```

Figure 6. Abbreviated policy that logs all incoming email and prepends the string "SPAM:" to subject lines on messages flagged by a spam filter

concrete and abstract actions. In addition, our abstract actions are first-class objects that may be passed to and from procedures, and we support the convenience of general-purpose pattern matching.

The example policy in Figure 6 logs all incoming email and prepends the string "SPAM:" to subject lines of messages flagged by a spam filter. To log incoming mail, the policy first tests whether the trigger action matches the `GetMail` abstract action (from Figure 5), using the keyword `abs` in an action pattern to indicate that `GetMail` is abstract. Since `query` methods should not have effects, the policy returns an `OKSug` for each `GetMail` action; the policy logs the fetched messages in the `result` method. Polymer triggers a `done` action when the application terminates; the policy takes advantage of this feature to insert an action that closes the message log. If the `InsSug` recommending that the log be closed is accepted, the policy will be queried again with a `done` action after the inserted action has been executed. In the second query, the log file will already be closed, so the policy will return an `IrrSug`. The policy also intercepts calls to `getSubject` in order to mark email as spam. Instead of allowing the original call to execute, the policy fetches the original subject, prepends "SPAM:" if necessary, and returns the result via a `ReplSug`.

Sometimes, a policy requires notifying the target that executing its trigger action would be a security violation. When no suitable return value can indicate this condition to the target, the policy may make an `ExnSug` rather than a `ReplSug`. For example, an email `Attachments` policy that prevents executable files from being created may, rather than by halting the target outright, signal policy violations by making `ExnSug`s. These will cause `SecurityExceptions` to be raised, which can be caught by the application and dealt with in an application-specific manner.

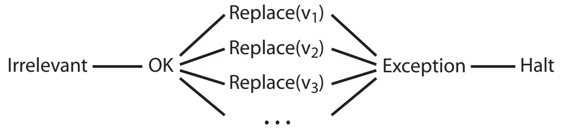
3.3 Policy Combinators

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 be returned by 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. We use each of the following kinds of combinators in the email policy described in Section 4.2.

Conjunctive combinator It is often useful to restrict an application's behavior by applying several policies at once and, for any particular trigger action, 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 suggests insertions, so does the combinator, with any insertions by the left (first) conjunct occurring prior to insertions by the right 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 conjunctive combinator computes and returns the least upper bound of the two suggestions, as described by the following lattice, which orders suggestions in terms of increasing semantic impact.



For instance, `IrrSug` has less impact than `OKSug` since an `IrrSug` indicates the current method is allowed but irrelevant to the policy whereas `OKSug` says it is allowed, but relevant and updates of security state may be needed. `ReplSug`s have more impact than `OKSug`s since they change the semantics of the application. `ReplSug`s containing different replacements are considered inequivalent; consequently, the "conjunction" of two `ReplSug`s is considered to be an `ExnSug`.

Note that a sequence of insertions made by one conjunct may affect the second conjunct. In fact, this is quite likely if the second conjunct considers the inserted actions security-relevant. In this case, 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 second conjunct might suggest that the application be halted.

An abbreviated version of the conjunctive combinator is shown in Figure 7. 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 `accept` and `result` methods to notify the appropriate subpolicies that their suggestions have been accepted and followed.

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

```
public class Conjunction extends Policy {
  private Policy p1, p2;
  public Conjunction(Policy p1, Policy p2) {
    this.p1 = p1; this.p2 = p2;
  }
  public Sug query(Action a) {
    Sug s1=p1.query(a), s2=p2.query(a);
    if(s1.isInsertion()) return SugUtils.getNewSug(
      s1, this, a, new Sug[]{s1});
    if(s2.isInsertion()) return SugUtils.getNewSug(
      s2, this, a, new Sug[]{s2});
    if(s1.isHalt() && s2.isHalt())
      return SugUtils.getNewSug(s1, this, a,
        new Sug[]{s1,s2});
    if(s1.isHalt()) return SugUtils.getNewSug(
      s1, this, a, new Sug[]{s1});
    ...
  }
  public void accept(Sug sug) {
    //notify subpolicies whose suggestions were accepted
    Sug[] sa = sug.getSuggestions();
    for(int i = 0; i < sa.length; i++) {
      sa[i].getSuggestingPolicy().accept(sa[i]);
    }
  }
  ...
}
```

Figure 7. Conjunctive policy combinator

queries its first subpolicy, and if that subpolicy returns an `IrrSug`, `OKSug`, or `InsSug`, it makes the same suggestion. Otherwise, the combinator defers judgment to the second subpolicy. The email policy described in Section 4.2 uses the `TryWith` combinator to join a policy that allows only HTTP connections with a policy that allows only POP and IMAP connections; the resulting policy allows exactly those kinds of connections and no others.

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. Note that if two subpolicies never consider the same action security-relevant, composing them with a `Dominates` combinator is equivalent to composing them with a `Conjunction` combinator, except the `Dominates` combinator is in general more efficient because it need not always query both subpolicies. In our email policy we use `Dominates` to construct a policy that both restricts the kinds of network connections that may be established and prevents executable files from being created. Since these two subpolicies regulate disjoint set of actions, composing them with the `Conjunction` combinator would have needlessly caused the second subpolicy to be queried even when the trigger action was regulated by the first subpolicy, and therefore clearly not of interest to the second.

Selectors Selectors are combinators that choose to enforce exactly one of their subpolicies. The `IsClientSigned` selector of Section 4.2, for example, enforces a weaker policy on the target application if the target is cryptographically signed; otherwise, the selector enforces a stronger policy.

Policy modifiers Policy modifiers are higher-order policies that enforce a single policy while also performing some other actions. Suppose, for example, that we want to log the actions of a target application and the suggestions made by a policy acting on that target. Rather than modifying the existing policy, we can accomplish this by wrapping the policy in an `Audit` unary superpolicy. When queried, `Audit` blindly suggests whatever the original policy's `query` method suggests. `Audit`'s `accept` and `result` methods perform logging operations before invoking the `accept` and `result` methods of the original policy.

Another example of a policy modifier is our `AutoUpdate` superpolicy. This policy checks a remote site once per day to determine if a new policy patch is available. If so, it makes a secure connection to the remote site, downloads the updated policy, and dynamically loads the policy into the JVM as its new subpolicy. Policies of this sort, which determine how to update other policies at run time, are useful because they allow new security constraints to be placed on target applications dynamically, as vulnerabilities are discovered. Note however that because library classes (such as `java.lang.Object`) cannot in general be reloaded while the JVM is running, policies loaded dynamically should consider security-relevant only actions appearing in the static action declaration file. For this reason, we encourage security programmers to be reasonably conservative when writing action declaration files for dynamically updateable policies.

A third useful sort of policy modifier is a `Filter` that blocks a policy from seeing certain actions. 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 `Filter` to prevent such loops. More generally, `Filter`s allow the superpolicy to determine whether an action is relevant to the subpolicy.

4. Empirical Evaluation

Experience implementing and using Polymer has been instrumental in confirming and refining our design.

4.1 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 immediately before and after executing a security-relevant method. The kind of pre- and post-invocation control-flow modifications to bytecode that we use to implement Polymer can be done by tools like AspectJ [10]. 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.

Custom class loaders have often been used to modify bytecode before executing it [2, 4]; we use this technique also. Since libraries used internally by the JVM cannot be rewritten by a custom class loader, we rewrite those libraries before starting the JVM and the target application.

Further discussion of the implementation, including design decisions and performance, can be found in a prior technical report [5].

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; and during execution. 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

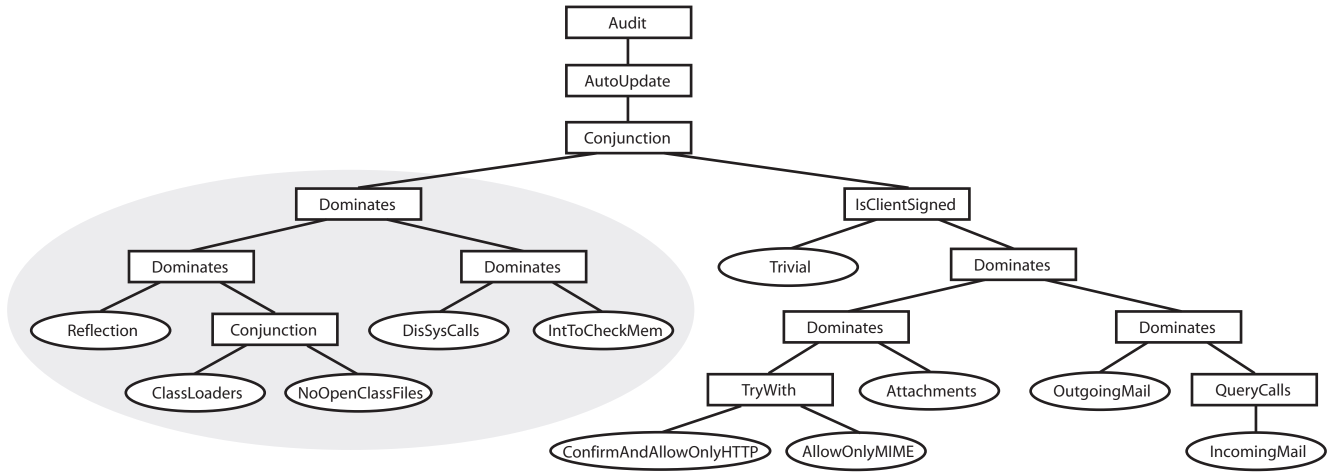


Figure 8. Email policy hierarchy

`java` and `javax` packages of Sun's Java API v.1.4.0) was 107 s, or 3.7 ms per instrumented method.[1] 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 12 ms, twice as long as the VM's default class loader required. The cost of transferring control to and from a Polymer policy while executing a target is very low (approximately 0.62 ms); the run-time overhead is dominated by the computations actually performed by the policy. Hence the cost of monitoring a program with Polymer is almost entirely dependent on the complexity of the security policy.

4.2 Case Study: Securing Email Clients

To test the usefulness of Polymer in practice, we have written a large-scale policy to secure untrusted email clients that use the JavaMail API. The entire policy, presented in Figure 8, is approximately 1800 lines of Polymer code. We have extensively tested the protections enforced by the policy on an email client called Pooka [16], without having to inspect or modify any of the approximately 50K lines of Pooka source code. The run-time cost of enforcing the complex constraints specified by our policy is difficult to measure because the performance of the email client depends largely on interactions with the user; however, our experience indicates that the overhead is rarely noticeable.

The component policies in Figure 8 each enforce a modular set of constraints. The `Trivial` and `Attachments` policies were described in Section 3.2; the `Conjunction`, `TryWith`, `Dominates`, `Audit`, and `AutoUpdate` superpolicies were described in Section 3.3. The left branch of the policy hierarchy (shaded in Figure 8) describes a generic policy that we include in all of our high-level Polymer policies. This branch of policies ensures that a target cannot use class loading, reflection, or system calls maliciously and alerts the user when the memory available to the virtual machine is nearly exhausted. The nonshaded branch of the policy hierarchy describes policies specifically designed for securing an email client and enforces constraints as follows.

[1] 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.

- `IsClientSigned` tests whether the email client is cryptographically signed. If it is, we run `Trivial` but continue to log security-relevant actions and allow dynamic policy updates. If the client is not signed, we run a more restrictive policy.
- `ConfirmAndAllowOnlyHTTP` pops up a window seeking confirmation before allowing HTTP connections, and disallows all other types of network connections.
- `AllowOnlyMIME` allows only standard email socket connections (POP and IMAP).
- `QueryCalls` is a policy modifier that allows security-sensitive actions invoked in the `query` method of its subpolicy to execute unconditionally. `QueryCalls` OKs these actions without requerying the subpolicy in order to prevent infinite loops that can occur when the subpolicy invokes actions that it also monitors. The implementation of `QueryCalls` inspects the dynamic call stack to determine whether a trigger action was invoked in the subpolicy's `query` method.
- `OutgoingMail` logs all mail being sent, pops up a window confirming the recipients of messages (to prevent a malicious client from quietly sending mail on the user's behalf), backs up every outgoing message by sending a BCC to polydemo@cs.princeton.edu, and automatically appends contact information to textual messages.
- `IncomingMail` was shown in an abbreviated form in Figure 6. In addition to logging incoming mail and prepending "SPAM:" to the subject lines of email that fails a spam filter, this policy truncates long subject lines and displays a warning when a message containing an attachment is opened.

5. Formal Semantics

In this section, we give a semantics to the core features of our language. The main purpose of the semantics is to communicate the central workings of our language in a precise and unambiguous manner. We have chosen to give the semantics in the context of a lambda calculus because lambda calculi are inherently simpler to specify than class-based languages such as Java.[2] More importantly, the central elements of our policy language do not depend upon Java-specific features such as classes, methods and inheritance. We could just as easily have implemented policies for a functional language such as ML or a type-safe imperative language.

[2] Even the lightest-weight specification of Java such as Featherweight Java is substantially more complex than the simply-typed lambda calculus.

types :
τ ::= Bool | $(\vec{\tau})$ | τ Ref | $\tau_1 \to \tau_2$ | Poly | Sug | Act | Res
programs :
P ::= $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$
monitored functions :
F ::= fun$f(x{:}\tau_1){:}\tau_2\{e\}$
memories :
M ::= $\cdot$ | $M, l : v$
values :
v ::= true | false | $(\vec{v})$ | l | $\lambda x{:}\tau.e$ | $\text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$ |
irrs | oks | inss(v) | repls(v) | exns | halts | act(f, v) |
result$(v{:}\tau)$
expressions :
e ::= v | x | $(\vec{e})$ | $e_1; e_2$ | ref e | $!e$ | $e_1{:=}e_2$ | $e_1\ e_2$ |
$\text{pol}(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}})$ | inss(e) | repls(e) | act(f, e) |
invk e | result$(e{:}\tau)$ | case e_1 of $(p \Rightarrow e_2 \mid \_ \Rightarrow e_3)$ |
try e_1 with e_2 | raise exn | abort
patterns :
p ::= x | true | false | $(\vec{p})$ | pol(x_1, x_2, x_3) | irrs | oks |
inss(p) | repls(p) | exns | halts | act(f, p) | result$(p{:}\tau)$

Figure 9. Formal syntax

Type safety protects the program monitor's state and code from the untrusted application.

Figure 9 describes the main syntactic elements of the calculus. The language is simply-typed with types for booleans, n-ary tuples, references, and functions. Our additions include simple base types for policies (Poly), suggestions (Sug), actions (Act), which are suspended function applications, and results of those suspended function applications (Res).

Programs as a whole are 4-tuples consisting of a collection of functions that may be monitored, a memory that maps memory locations to values, and two expressions. The first expression represents the security policy; the second expression represents the untrusted application. Execution of a program begins by reducing the policy expression to a policy value. It continues by executing the application expression in the presence of the policy.

Monitored functions (fun$f(x{:}\tau_1){:}\tau_2\{e\}$) are syntactically separated from ordinary functions ($\lambda x{:}\tau.e$).[3] Moreover, we treat monitored function names f as a syntactically separate class of variables from ordinary variables x. Monitored function names may only appear wrapped up as actions as in $\text{act}(f, e)$. These actions are suspended computations that must be explicitly *invoked* with the command $\text{invk}\ e$. Invoking an action causes the function in question to be executed and its result wrapped in a result constructor $\text{result}(e{:}\tau)$. The elimination forms for results and most other objects discussed above is handled through a generic case expression and pattern matching facility. The class of patterns p includes variable patterns x as well as patterns for matching constructors. Ordinary, unmonitored functions are executed via the usual function application command ($e_1\ e_2$).

To create a policy, one applies the policy constructor pol to a query function (e_{query}), which produces suggestions, and security state update functions that execute before (e_{acc}) and after (e_{res}) the monitored method. Each suggestion (irrs, oks, inss, repls, exns, and halts) also has its own constructor. For instance, the repls constructor takes a result object as an argument and the inss suggestion takes an action to execute as an argument. Each suggestion will be given a unique interpretation in the operational semantics.

[3] As usual, we treat expressions that differ only in the names of their bound variables as identical. We often write let $x = e_1$ in e_2 for $(\lambda x{:}\tau.e_2)e_1$.

$\boxed{S; C \vdash e : \tau}$

$$\frac{S; C \vdash e_{\text{query}} : \text{Act} \to \text{Sug} \quad S; C \vdash e_{\text{acc}} : (\text{Act}, \text{Sug}) \to () \quad S; C \vdash e_{\text{res}} : \text{Res} \to ()}{S; C \vdash \text{pol}(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}}) : \text{Poly}}$$

$$\frac{}{S; C \vdash \text{irrs} : \text{Sug}} \qquad \frac{}{S; C \vdash \text{oks} : \text{Sug}}$$

$$\frac{S; C \vdash e : \text{Act}}{S; C \vdash \text{inss}(e) : \text{Sug}} \qquad \frac{S; C \vdash e : \text{Res}}{S; C \vdash \text{repls}(e) : \text{Sug}}$$

$$\frac{}{S; C \vdash \text{exns} : \text{Sug}} \qquad \frac{}{S; C \vdash \text{halts} : \text{Sug}}$$

$$\frac{C(f) = \tau_1 \to \tau_2 \quad S; C \vdash e : \tau_1}{S; C \vdash \text{act}(f, e) : \text{Act}} \qquad \frac{S; C \vdash e : \text{Act}}{S; C \vdash \text{invk}\ e : \text{Res}}$$

$$\frac{S; C \vdash e : \tau}{S; C \vdash \text{result}(e{:}\tau) : \text{Res}}$$

$$\frac{S; C \vdash e_1 : \tau' \quad C \vdash p : (\tau'; C') \quad S; C, C' \vdash e_2 : \tau \quad S; C \vdash e_3 : \tau}{S; C \vdash \text{case}\ e_1\ \text{of}\ (p \Rightarrow e_2 \mid \_ \Rightarrow e_3) : \tau}$$

$\boxed{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$

$$\frac{\vdash \vec{F} : C \quad C \vdash M : S \quad S; C \vdash e_{\text{pol}} : \text{Poly} \quad S; C \vdash e_{\text{app}} : \tau}{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$$

Figure 10. Static semantics (selected rules)

Static Semantics Figure 10 presents selected rules from the static semantics for the language. The main judgment, which types expressions, has the form $S; C \vdash e : \tau$ where S maps reference locations to their types and C maps variables to types. Whenever we add a new binding $x{:}\tau$ to the context, we implicitly alpha-vary x to ensure it does not clash with other variables in the context. A secondary judgment $C \vdash p : (\tau; C')$ is used to check that a pattern p will match objects with type τ and binds variables with types given by C'.

We have worked hard to make the static semantics a simple but faithful model of the implementation. In particular, notice that all actions share the same type (Act) regardless of the type of object they return when invoked. Dynamically, the result of invoking an action is a value wrapped up as a result with type Res. Case analysis is used to safely extract the proper value. This choice allows policy objects to process and react to arbitrary actions. To determine the precise nature of any action and give it a more refined type, the policy will use pattern matching. We have a similar design for action results and replacement values.

The judgement for overall program states has the form $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ where τ is the type of the application code e_{app}. This judgment relies on two additional judgments (definitions not shown) which give types to a library of monitored functions $\vec{F}$ and types to locations in memory M.

$$\boxed{(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})}$$

$$\frac{(\vec{F}, M, \text{Triv}, e) \rightarrow_\beta (M', e')}{(\vec{F}, M, E[e], e_{\text{app}}) \mapsto (\vec{F}, M', E[e'], e_{\text{app}})}$$

where $\text{Triv} = \text{pol}(\lambda x{:}\text{Act.irrs}, \lambda x{:}(\text{Act}, \text{Sug}).(), \lambda x{:}\text{Res}.())$

$$\frac{(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_\beta (M', e')}{(\vec{F}, M, v_{\text{pol}}, E[e]) \mapsto (\vec{F}, M', v_{\text{pol}}, E[e'])}$$

$$\boxed{(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_\beta (M', e'_{\text{app}})}$$

$$\frac{}{(\vec{F}, M, v_{\text{pol}}, (\lambda x{:}\tau.e)v) \rightarrow_\beta (M, e[v/x])}$$

$$\frac{F_i \in \vec{F} \quad F_i = \text{fun} f(x{:}\tau_1){:}\tau_2\{e\}}{(\vec{F}, M, v_{\text{pol}}, \text{invk act}(f, v)) \rightarrow_\beta (M, \text{Wrap}(v_{\text{pol}}, F_i, v))}$$

```
where $\text{Wrap}(\text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}), \text{fun} f(x{:}\tau_1){:}\tau_2\{e\}, v) =$
  let $s = v_{\text{query}}(\text{act}(f, v))$ in
  case $s$ of
    irrs       $\Rightarrow$ let $x = v$ in result$(e{:}\tau_2)$
   | oks       $\Rightarrow v_{\text{acc}}(\text{act}(f, v), s);$
                   let $x = v$ in let $r = \text{result}(e{:}\tau_2)$ in $v_{\text{res}}\ r;\ r$
   | repls$(r)$ $\Rightarrow v_{\text{acc}}(\text{act}(f, v), s);\ r$
   | exns      $\Rightarrow v_{\text{acc}}(\text{act}(f, v), s);$ raise exn
   | inss$(a)$  $\Rightarrow v_{\text{acc}}(\text{act}(f, v), s);\ v_{\text{res}}(\text{invk}\ a);$ invk act$(f, v)$
   | _         $\Rightarrow$ abort
```

Figure 11. Dynamic semantics (selected rules)

Dynamic Semantics To explain execution of monitored programs, we use a context-based semantics. The first step is to define a set of evaluation contexts E, which mark where a beta-reduction can occur. Our contexts specify a left-to-right, call-by-value evaluation order. (We omit the definition to conserve space.)

We specify execution through a pair of judgments, one for top-level evaluation and one for basic reductions as shown in Figure 11. The top-level judgment reveals that the policy expression is first reduced to a value, and then execution of the untrusted application code begins. Execution of many of the constructs is relatively straightforward. One exception is execution of function application. For ordinary functions, we use the usual capture-avoiding substitution. Monitored functions, on the other hand, may only be executed if they are wrapped up as actions and then invoked using the invk command. The invk command applies the query method to discover the suggestion the current policy makes and then interprets the suggestion. Notice, for instance, that to respond to the irrelevant suggestion (irrs), the application simply proceeds to execute the body of the security-relevant action. To respond to the OK suggestion (oks), the application first calls the policy's accept method, then executes the security-relevant action before calling the policy's result method, and finally returns the result of executing the security-relevant action.

Language Properties To check that our language is well-defined, we have proven a standard type-safety result in terms of Preservation and Progress lemmas. Due to space considerations, we have omitted the proofs.

Theorem 1
If $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$
and $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$
then $\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau$.

Theorem 2
If $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ *then either* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$ *is finished (i.e.,* e_{app} *is a value, or* e_{pol} *or* e_{app} *is* $E[\text{abort}]$*, or* e_{pol} *or* e_{app} *is* $E[\text{raise exn}]$ *where* $E \neq E'[\text{try } E'' \text{ with } e]$*), or there exists a configuration* $(\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ *such that* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$.

Observations The semantics gives insight into some of the subtler elements of our implementation, which are important both to system users and to us as implementers.

For example, one might want to consider what happens if a program monitor raises but does not catch an exception (such as a null pointer exception). Tracing through the operational semantics, one can see that the exception will percolate from the monitor into the application itself. If this behavior is undesired, a security programmer can create a top-level superpolicy that catches all exceptions raised by the other policies and deals with them as the programmer sees fit.

As another example, analysis of the operational semantics shows a corner case in which we are unable to fully obey the principle of complete mediation. During the first stage of execution, while the policy itself is evaluated, monitored functions are only protected by a trivial policy that accepts all actions because the actual policy we want to enforce is the one being initialized. Policy writers need to be aware of this unavoidable behavior in order to implement policies correctly.

6. Summary

We have developed a programming methodology for writing general-purpose security policies. The design is radically different from existing policy-specification languages in its division of policies into effectless methods that make suggestions regarding how to handle trigger actions and effectful methods that are called when the policy's suggestions are followed. This design allows general security policies to be composed in meaningful and productive ways. We have implemented our design and shown a sound formal semantics for it. We also demonstrated the practicality of the language by building a sophisticated security policy for email clients from simple, modular, and reuseable policies.

Acknowledgments

We are grateful to Greg Morrisett and the anonymous reviewers for supplying valuable feedback on earlier versions of this paper. This research was supported in part by ARDA grant no. NBCHC030106, by National Science Foundation grants no. CCR-0238328 and CCR-0306313, by the Army Research Office through grant no. DAAD19-02-1-0389, and by a Sloan Fellowship.

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. A language and system for composing security policies. Technical Report TR-699-04, Princeton University, Jan. 2004.

[6] 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.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, Feb. 2005.

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

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

[16] A. Petersen. Pooka: A Java email client, 2003. `http://www.suberic.net/pooka/`.

[17] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *IEEE 63, 9*, pages 1278–1308, Sept. 1975.

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

[19] 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.

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