

Composing Expressive Run-time Security Policies

LUJO BAUER

Carnegie Mellon University

JAY LIGATTI

University of South Florida

and

DAVID WALKER

Princeton University

Program monitors enforce security policies by interposing themselves into the control flow of untrusted software whenever this software attempts to execute security-relevant actions. At the point of interposition, a monitor has authority to permit or deny (perhaps conditionally) the untrusted software's attempted action. Program monitors are common security enforcement mechanisms and integral parts of operating systems, virtual machines, firewalls, network auditors, and anti-virus and anti-spyware tools.

Unfortunately, the run-time policies we require program monitors to enforce grow more complex both as the monitored software is given new capabilities and as policies are refined in response to attacks and user feedback. We propose dealing with policy complexity by organizing policies in such a way as to make them composeable, so that complex policies can be specified more simply as compositions of smaller subpolicy modules. We present a fully implemented language and system called Polymer that allows security engineers to specify and enforce composeable policies on Java applications. We also formalize the central workings of Polymer by defining an unambiguous semantics for our language.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2007 ACM 0000-0000/2007/0000-0001 \$5.00

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*frameworks; patterns; procedures, functions, and subroutines*; D.2.1 [**Software Engineering**]: Requirements/Specifications—*languages; tools*; D.2.5 [**Software Engineering**]: Testing and Debugging—*monitors*; D.3.1 [**Programming Languages**]: Formal Definitions and Theory—*semantics; syntax*

General Terms: Design, Languages, Security

Additional Key Words and Phrases: Policy composition, policy-specification language, policy enforcement

1. INTRODUCTION

Program monitors enforce security policies by interposing themselves into the control flow of untrusted software whenever this software attempts to execute security-relevant actions. At the point of interposition, a monitor has authority to permit or deny (perhaps conditionally) the untrusted software’s attempted action. Program monitors are common security enforcement mechanisms and integral parts of operating systems, virtual machines, firewalls, network auditors, and anti-virus and anti-spyware tools.

The run-time policies we need to enforce in practice tend to grow ever more complex, which makes them very difficult to reason about and manage. The increased complexity occurs for several reasons. First, as software becomes more sophisticated, so do our notions of what constitutes valid (secure) and invalid (insecure) behavior. Witness, for example, the increased complexity of reasoning about security in a multi-user and networked system versus a single-user, stand-alone machine. Similarly, new application domains often require new security constraints; e.g., electronic-commerce and medical-database applications require sophisticated authentication and privacy constraints.

Second, practical security policies also grow more complex as security engineers tighten policies in response to new attacks. When engineers discover an attack, they often add rules to their security policies (increasing policy complexity) to avoid the

ACM Journal Name, Vol. V, No. N, February 2007.

newly observed attacks. For instance, a security engineer might add policy rules that disallow insecure default configurations or that require displaying a warning and asking for user confirmation before downloading dangerous files.

Third, relaxing overly tight policies also often leads to increased policy complexity. In this case, the complexity increases because the original policy forbade too much and needs more sophisticated reasoning to distinguish between safe and dangerous behaviors. For example, an older version of the Java Development Kit (JDK 1.0) required all applets to be sandboxed. User feedback led to the adoption of a more relaxed policy, that only unsigned applets be sandboxed, in a later version (JDK 1.1) [McGraw and Felten 1999]. This relaxation increased policy complexity by additionally requiring the policy to reason about cryptographic signatures.

Complex policies are not only difficult for security engineers to maintain; they also often require complex and powerful enforcement mechanisms, which are themselves prone to error. To have assurance that a desired policy will be enforced, both the policy-specification language and the mechanism that enforces it should be implemented in a principled manner and amenable to formal reasoning about their correctness.

This article addresses the problem of policy complexity by describing Polymer, a programming language and system in which complex run-time policies can be specified and enforced more simply as compositions of smaller *subpolicy* modules. Our compositional design allows security architects to create, reuse, update, and analyze subpolicies in isolation.

1.1 Polymer Language Overview

In Polymer, security policies are first-class objects structured to be arbitrarily composed with other policies. This design allows users to specify complex policies as compositions of simpler policy modules.

Polymer policies specify run-time constraints on untrusted Java bytecode programs. Programmers implement policies by extending Polymer's `Policy` class, which is given a special interpretation by the underlying run-time system. Intuitively, each

Policy object contains the following three elements.

- (1) An effect-free decision procedure that determines how to react to security-sensitive application *actions*, which are suspended method calls that monitors intercept.
- (2) Security state, which can be used to keep track of the application's activity during execution.
- (3) Methods to update the policy's security state.

We call the decision procedure mentioned in (1) above a *query* method. This method takes as input an action object, which contains the security-sensitive method's name, signature, calling object (if any), and actual arguments (if any). A query method returns one of six *suggestions* indicating that: the action is *irrelevant* to the policy; the action is *OK* but relevant; the action should be reconsidered after some other code is *inserted*; the return value of the action should be *replaced* by a precomputed value (which may have been computed using earlier insertion suggestions); a security *exception* should be thrown instead of executing the action; or, the application should be *halted*. We call these return objects **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 supporting pattern matching for actions and developing mechanisms that allow programmers to summarize a collection of application events as an *abstract action*.

1.2 Polymer System Overview

Similarly to the designs of Naccio [Evans and Twyman 1999] and PoET/Pslang [Erlingsson and Schneider 2000], 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 perform the following six steps.

- (1) Write an *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 using the bytecode rewriter. 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 2; its formal semantics appear in Section 4.
- (4) Start the JVM with the modified libraries.
- (5) Load the target application. During this loading, Polymer's custom class loader rewrites the target code in the same way we rewrote the libraries in step two.
- (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 carried out by

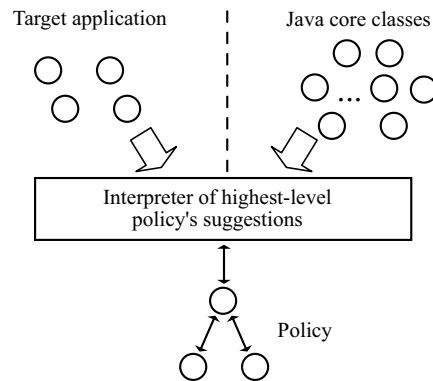


Fig. 1. A secure Polymer application

the policy manager.

1.3 Contributions

The 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 when a policy's suggestion is followed.
- (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.

There are a number of smaller contributions as well. For instance, unlike closely related systems such as Naccio and PoET [Evans and Twyman 1999; Erlingsson and Schneider 2000], Polymer allows 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 2.2, require this capability. In addition, we faithfully implement the principle of *complete mediation* [Saltzer and Schroeder 1975]. 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.

Large portions of this article derive from a paper entitled “Composing Security Policies with Polymer”, which was presented at the ACM SIGPLAN Symposium on Programming Languages Design and Implementation in June, 2005 [Bauer et al. 2005a]. This article extends and completes that conference paper in several ways:

- We give related work a much more complete treatment, in a new Section 5.
- We describe more thoroughly and in greater detail how to write policies using Polymer. We present more examples of Polymer policies (e.g., in Section 2.3). We also provide more in-depth discussions of the example policies and more complete code examples (e.g., for the case study policy in Section 3.2).
- We present the complete semantics and type-safety proof for the Polymer language in Section 4 and the Appendix.

Roadmap. We proceed as follows. Section 2 describes the Polymer language and demonstrates how simple policies can be composed to build complex policies. Section 3 describes our experiences implementing and testing Polymer. Section 4 presents a complete formal semantics for the core features of the Polymer language. Section 5 discusses related work, and Section 6 concludes by summarizing and

describing some directions for future work.

2. THE POLYMER LANGUAGE

This section describes the core features of the Polymer language. We begin with the basic concepts (Section 2.1) and show how to program simple policies (Section 2.2). Then, we demonstrate how to create more complex policies by composing simpler ones (Section 2.3).

2.1 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. The Action API is shown in Figure 2. An Action object is instantiated by Polymer at the call site of every security-sensitive method and passed to the policy.

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 wild-cards: * 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

ACM Journal Name, Vol. V, No. N, February 2007.


```

public class Action {
    // 'caller' is the object on which this action is invoked,
    // 'fullSig' is the action's full signature, and
    // 'params' are the action's actual arguments
    public Action(Object caller, String fullSig, Object[ ] params)
    public boolean equals(Action a)
    public Object[ ] getParams()
    public Class[ ] getParamClasses()
    public Object getCaller()
    public String getMethodName()
    public String getPackageName()
    public String getClassName()
    public String getSignature()
    public String toString()
}

```

Fig. 2. Polymer API for Action objects

action patterns in `aswitch` statements to determine with which security-sensitive action they are dealing. `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 `System.exit`'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 conveys its decision about a particular trigger action using a `Sug` object. Polymer supplies a concrete subclass of the abstract `Sug` class for each type

of suggestion mentioned in Section 1.1:

- 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 `InsSugs` 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 2.3).

The reader might wonder why we distinguish between *irrelevant* and *OK* suggestions; there are two reasons. First, for performance: we need not update policy state when a policy considers the current action irrelevant. Second, some interesting superpolicies, such as the `Dominates` combinator described in Section 2.3, make a semantic distinction between subpolicies' `IrrSug` and `OKSug` suggestions.

Figure 3 shows our `Sug` class. The class contains convenience methods for dynam-

ACM Journal Name, Vol. V, No. N, February 2007.

```

public abstract class Sug {
    public abstract boolean isIrr();
    public abstract boolean isOK();
    public abstract boolean isRepl();
    public abstract boolean isExn();
    public abstract boolean isHalt();
    public abstract boolean isInsertion();
    public abstract Policy getSuggestingPolicy();
    public abstract Action getTrigger();
    public abstract Suggestion[ ] getSuggestions();
    public abstract Object getAuxiliaryObject();
}

```

Fig. 3. Polymer’s abstract `Sug` class

ically determining a suggestion’s concrete type, as well as methods for obtaining the policy that made the suggestion, the action that triggered that policy to make the suggestion, and any other suggestions and auxiliary objects the suggestion’s creator found convenient to store in the `Sug` object. Thus, when given a `Sug` object, Polymer policies can determine the precise circumstances under which that suggestion was made.

Policies. Programmers encode a run-time monitor in Polymer by extending the base `Policy` class (Figure 4). 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 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

```

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

```

Fig. 4. The parent class of all Polymer policies

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 trigger action (in response to which the policy had just suggested an `InsSug`). Thus, `InsSugs` 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.

2.2 Simple Policies

To give a feel for how to write Polymer policies, we define several simple examples in this section; Sections 2.3 and 3.2 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 5. 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`.

```

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

```

Fig. 5. Polymer policy that allows all actions

```

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) {
        if(s.isHalt()) {
            System.err.println("Illegal exec method called");
            System.err.println("About to halt target");
        }
    }
}

```

Fig. 6. Polymer policy that disallows Runtime.exec methods

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 6, 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 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 7 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 7 allows the `GetMail` abstract action to export a return type of `Message[]`.

Naccio [Evans and Twyman 1999] 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 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 8 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 7), using the keyword `abs` in an action pattern to indicate that `GetMail` is abstract. Since query methods should not have effects, the policy

ACM Journal Name, Vol. V, No. N, February 2007.

```

public class GetMail extends AbsAction {
  public boolean matches(Action a) {
    aswitch(a) {
      case <Message IMAPFolder.getMessage(int)> :
      case <void IMAPFolder.fetch(Message[ ], *)> :
      case <Message IMAPFolder.getMessageByUID(long)> :
      case <Message[ ] IMAPFolder.getMessagesByUID(long[ ])> :
      case <Message[ ] IMAPFolder.getMessagesByUID(long, long)> :
      case <Message[ ] IMAPFolder.search(..)> :
      case <Message[ ] IMAPFolder.expunge():
      case <Message[ ] POP3Folder.expunge():
      case <void POP3Folder.fetch(Message[ ], *)>:
      case <Message POP3Folder.getMessage(int)>:
        return true;
    }
    return false;
  }

  public static Object convertResult(Action a, Object res) {
    aswitch(a) {
      case <void IMAPFolder.fetch(Message[ ] ma, *)> :
        return ma;
      case <Message IMAPFolder.getMessage(int)> :
      case <Message IMAPFolder.getMessageByUID(long)> :
      case <Message POP3Folder.getMessage(int)>:
        return new Message[ ] {(Message)res};
      case <void POP3Folder.fetch(Message[ ] ma, *)>:
        return ma;
      default:
        return res;
    }
  }
}

```

Fig. 7. Abstract action for receiving email messages; the action's signature is `Message[] GetMail()`

```

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()", new Object[ ]{}));
        }
        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));
    }
}

```

Fig. 8. Abbreviated Polymer policy that logs all incoming email and prepends “SPAM:” to subject lines of messages flagged by a spam filter

returns an `OKSug` for each `GetMail` action; the policy logs the fetched messages in the `result` method. Polymer triggers a `done` action when the target 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

ACM Journal Name, Vol. V, No. N, February 2007.

the original subject, prepends “SPAM:” if necessary, and returns the result via a `ReplSug`. Running a such a spam filter on an email client allows end users to filter based on individually customized rules.

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 (e.g., when returning `null` would not be interpreted as an error), the policy may make an `ExnSug` rather than a `ReplSug`. For example, the email `Attachments` policy in Figure 9 seeks user confirmation before creating an executable file. If the user fails to provide the confirmation, the `Attachments` policy signals a policy violation by returning an `ExnSug`, rather than by halting the target outright. Unless overruled by an enclosing policy, this `ExnSug` causes a `SecurityException` to be thrown, which the application can catch and handle in an application-specific manner.

2.3 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 and 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 and distribute with Polymer a library of common combinators [Bauer et al. 2005b]; however, policy architects are always free to develop new combinators to suit their own specific needs.

We next describe several types of combinators we have developed and found particularly useful in practice. The email policy described in Section 3.2 includes all of them. Although our combinators were developed through experience and seem to match intuitive notions of policy conjunction, precedence, etc., we have not formalized their semantics beyond the intuition given in this subsection and their Polymer code implementations. Nonetheless, Krishnan provides formal semantics for several of our combinators [Krishnan 2005].

```

import javax.swing.*;

public class Attachments extends Policy {
    private boolean isNameBad(String fn) {
        return(fn.endsWith(".exe") || fn.endsWith(".vbs") ||
            fn.endsWith(".hta") || fn.endsWith(".mdb"));
    }

    private boolean userCancel = false; //user disallowed the file?
    private boolean noAsk = false; //can we skip the confirmation?

    public Sug query(polymer.Action a) {
        aswitch(a) {
            case (abs void examples.mail.FileWrite(String fn)):
                if(noAsk) return new OKSug(this, a);
                if(userCancel) return new ExnSug(this, a);
                polymer.Action insAct = new polymer.Action(null,
                    "javax.swing.JOptionPane.showConfirmDialog( java.awt.Component," +
                    " java.lang.Object, java.lang.String, int)",
                    new Object[ ]{null, "The target is creating file: " + fn +
                    " This is a dangerous file type. Are you sure you want to create this file?",
                    "Security Question", new Integer(JOptionPane.YES_NO_OPTION)});
                if(isNameBad(fn)) return new InsSug(this, a, insAct);
                return new IrrSug(this, a);
            }
        return new IrrSug(this);
    }

    public synchronized void accept(Sug s) {
        if(s.isExn()) userCancel = false;
        if(s.isOK()) noAsk = false;
    }

    public synchronized void result(Sug s, Object result, boolean wasExnThn) {
        if(s.isInsertion() && ((Integer)result).intValue() == JOptionPane.NO_OPTION)
            userCancel = true;
        if(s.isInsertion() && ((Integer)result).intValue() == JOptionPane.YES_OPTION)
            noAsk = true;
    }
}

```

ACM Journal Name, Vol. V, No. N, February 2007.

Fig. 9. Polymer policy that seeks confirmation before creating .exe, .vbs, .hta, and .mdb files

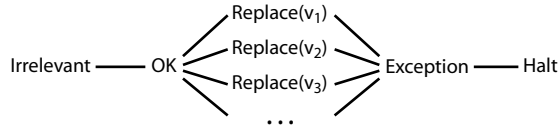


Fig. 10. Lattice ordering of Polymer suggestions' semantic impact

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 lattice in Figure 10, 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. `ReplSugs` have more impact than `OKSugs` since they change the semantics of the application. `ReplSugs` containing different replacements are considered inequivalent; consequently, the “conjunction” of two `ReplSugs` 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 afterward. 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.

Figure 11 contains our conjunctive combinator. The invocations of `SugUtils.cpSug` in the query method create new suggestions with the same type as the first parameter in each call. 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 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 (shown in Figure 12), which 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 3.2 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 the `Dominates` combinator is equivalent to composing them with the `Conjunction` combinator, except the `Dominates` combinator is in general more efficient because it need not always query both subpolicies. In our email policy we

ACM Journal Name, Vol. V, No. N, February 2007.

```

public class Conjunction extends Policy {
  private Policy p1, p2; //subpolicies
  public Conjunction(Policy p1, Policy p2) {this.p1=p1; this.p2=p2;}

  public Sug query(Action a) {
    //return the most restrictive subpolicy suggestion
    Sug s1=p1.query(a), s2=p2.query(a);
    if(SugUtils.areSugsEqual(s1,s2)) return SugUtils.cpSug(s1, this, a, new Sug[ ]{s1,s2});
    if(s1.isInsertion()) return SugUtils.cpSug(s1, this, a, new Sug[ ]{s1});
    if(s2.isInsertion()) return SugUtils.cpSug(s2, this, a, new Sug[ ]{s2});
    if(s1.isHalt()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isHalt()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    if(s1.isExn()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isExn()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    if(s1.isRepl() && s2.isRepl()) return new ExnSug(this, a);
    if(s1.isRepl()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isRepl()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    if(s1.isOK()) return SugUtils.cpSug(s1,this,a,new Sug[ ]{s1});
    if(s2.isOK()) return SugUtils.cpSug(s2,this,a,new Sug[ ]{s2});
    return new IrrSug(this, a);
  }

  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]);
    }
  }

  public void result(Sug sug, Object result, boolean wasExnThn) {
    //notify subpolicies whose suggestions were followed
    Sug[ ] sa = sug.getSuggestions();
    for(int i = 0; i < sa.length; i++) {
      sa[i].getSuggestingPolicy().result(sa[i], result, wasExnThn);
    }
  }
}

```

ACM Journal Name, Vol. V, No. N, February 2007.
 Fig. 11. A conjunctive policy combinator

```

public class TryWith extends Policy {
    //subpolicies
    private Policy p1, p2;

    public TryWith(Policy p1, Policy p2) {this.p1=p1; this.p2=p2;}

    public Sug query(Action a) {
        Sug s1=p1.query(a);

        //if p1 accepts or inserts, return its suggestion
        if(s1.isInsertion() || s1.isOK() || s1.isLrr())
            return SugUtils.cpSug(s1, this, a, new Sug[ ]{s1});

        //otherwise return whatever p2 suggests
        Sug s2=p2.query(a);
        return SugUtils.cpSug(s2, this, a, new Sug[ ]{s2});
    }

    public void accept(Sug sug) {
        //notify the subpolicy that made the now accepted suggestion
        Sug[ ] sa = sug.getSuggestions();
        sa[0].getSuggestingPolicy().accept(sa[0]);
    }

    public void result(Sug sug, Object result, boolean wasExnThn) {
        //notify the subpolicy that made the now followed suggestion
        Sug[ ] sa = sug.getSuggestions();
        sa[0].getSuggestingPolicy().result(sa[0], result, wasExnThn);
    }
}

```

Fig. 12. The TryWith policy combinator

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 3.2, 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 the `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 the actions that appear 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, Filters allow the superpolicy to determine whether an action should be made invisible to the subpolicy.

3. EMPIRICAL EVALUATION

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

3.1 Implementation

The principal requirement for enforcing the run-time policies in which we are interested is that the flow of control of a running program passes to a monitor immediately before and after executing security-relevant methods. 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 [Kiczales et al. 2001]. 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 [Apache Software Foundation 2003] to develop our own bytecode rewriting tool.

Custom class loaders have often been used to modify bytecode before executing it [Agesen et al. 1997; Bauer et al. 2003]; 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.

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 `java` and `javax` packages of Sun's Java API v.1.4.0), inserting monitor invocations at the proper times before and after every method executes, was 107 s, or 3.7 ms per instrumented method.¹ Because we only insert hooks for calling the interpreter of the highest-level policy's suggestions (see Figure 1), rather than inlining or invoking any particular policy, we introduce a level of indirection that permits policies to be updated dynamically without rewriting application or library code. Hence, this instrumentation only needs to be performed once.

The average time to load nonlibrary 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.

3.2 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, outlined in Figure 13, is approximately 1800 lines of Polymer code. We have extensively tested the protections enforced by the policy on an email client called Pooka [Petersen 2003], 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

¹The 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.

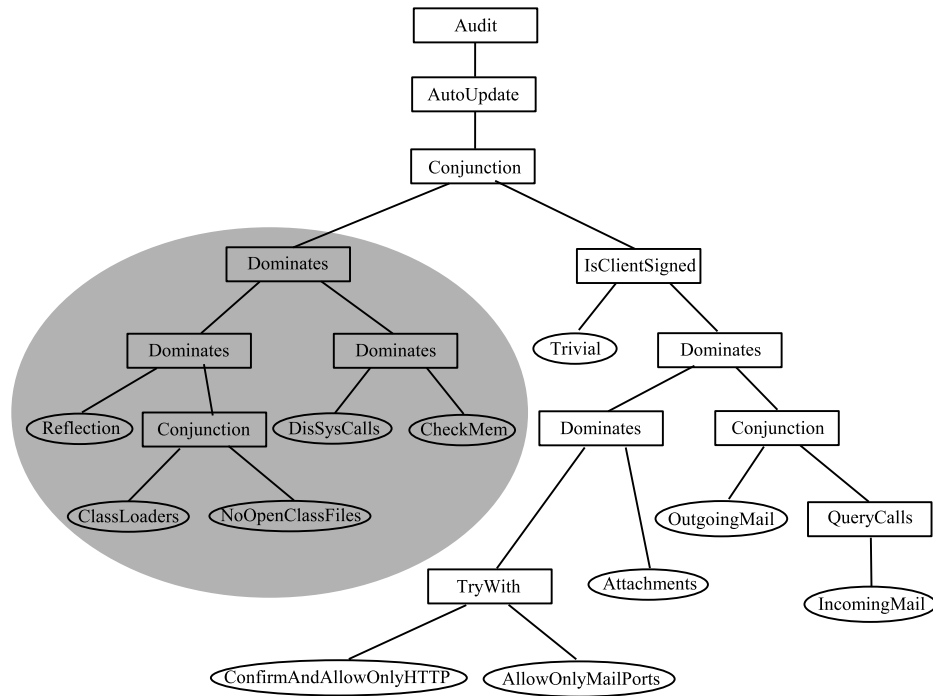


Fig. 13. Email policy hierarchy

experience indicates that the overhead is rarely noticeable.

The component policies in Figure 13 each enforce a modular set of constraints. The `Trivial` and `Attachments` policies were described in Section 2.2; the `Conjunction`, `TryWith`, `Dominates`, `Audit`, and `AutoUpdate` superpolicies were described in Section 2.3. The left branch of the policy hierarchy (shaded in Figure 13) 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 (determined by generating interrupts to poll the `java.lang.Runtime.totalMemory` and `java.lang.Runtime.maxMemory` methods every four seconds). The nonshaded branch of the policy hierarchy describes policies specifically designed for securing an email client and enforces constraints as follows.

—`IsClientSigned` tests whether the email client is cryptographically signed. If it is,

```

public class AllowOnlyMailPorts extends Policy {
  public Sug query(Action a) {
    aswitch(a) {
      case <abs void examples.mail.NetOpen(String addr, int port)>:
        String logStr = "Connection made to "+addr+", port "+port;
        if(port==143           //IMAP connection
          || port==993        //SSL IMAP connection
          || port==25         //SMTP connection
          || port==110        //POP3 connection
          || port==995        //SSL POP3 connection
          return new OKSug(this, a, null, logStr);
        else return new ExnSug(this, a);
      }
      return new IrrSug(this);
    }
    public void accept(Sug s) {
      if(s.isOK()) System.out.println(s.getAuxiliaryObject());
    }
  }
}

```

Fig. 14. Polymer policy that only allows network connections to email ports

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.
- `AllowOnlyMailPorts` only allows socket connections to standard email ports (SMTP, regular POP and IMAP, and SSL-based POP and IMAP). This policy suggests throwing `SecurityExceptions` to prevent the target from making any other types of network connections. Figure 14 contains the code for this policy. Note that with a lot of engineering effort and run-time overhead, we could encode the low-level details of the email protocols into our policy to enforce a stronger `AllowOnlyMail` policy that would precisely ensure that untrusted email clients only make connections that obey standard email protocols. However, we have chosen to enforce

the much simpler and lower-overhead `AllowOnlyMailPorts` policy that provides weaker, port-based guarantees.

- `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 `poly-demo@cs.princeton.edu`, and automatically appends contact information to textual messages.
- `IncomingMail` was shown in an abbreviated form in Figure 8. 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.

4. FORMAL SEMANTICS OF THE POLYMER LANGUAGE

We next give a complete semantics to the core features of the Polymer language in order to precisely and unambiguously communicate the central workings of our language and to highlight their simplicity. We used Java as the basis for our Polymer implementation to make the system widely accessible and to take advantage of the wealth of existing Java libraries and applications. However, we choose to give the Polymer semantics in the context of a lambda calculus because lambda calculi are inherently simpler to specify than class-based languages such as Java (even the lightest-weight specification of Java such as Featherweight Java [Igarashi et al. 1999] is substantially more complex than the simply-typed lambda calculus). More importantly, the central elements of our policy language do not depend

ACM Journal Name, Vol. V, No. N, February 2007.

upon Java-specific features such as classes, methods, and inheritance. We could just as easily have implemented Polymer policies for a functional language such as ML [Milner et al. 1997] or a type-safe imperative language (type safety ensures complete mediation and protects the program monitor’s state and code from being tampered with by the untrusted application).

4.1 Syntax

Figure 15 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 ($\text{fun } f(x:\tau_1):\tau_2\{e\}$) have global scope and are syntactically separated from ordinary functions ($\lambda x:\tau.e$).² Moreover, we treat monitored function names f , which have global scope and may therefore alpha-vary over entire programs, as a syntactically separate class of variables from ordinary variables x . Monitored function names are unique and 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 are handled through a generic case expression and pattern matching facility. The class of pat-

²As usual, we treat expressions that differ only in the names of their bound variables as identical. We often write $\text{let } x = e_1 \text{ in } e_2$ for $(\lambda x:\tau.e_2)e_1$.

Types	$\tau ::= \text{Bool} \mid (\vec{\tau}) \mid \tau \text{ Ref} \mid \tau_1 \rightarrow \tau_2 \mid \text{Poly} \mid \text{Sug} \mid \text{Act} \mid \text{Res}$
Programs	$P ::= (\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$
Monitored functions	$F ::= \text{fun}f(x:\tau_1):\tau_2\{e\}$
Memories	$M ::= \cdot \mid M, l : v$
Values	$v ::= \text{true} \mid \text{false} \mid (\vec{v}) \mid l \mid \lambda x:\tau.e \mid \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}) \mid \text{irrs} \mid \text{oks} \mid \text{inss}(v) \mid \text{repls}(v) \mid \text{exns} \mid \text{halts} \mid \text{act}(f, v) \mid \text{result}(v:\tau)$
Expressions	$e ::= v \mid x \mid (\vec{e}) \mid e_1; e_2 \mid \text{ref } e \mid !e \mid e_1 := e_2 \mid e_1 \ e_2 \mid \text{pol}(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}}) \mid \text{inss}(e) \mid \text{repls}(e) \mid \text{act}(f, e) \mid \text{invk } e \mid \text{result}(e:\tau) \mid \text{case } e_1 \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3) \mid \text{try } e_1 \text{ with } e_2 \mid \text{raise exn} \mid \text{abort}$
Patterns	$p ::= x \mid \text{true} \mid \text{false} \mid (\vec{p}) \mid \text{pol}(x_1, x_2, x_3) \mid \text{irrs} \mid \text{oks} \mid \text{inss}(p) \mid \text{repls}(p) \mid \text{exns} \mid \text{halts} \mid \text{act}(f, p) \mid \text{result}(p:\tau)$

Fig. 15. Abstract syntax for the Polymer calculus

terns 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.

4.2 Static Semantics

Figures 16, 17, and 18 present the rules for the language’s static semantics. 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. More precisely, S and C have the following forms.

Label stores $S ::= \cdot \mid S, l : \tau$

Variable contexts $C ::= \cdot \mid C, x : \tau \mid C, f : \tau$

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.

We have worked hard to make the static semantics a simple but faithful model of the implementation. In particular, notice in Figure 16 that well-typed policies contain a query method, which takes an action and returns a suggestion, and accept and result methods, which perform state updates. In addition, notice in Figure 16 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 judgment for overall program states (shown in the middle of Figure 16) 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 (shown in the bottom of Figure 16), which give types to a library of monitored functions \vec{F} and types to locations in memory M .

Figure 17 presents the static semantics for case expressions and pattern matching. The auxiliary judgment $C \vdash p : (\tau; C')$ is used to check that a pattern p matches objects with type τ and binds variables with types given by C' .

Finally, Figure 18 displays the remaining, straightforward rules for the static

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

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

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

$$\frac{}{S; C \vdash \text{exns} : \text{Sug}} \quad \frac{}{S; C \vdash \text{halts} : \text{Sug}} \quad \frac{C(f) = \tau_1 \rightarrow \tau_2 \quad S; C \vdash e : \tau_1}{S; C \vdash \text{act}(f, e) : \text{Act}}$$

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

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

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

$$\boxed{\vdash \vec{F} : C}$$

$$\frac{\vec{F} = \text{fun } f_1(x_1:\tau_1):\tau'_1\{e_1\}, \dots, \text{fun } f_n(x_n:\tau_n):\tau'_n\{e_n\} \quad C = f_1 : \tau_1 \rightarrow \tau'_1, \dots, f_n : \tau_n \rightarrow \tau'_n \quad \forall i \in \{1..n\} . \dots ; C, x_i : \tau_i \vdash e_i : \tau'_i}{\vdash \vec{F} : C}$$

$$\boxed{C \vdash M : S}$$

$$\frac{\text{dom}(M) = \text{dom}(S) \quad \forall l \in \text{dom}(M) . S; C \vdash M(l) : S(l)}{C \vdash M : S}$$

Fig. 16. Static semantics (rules for policies, suggestions, actions, and programs)

semantics of standard expressions in our language.

4.3 Dynamic Semantics

To explain execution of monitored programs, we use a small-step, context-based semantics. We first define a set of evaluation contexts E , which mark where a beta-

ACM Journal Name, Vol. V, No. N, February 2007.

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

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

$$\boxed{C \vdash p : (\tau'; C')}$$

$$\overline{C \vdash \text{pol}(x_1, x_2, x_3) : (\text{Poly}; x_1 : \text{Act} \rightarrow \text{Sug}, x_2 : (\text{Act}, \text{Sug}) \rightarrow (), x_3 : \text{Res} \rightarrow ())}$$

$$\overline{C \vdash \text{oks} : (\text{Sug}; \cdot)} \quad \overline{C \vdash \text{halts} : (\text{Sug}; \cdot)} \quad \overline{C \vdash \text{irrs} : (\text{Sug}; \cdot)} \quad \overline{C \vdash \text{exns} : (\text{Sug}; \cdot)}$$

$$\frac{C \vdash p : (\text{Res}; C')}{C \vdash \text{repls}(p) : (\text{Sug}; C')} \quad \frac{C \vdash p : (\text{Act}; C')}{C \vdash \text{inss}(p) : (\text{Sug}; C')} \quad \frac{C \vdash p : (\tau; C')}{C \vdash \text{result}(p : \tau) : (\text{Res}; C')}$$

$$\frac{C(f) = \tau_1 \rightarrow \tau_2 \quad C \vdash p : (\tau_1; C')}{C \vdash \text{act}(f, p) : (\text{Act}; C')} \quad \overline{C \vdash \text{true} : (\text{Bool}; \cdot)} \quad \overline{C \vdash \text{false} : (\text{Bool}; \cdot)}$$

$$\frac{\forall i \in \{1..n\} . C \vdash p_i : (\tau_i; C_i)}{C \vdash (p_1, \dots, p_n) : ((\tau_1, \dots, \tau_n); C_1, \dots, C_n)} \quad \overline{C \vdash x : (\tau; x : \tau)}$$

Fig. 17. Static semantics (rules for case expressions)

reduction can occur. Our contexts specify a left-to-right, call-by-value evaluation order, as shown in Figure 19.

We specify execution through a pair of judgments, one for top-level evaluation (shown in the top of Figure 20) and one for basic reductions (shown in the bottom of Figure 20 and in Figures 21 and 22). The top-level judgment has the form $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}', M', e'_{\text{pol}}, e'_{\text{app}})$ and defines how whole programs take small steps dynamically. This top-level judgment reveals that the policy expression in a program is first reduced to a value before execution of the untrusted application code begins.

Execution of many of the constructs is relatively straightforward. One exception is execution of function application, the rules for which are given in the bottom half of Figure 20. For ordinary functions, we use the usual capture-avoiding sub-

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

$$\frac{C(x) = \tau}{S; C \vdash x : \tau} \quad \frac{}{S; C \vdash \text{true} : \text{Bool}} \quad \frac{}{S; C \vdash \text{false} : \text{Bool}}$$

$$\frac{\forall i \in \{1..n\} . S; C \vdash e_i : \tau_i}{S; C \vdash (e_1, \dots, e_n) : (\tau_1, \dots, \tau_n)} \quad \frac{S; C \vdash e_1 : () \quad S; C \vdash e_2 : \tau}{S; C \vdash e_1; e_2 : \tau} \quad \frac{S(l) = \tau}{S; C \vdash l : \tau \text{ Ref}}$$

$$\frac{S; C \vdash e : \tau}{S; C \vdash \text{ref } e : \tau \text{ Ref}} \quad \frac{S; C \vdash e : \tau \text{ Ref}}{S; C \vdash !e : \tau} \quad \frac{S; C \vdash e_1 : \tau \text{ Ref} \quad S; C \vdash e_2 : \tau}{S; C \vdash e_1 := e_2 : ()}$$

$$\frac{S; C, x : \tau \vdash e : \tau'}{S; C \vdash \lambda x : \tau. e : \tau \rightarrow \tau'} \quad \frac{S; C \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad S; C \vdash e_2 : \tau_1}{S; C \vdash e_1 \ e_2 : \tau_2}$$

$$\frac{}{S; C \vdash \text{abort} : \tau} \quad \frac{}{S; C \vdash \text{raise exn} : \tau} \quad \frac{S; C \vdash e_1 : \tau \quad S; C \vdash e_2 : \tau}{S; C \vdash \text{try } e_1 \text{ with } e_2 : \tau}$$

Fig. 18. Static semantics (standard rules)

Evaluation

contexts $E ::= [] \mid (\vec{v}, E, \vec{e}) \mid E; e_2 \mid \text{ref } E \mid !E \mid E := e_2 \mid v := E \mid E \ e \mid v \ E \mid \text{pol}(E, e_2, e_3) \mid$
 $\text{pol}(v_1, E, e_3) \mid \text{pol}(v_1, v_2, E) \mid \text{inss}(E) \mid \text{repls}(E) \mid \text{act}(f, E) \mid \text{invk } E \mid$
 $\text{result}(E : \tau) \mid \text{case } E \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3) \mid \text{try } E \text{ with } e$

Fig. 19. Evaluation contexts

stitution. 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.

The other beta reductions (i.e., besides the ones for function application) are

ACM Journal Name, Vol. V, No. N, February 2007.

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

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

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

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

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

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

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

where $\text{Wrap}(\text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}), \text{funf}(x:\tau_1):\tau_2\{e\}, v) =$

let $s = v_{\text{query}}(\text{act}(f, v))$ in

case s of

- irrs \Rightarrow let $x = v$ in $\text{result}(e:\tau_2)$
- | 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 $\Rightarrow v_{\text{acc}}(\text{act}(f, v), s); \text{raise exn}$
- | inss(a) $\Rightarrow v_{\text{acc}}(\text{act}(f, v), s); v_{\text{res}}(\text{invk } a); \text{invk act}(f, v)$
- | - $\Rightarrow \text{abort}$

Fig. 20. Dynamic semantics (policy and target steps; beta steps for functions)

straightforward and appear in Figures 21 and 22. The rules for case expressions and pattern matching (Figure 21) rely on an auxiliary judgment $v \sim p : V$, which holds when value v matches pattern p , and the matching produces capture-avoiding variable substitutions V .

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

$$\frac{v \sim p : V}{(\vec{F}, M, v_{\text{pol}}, \text{case } v \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3)) \rightarrow_{\beta} (M, e_2[V])}$$

$$\frac{\neg \exists V. v \sim p : V}{(\vec{F}, M, v_{\text{pol}}, \text{case } v \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3)) \rightarrow_{\beta} (M, e_3)}$$

$$\boxed{v \sim p : V}$$

$$\overline{\text{pol}(v_1, v_2, v_3) \sim \text{pol}(x_1, x_2, x_3) : v_1/x_1, v_2/x_2, v_3/x_3} \quad \overline{\text{true} \sim \text{true} : \cdot} \quad \overline{\text{false} \sim \text{false} : \cdot}$$

$$\overline{\text{oks} \sim \text{oks} : \cdot} \quad \overline{\text{halts} \sim \text{halts} : \cdot} \quad \overline{\text{irrs} \sim \text{irrs} : \cdot} \quad \overline{\text{exns} \sim \text{exns} : \cdot}$$

$$\frac{v \sim p : V}{\text{repls}(v) \sim \text{repls}(p) : V} \quad \frac{v \sim p : V}{\text{inss}(v) \sim \text{inss}(p) : V} \quad \frac{v \sim p : V}{\text{act}(f, v) \sim \text{act}(f, p) : V}$$

$$\frac{v \sim p : V}{\text{result}(v : \tau) \sim \text{result}(p : \tau) : V} \quad \overline{v \sim x : v/x} \quad \frac{\forall i \in \{1..n\} . v_i \sim p_i : V_i}{(v_1, \dots, v_n) \sim (p_1, \dots, p_n) : V_1, \dots, V_n}$$

Fig. 21. Dynamic semantics (beta steps for case expressions)

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

$$\frac{l \notin \text{dom}(M)}{(\vec{F}, M, v_{\text{pol}}, \text{ref } v) \rightarrow_{\beta} ((M, l : v), l)} \quad \frac{M(l) = v}{(\vec{F}, M, v_{\text{pol}}, !l) \rightarrow_{\beta} (M, v)}$$

$$\overline{(\vec{F}, M, v_{\text{pol}}, (); e) \rightarrow_{\beta} (M, e)} \quad \overline{(\vec{F}, M, v_{\text{pol}}, l := v) \rightarrow_{\beta} ([l \rightarrow v]M, ())}$$

$$\overline{(\vec{F}, M, v_{\text{pol}}, \text{try } v \text{ with } e) \rightarrow_{\beta} (M, v)} \quad \frac{E \neq E'[\text{try } E'' \text{ with } e']}{(\vec{F}, M, v_{\text{pol}}, \text{try } E[\text{raise exn}] \text{ with } e) \rightarrow_{\beta} (M, e)}$$

Fig. 22. Dynamic semantics (standard beta steps)

4.4 Semantics-based Observations

The formal 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.

4.5 Type Safety

To check that our language is sound, we have proven a standard type-safety result in terms of Preservation and Progress theorems, which together imply that statically well-typed programs do not “get stuck” dynamically. Type safety is a particularly important result in the context of monitor-based policy-specification languages because it guarantees that when whole programs are well typed, their untrusted applications can never tamper with monitors’ state or code. Type safety also guarantees that well-typed programs obey complete mediation—that control always properly flows to the monitoring code before and after security-sensitive application methods execute. These guarantees follow immediately from the type-safety theorems and the language’s static and dynamic semantics.

We state below the lemmas and theorems in our proof of type safety and provide the proof technique for each. Additional proof details appear in the Appendix.

LEMMA 4.1 VARIABLE SUBSTITUTION. *If $S; C, x : \tau' \vdash e : \tau$ and $S; C \vdash e' : \tau'$ then $S; C \vdash e[e'/x] : \tau$.*

PROOF. By induction on the derivation of $S; C, x : \tau' \vdash e : \tau$. \square

LEMMA 4.2 STORE SUBSTITUTION. *If $C \vdash M : S$ and $S(l) = \tau$ and $S; C \vdash v : \tau$ then $C \vdash [l \rightarrow v]M : S$.*

PROOF. Immediate by the sole typing rule for $C \vdash M : S$. \square

LEMMA 4.3 WEAKENING. *If $S; C \vdash e : \tau$ and S' extends S and C' extends C then $S'; C' \vdash e : \tau$.*

PROOF. By induction on the derivation of $S; C \vdash e : \tau$. \square

LEMMA 4.4 INVERSION OF TYPING. *Every typing rule is invertible. That is, if the conclusion of any typing rule (in Figures 16, 17, and 18) holds then its premises must also hold. For example, if $S; C \vdash \text{inss}(e) : \text{Sug}$ then $S; C \vdash e : \text{Act}$; as another example, if $C \vdash \text{act}(f, p) : (\text{Act}; C')$ then $C(f) = \tau_1 \rightarrow \tau_2$ and $C \vdash p : (\tau_1; C')$.*

PROOF. Immediate by inspection of the typing rules. \square

LEMMA 4.5 CANONICAL FORMS. *If $S; C \vdash v : \tau$ then*

- $\tau = \text{Bool}$ implies $v = \text{true}$ or $v = \text{false}$
- $\tau = (\tau_1, \dots, \tau_n)$ implies $v = (v_1, \dots, v_n)$
- $\tau = \tau' \text{ Ref}$ implies $v = l$
- $\tau = \tau_1 \rightarrow \tau_2$ implies $v = \lambda x : \tau_1. e$
- $\tau = \text{Poly}$ implies $v = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$
- $\tau = \text{Sug}$ implies $v = \text{irrs}$ or $v = \text{oks}$ or $v = \text{inss}(\text{act}(f, v))$ or $v = \text{exns}$ or $v = \text{halts}$ or $v = \text{repls}(\text{result}(v : \tau))$
- $\tau = \text{Act}$ implies $v = \text{act}(f, v)$
- $\tau = \text{Res}$ implies $v = \text{result}(v : \tau)$.

PROOF. By induction on the derivation of $S; C \vdash v : \tau$, using the definition of values (given in Figure 15). \square

Definition 4.6 Well-typed Context. A context E is well typed, written $S; C \vdash E_\tau : \tau'$, if and only if $S; C, x : \tau \vdash E[x] : \tau'$ (where x is not a free variable in E).

LEMMA 4.7 WELL-TYPED, FILLED CONTEXT. *If $S; C \vdash E_\tau : \tau'$ and $S; C \vdash e : \tau$ then $S; C \vdash E[e] : \tau'$.*

PROOF. Immediate by Definition 4.6 (well-typed context) and Lemma 4.1 (variable substitution). \square

LEMMA 4.8 CONTEXT DECOMPOSITION. *If $S; C \vdash E[e] : \tau$ then there exists a τ' such that $S; C \vdash E_{\tau'} : \tau$ and $S; C \vdash e : \tau'$.*

PROOF. By induction on the structure of E . \square

Definition 4.9 Well-typed Substitutions. A sequence of variable substitutions V has type C' , written $S; C \vdash V : C'$, if and only if for all $x \in \text{dom}(C')$ there exists v such that $v/x \in V$ and $S; C \vdash v : C'(x)$.

LEMMA 4.10 PATTERN TYPES. *If $S; C \vdash v : \tau'$ and $v \sim p : V$ and $C \vdash p : (\tau'; C')$ then $S; C \vdash V : C'$.*

PROOF. By induction on $v \sim p : V$. \square

LEMMA 4.11 MULTIPLE SUBSTITUTIONS. *If $S; C \vdash V : C'$ and $S; C, C' \vdash e : \tau$ then $S; C \vdash e[V] : \tau$.*

PROOF. By induction on the length of V , using Lemma 4.1 (variable substitution). \square

LEMMA 4.12 BASIC DECOMPOSITION. *If $S; C \vdash e : \tau$ and $\vec{F} : C$ and $C \vdash M : S$ and $v_{\text{pol}} = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$ then either:*

- e is a value v , or
- e can be decomposed into $E[e']$ such that one of the following is true.
 - $(\vec{F}, M, v_{\text{pol}}, e') \rightarrow_\beta (M', e'')$, for some M' and e''
 - $e' = \text{raise exn}$ and $E \neq E'[\text{try } E'' \text{ with } e'']$
 - $e' = \text{abort}$.

PROOF. By induction on the derivation of $S; C \vdash e : \tau$. \square

LEMMA 4.13 POLICY DECOMPOSITION. *If $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ then either:*

- e_{pol} is a value v_{pol} , or
- e_{pol} can be decomposed into $E[e]$ such that one of the following is true.
 - $(\vec{F}, M, \text{Triv}, e) \rightarrow_{\beta} (M', e')$, for some M' and e' , where Triv is the trivial policy defined in Figure 20
 - $e = \text{raise exn}$ and $E \neq E'[\text{try } E'' \text{ with } e'']$
 - $e = \text{abort}$.

PROOF. Immediate by Lemma 4.4 (inversion of $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$) and Lemma 4.12 (basic decomposition). \square

LEMMA 4.14 APPLICATION DECOMPOSITION. *If $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$ then either:*

- e_{app} is a value v_{app} , or
- e_{app} can be decomposed into $E[e]$ such that one of the following is true.
 - $(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_{\beta} (M', e')$, for some M' and e'
 - $e = \text{raise exn}$ and $E \neq E'[\text{try } E'' \text{ with } e'']$
 - $e = \text{abort}$.

PROOF. Immediate by Lemma 4.4 (inversion of $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$), Lemma 4.5 (canonical forms for $S; C \vdash v_{\text{pol}} : \text{Poly}$), and Lemma 4.12 (basic decomposition). \square

LEMMA 4.15 β PRESERVATION. *If $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$ and $\vdash \vec{F} : C$ and $C \vdash M : S$ and $(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_{\beta} (M', e'_{\text{app}})$ then there exists an S' extending S such that $S'; C \vdash e'_{\text{app}} : \tau$ and $C \vdash M' : S'$.*

PROOF. By induction on $(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_{\beta} (M', e'_{\text{app}})$. \square

THEOREM 4.16 PRESERVATION. *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 $\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau$.*

PROOF. Examination of the rules for $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ shows that either $e_{\text{pol}} = E[e]$ or $e_{\text{pol}} = v_{\text{pol}}$ (that is, either the policy or the application is being evaluated). The desired result holds in both cases. \square

Definition 4.17 Finished Programs. A program configuration $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$ is “finished” if and only if at least one of the following is true.

- e_{pol} and e_{app} are values
- $e_{\text{pol}} = E[\text{abort}]$ or $e_{\text{app}} = E[\text{abort}]$
- $e_{\text{pol}} = E[\text{raise exn}]$ or $e_{\text{app}} = E[\text{raise exn}]$, where $E \neq E'[\text{try } E'' \text{ with } e]$.

THEOREM 4.18 PROGRESS. *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 or there exists a program configuration $(\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ such that $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$.*

PROOF. By applying Lemma 4.13 (policy decomposition) to the assumption that $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$, we have either $e_{\text{pol}} = v_{\text{pol}}$ or $e_{\text{pol}} = E[e]$ such that $e = \text{raise exn}$ (where $E \neq E'[\text{try } E'' \text{ with } e'']$) or $e = \text{abort}$ or $(\vec{F}, M, \text{Triv}, e) \rightarrow_{\beta} (M', e')$. When $e_{\text{pol}} = E[e]$ such that $e = \text{raise exn}$ or $e = \text{abort}$, the program $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$ is finished. When $e_{\text{pol}} = E[e]$ and $(\vec{F}, M, \text{Triv}, e) \rightarrow_{\beta} (M', e')$, we have $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', E[e'], e_{\text{app}})$, as required.

When $e_{\text{pol}} = v_{\text{pol}}$, Lemma 4.14 (application decomposition) implies that either $e_{\text{app}} = v_{\text{app}}$ or $e_{\text{app}} = E[e]$ such that $e = \text{raise exn}$ (where $E \neq E'[\text{try } E'' \text{ with } e'']$) or $e = \text{abort}$ or $(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_{\beta} (M', e')$. All of these possibilities correspond to finished program configurations, except the case where $e_{\text{pol}} = v_{\text{pol}}$ and $e_{\text{app}} = E[e]$ and $(\vec{F}, M, v_{\text{pol}}, e) \rightarrow_{\beta} (M', e')$. In this case, we have $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e_{\text{pol}}, E[e'])$, as required. \square

5. RELATED WORK

We next compare and contrast closely related policy-specification-language efforts with Polymer.

Implemented Policy-specification Languages. A rich variety of expressive runtime policy-specification languages and systems has been implemented [Liao and Cohen 1992; Jeffery et al. 1998; Edjlali et al. 1998; Damianou et al. 2001; Erlingsson and Schneider 2000; 1999; Evans and Twyman 1999; Evans 2000; Robinson 2002; Kim et al. 1999; Bauer et al. 2003; Erlingsson 2003; Sen et al. 2004; Havelund and Roşu 2004]. These languages allow security engineers to write a centralized policy specification; the systems then use a tool to automatically insert code into untrusted target applications (i.e., they *instrument* the target application) in order to enforce the centrally specified policy on the target application. This centralized-policy architecture makes reasoning about policies a simpler and more modular task than the alternative approach of scattering security checks throughout application or execution-environment code. With a centralized policy, it is easy to locate the policy-relevant code and analyze or update it in isolation. Our fully implemented policy-specification language and enforcement system, Polymer, also applies this centralized-policy architecture.

The implementation efforts cited above thus deal with one part of the policy complexity problem—they ensure that policies exist in a centralized location rather than being dispersed throughout application or execution-environment code. The next step in dealing with the problem of policy complexity is to break complex, though centralized, policies into smaller pieces. Although some of the cited projects support limited policy decomposition via fixed sets of policy combinators [Evans 2000; Jeffery et al. 1998; Edjlali et al. 1998; Damianou et al. 2001; Bauer et al. 2003], they lack mechanisms to define new combinators that can arbitrarily modify previously written policies and dynamically create policies. By providing these abilities, Polymer enables users to break up complex centralized policies into simpler subpolicy modules.

In addition, none of the previous projects provide a methodology for making all policies composable. Problems arise when we try to compose general policies without constraining their effects. For example, one policy module may print an

ACM Journal Name, Vol. V, No. N, February 2007.

error message and then halt the target when it observes what it considers to be an illegal target action a , while another policy might print an acceptance message and allow the target to continue operating when it observes a . If we enforce the conjunction of these two policies, we expect the target to be halted when observing a , but what, if anything, should be printed? Polymer distinguishes itself from the related work cited above by providing a methodology for dealing with these kinds of conflicting effects when policies are composed.

Semantics of Policy-specification Languages. Of the implemented languages cited above, PoET/Pslang [Erlingsson and Schneider 2000; Erlingsson 2003] and Nacio [Evans and Twyman 1999] are the most closely related to Polymer because they support the specification of arbitrary imperative policies that contain both security state and methods to update security state when policy-relevant methods execute. A major difference between Polymer and these closely related projects, in addition to the differences noted above regarding policy composeability, is that Polymer provides a precise, formal semantics for its core language (Section 4). We consider the semantics an important contribution because it distills and unambiguously communicates the central workings of the Polymer language.

Recently, Krishnan has created a monitoring policy calculus based on the semantics of Polymer [Krishnan 2005]. He achieves simplicity by removing our compositionality constraint on policies (i.e., that all policies be separated into effectless query methods and effectful bookkeeping methods). Krishnan encodes most of our policy combinators into his calculus, gives our combinators formal semantics, states interesting properties about the combinators such as associativity, and explains how to encode several types of policies, including dynamically updateable policies, email policies similar to the one we have implemented in Polymer (Section 3.2), and privacy policies, in his calculus. Our own earlier work also provides precise semantics for some policy combinators, but in a less general and more complicated language [Bauer et al. 2003]. We showed how policies composed using a common, though fixed, set of combinators can be analyzed statically to ensure that

their effects do not conflict dynamically.

Aspect-oriented Languages. Our policy-specification language can also be viewed as an aspect-oriented programming language (AOPL) [Kiczales et al. 1996] in the style of AspectJ [Kiczales et al. 2001]. The main high-level differences between our work and previous AOPLs are 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 [Tucker and Krishnamurthi 2003; Walker et al. 2003] describe functional, as opposed to object-oriented, AOPLs with first-class aspect-oriented advice and formal semantics. However, they do not support aspect combinators like the ones we develop 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.

6. CONCLUSIONS

This article proposes a run-time policy-specification language centrally organized to enable and control expressive policy composition. This section briefly summarizes our contributions (Section 6.1) and enumerates several directions for future work (Section 6.2).

6.1 Summary

Polymer is a language for specifying complex run-time security policies as compositions of simpler subpolicy modules. Its benefits include all of the normal benefits of software modularity; for example, policies can be created, updated, reasoned about, and reused in isolation. We have developed a library of policy combinators and used them in a complex case-study application. Finally, we have defined a simple type-safe Polymer calculus, which formally defines and distills the key novelties of the Polymer language. Our language, libraries, and example policies are fully implemented and available for download from the Polymer project website [Bauer et al. 2005b].

6.2 Future Work

We conclude by discussing several open problems and opportunities for extension.

Transactional Policies. One might view the Polymer language as enabling policy composition via transactional policy updates. The separation of policies into effectless `query` and effectful `result` methods implements a form of rollback so that the highest-level superpolicy can commit to one suggestion atomically, without directly managing and rolling back subpolicy state and effects (which may be irrevocable).

In general, strong ties seem to exist between run-time policy enforcement and transactions [Ligatti et al. 2005]. It would be interesting to explore these ties further and to examine in exactly which ways languages with explicit transactional support (e.g., [Hindman and Grossman 2006]) further facilitate run-time policy specification and enforcement.

Concurrency. We have avoided providing direct support for concurrency in Polymer, instead leaving it up to policy writers to ensure that their policies are thread safe. In the future we plan to allow policies to turn on automatic locking mechanisms in the interpreter of the highest-level policy’s suggestions (see Figure 1); in this case the interpreter will obtain a lock before initiating a `query` and hold that lock until the corresponding `accept` method has returned (`result` methods must likewise be synchronized).

Combinator Analysis. We designed the Polymer language to permit arbitrary policy composition. This generality is useful because the definition of which combinators are the “right” ones to have available is user and application specific. For example, one set of combinators might be the minimal necessary to express all compositions of a common sort of policy (such as access-control policies), while different sets of combinators might be guaranteed to terminate or to satisfy other useful properties such as associativity and commutativity. Although Polymer permits general policy compositions, it would be interesting in the future to analyze particular sets of combinators and prove that they satisfy these sorts of properties. Krishnan has

already made progress on formalizing many of our combinators [Krishnan 2005].

Polymer GUI. Polymer policies, while expressive, have to be written at too low of a level (at the level of Java source code) to be convenient for many users who might benefit from creating custom policy compositions. An interesting avenue for future work would be to extend Polymer with a graphical user interface (GUI) that would allow, for example, system administrators to easily form provably safe policy hierarchies using prepackaged base policies and policy combinators. Polymer seems well suited to support this kind of interface because of its emphasis on compositionality and the ability to reuse libraries of policies and policy combinators. At the same time, holistic policy visualization (similar to what we show in Figure 13) can be very useful in understanding (and manipulating) a large policy in its entirety.

APPENDIX

This appendix contains proofs for nontrivial cases in the lemmas and theorems guaranteeing Polymer’s type safety.

LEMMA 4.8 CONTEXT DECOMPOSITION. *If $S; C \vdash E[e] : \tau$ then there exists a τ' such that $S; C \vdash E_{\tau'} : \tau$ and $S; C \vdash e : \tau'$.*

PROOF. By induction on the structure of E . We show the case for $E = \text{invk } E'$. The other cases are similar (except the case $E = []$, which is trivial).

Case $E = \text{invk } E'$:

By assumption,

$$E = \text{invk } E' \tag{1}$$

$$S; C \vdash E[e] : \tau \tag{2}$$

By (1) and (2),

$$S; C \vdash \text{invk } E'[e] : \tau \tag{3}$$

By (3) and Lemma 4.4 (inversion of the typing rule for `invk`),

$$S; C \vdash E'[e] : \text{Act} \tag{4}$$

By the inductive hypothesis and (4),

$$S; C \vdash E'_{\tau''} : \text{Act} \tag{5}$$

$$S; C \vdash e : \tau'' \quad (6)$$

By (5) and Definition 4.6 (well-typed context),

$$S; C, x : \tau'' \vdash E'[x] : \text{Act} \quad (x \notin FV(E')) \quad (7)$$

By (7) and the typing rule for `invk`,

$$S; C, x : \tau'' \vdash \text{invk } E'[x] : \text{Res} \quad (8)$$

By (1) and (8),

$$S; C, x : \tau'' \vdash E[x] : \text{Res} \quad (9)$$

By (9) and Definition 4.6 (well-typed context),

$$S; C \vdash E_{\tau''} : \text{Res} \quad (10)$$

By (3) and the typing rule for `invk`,

$$\tau = \text{Res} \quad (11)$$

Result is from (6), (10), and (11).

□

LEMMA 4.10 PATTERN TYPES. *If $S; C \vdash v : \tau'$ and $v \sim p : V$ and $C \vdash p : (\tau'; C')$ then $S; C \vdash V : C'$.*

PROOF. By induction on $v \sim p : V$. We show one case; the others are similar.

Case $p = \text{pol}(x_1, x_2, x_3)$:

By assumption,

$$S; C \vdash v : \tau' \quad (1)$$

$$v = \text{pol}(v_1, v_2, v_3) \quad (2)$$

$$V = v_1/x_1, v_2/x_2, v_3/x_3 \quad (3)$$

$$C' = x_1 : \text{Act} \rightarrow \text{Sug}, x_2 : (\text{Act}, \text{Sug}) \rightarrow (), x_3 : \text{Res} \rightarrow () \quad (4)$$

By (1), (2), and Lemma 4.4 (inversion of the typing rule for policies),

$$S; C \vdash v_1 : \text{Act} \rightarrow \text{Sug} \quad (5)$$

$$S; C \vdash v_2 : (\text{Act}, \text{Sug}) \rightarrow () \quad (6)$$

$$S; C \vdash v_3 : \text{Res} \rightarrow () \quad (7)$$

By (3), (4), (5), (6), and (7),

$$\forall x \in \text{dom}(C') \exists v . (v/x \in V \wedge S; C \vdash v : C'(x)) \quad (8)$$

By (8) and Definition 4.9 (well-typed substitutions),

$$S; C \vdash V : C' \quad (9)$$

Case $p = \text{act}(f, p')$:

By assumption,

$$S; C \vdash v : \tau' \quad (1)$$

$$v = \text{act}(f, v') \quad (2)$$

$$v' \sim p' : V \quad (3)$$

$$C \vdash p : (\text{Act}; C') \quad (4)$$

By Lemma 4.4 (inversion of the typing rule for (4)),

$$C \vdash p' : (\tau_1; C') \quad (\text{where } C(f) = \tau_1 \rightarrow \tau_2) \quad (5)$$

By (1), (2), and Lemma 4.4 (inversion of the typing rule for $\text{act}(f, v')$),

$$S; C \vdash v' : \tau_1 \quad (6)$$

By (3), (5), (6), and the inductive hypothesis,

$$S; C \vdash V : C' \quad (7)$$

□

LEMMA 4.15 β PRESERVATION. *If $\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau$ and $\vdash \vec{F} : C$ and $C \vdash M : S$ and $(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_{\beta} (M', e'_{\text{app}})$ then there exists an S' extending S such that $S'; C \vdash e'_{\text{app}} : \tau$ and $C \vdash M' : S'$.*

PROOF. By induction on $(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_{\beta} (M', e'_{\text{app}})$. We show the most difficult case here.

Case $e_{\text{app}} = \text{invk act}(f, v)$:

By assumption,

$$\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau \quad (1)$$

$$\vdash \vec{F} : C \quad (2)$$

$$C \vdash M : S \quad (3)$$

$$(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_{\beta} (M', e'_{\text{app}}) \quad (4)$$

$$e_{\text{app}} = \text{invk act}(f, v) \quad (5)$$

$$\text{fun}f(x:\tau_1):\tau_2\{e\} \in \vec{F} \quad (6)$$

$$e'_{\text{app}} = \text{Wrap}(v_{\text{pol}}, \text{fun}f(x:\tau_1):\tau_2\{e\}, v) \quad (7)$$

By (2), (3), and Lemma 4.4 (inversion of the typing rule for (1)),

$$S; C \vdash v_{\text{pol}} : \text{Poly} \quad (8)$$

$$S; C \vdash e_{\text{app}} : \tau \quad (9)$$

By (5), (9), and the typing rule for `invk`,

$$S; C \vdash e_{\text{app}} : \text{Res} \quad (10)$$

By (5), (10), and Lemma 4.4 (inversion of the typing rule for `invk`),

$$S; C \vdash \text{act}(f, v) : \text{Act} \quad (11)$$

By (11) and Lemma 4.4 (inversion of the typing rule for `act`(f, v)),

$$C(f) = \tau'_1 \rightarrow \tau'_2 \quad (12)$$

$$S; C \vdash v : \tau'_1 \quad (13)$$

By (6), (12), and Lemma 4.4 (inversion of the typing rule for (2)),

$$\tau'_1 = \tau_1 \quad (14)$$

$$\tau'_2 = \tau_2 \quad (15)$$

$$\cdot ; C, x : \tau_1 \vdash e : \tau_2 \quad (16)$$

By (12), (13), (14), and (15),

$$C(f) = \tau_1 \rightarrow \tau_2 \quad (17)$$

$$S; C \vdash v : \tau_1 \quad (18)$$

By (8) and Lemma 4.5 (canonical forms),

$$v_{\text{pol}} = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}) \quad (19)$$

By (8), (19), and Lemma 4.4 (inversion of the typing rule for `Poly`),

$$S; C \vdash v_{\text{query}} : \text{Act} \rightarrow \text{Sug} \quad (20)$$

$$S; C \vdash v_{\text{acc}} : (\text{Act}, \text{Sug}) \rightarrow () \quad (21)$$

$$S; C \vdash v_{\text{res}} : \text{Res} \rightarrow () \quad (22)$$

By (7), (16), (17), (18), (19), (20), (21), (22), and the definition of `Wrap`,

$$S; C \vdash e'_{\text{app}} : \text{Res} \quad (23)$$

By (9) and (10),

$$\tau = \text{Res} \quad (24)$$

By (3), (23), and (24),

$$S; C \vdash e'_{\text{app}} : \tau \wedge C \vdash M : S \quad (25)$$

Case $e_{\text{app}} = \text{case } v \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3)$ and $v \sim p : V$:

By assumption,

$$\vdash (\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) : \tau \quad (1)$$

$$\vdash \vec{F} : C \quad (2)$$

$$C \vdash M : S \quad (3)$$

$$(\vec{F}, M, v_{\text{pol}}, e_{\text{app}}) \rightarrow_{\beta} (M, e_2[V]) \quad (4)$$

$$v \sim p : V \quad (5)$$

$$e_{\text{app}} = \text{case } v \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3) \quad (6)$$

By (2), (3), and Lemma 4.4 (inversion of the typing rule for (1)),

$$S; C \vdash e_{\text{app}} : \tau \quad (7)$$

By (6), (7), and Lemma 4.4 (inversion of the typing rule for case expressions),

$$S; C \vdash v : \tau' \quad (8)$$

$$C \vdash p : (\tau'; C') \quad (9)$$

$$S; C, C' \vdash e_2 : \tau \quad (10)$$

By (5), (8), (9), and Lemma 4.10 (pattern types),

$$S; C \vdash V : C' \quad (11)$$

By (10), (11), and Lemma 4.11 (multiple substitutions),

$$S; C \vdash e_2[V] : \tau \quad (12)$$

Result is from (3), (4), and (12).

□

THEOREM 4.16 PRESERVATION. *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 $\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau$.*

PROOF. Examination of the rules for $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$ shows that either $e_{\text{pol}} = E[e]$ or $e_{\text{pol}} = v_{\text{pol}}$ (that is, either the policy or the application is
ACM Journal Name, Vol. V, No. N, February 2007.

being evaluated). The two cases are similar; we show the first case.

Case $e_{\text{pol}} = E[e]$:

By assumption,

$$\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau \quad (1)$$

$$e_{\text{pol}} = E[e] \quad (2)$$

$$(\vec{F}, M, \text{Triv}, e) \rightarrow_{\beta} (M', e') \quad (3)$$

$$e'_{\text{pol}} = E[e'] \quad (4)$$

$$e_{\text{app}} = e'_{\text{app}} \quad (5)$$

By (2) and Lemma 4.4 (inversion of the typing rule for (1)),

$$\vdash \vec{F} : C \quad (6)$$

$$C \vdash M : S \quad (7)$$

$$S; C \vdash E[e] : \text{Poly} \quad (8)$$

$$S; C \vdash e_{\text{app}} : \tau \quad (9)$$

By (8) and Lemma 4.8 (context decomposition),

$$S; C \vdash E_{\tau'} : \text{Poly} \quad (10)$$

$$S; C \vdash e : \tau' \quad (11)$$

By the definition of Triv,

$$\cdot ; \cdot \vdash \text{Triv} : \text{Poly} \quad (12)$$

By (5) and (9),

$$S; C \vdash e'_{\text{app}} : \tau \quad (13)$$

By (6), (7), (11), (12), and the typing rule for program configurations,

$$\vdash (\vec{F}, M, \text{Triv}, e) : \tau' \quad (14)$$

By (3), (6), (7), (14), and Lemma 4.15 (β preservation),

$$S' \text{ extends } S \quad (15)$$

$$S'; C \vdash e' : \tau' \quad (16)$$

$$C \vdash M' : S' \quad (17)$$

By (10), (15), Definition 4.6 (well-typed context), and Lemma 4.3 (weakening),

$$S'; C \vdash E_{\tau'} : \text{Poly} \quad (18)$$

By (4), (16), (18), and Lemma 4.7 (well-typed, filled context),

$$S'; C \vdash e'_{\text{pol}} : \text{Poly} \quad (19)$$

By (13) and Lemma 4.3 (weakening),

$$S'; C \vdash e'_{\text{app}} : \tau \quad (20)$$

By (6), (17), (19), (20), and the typing rule for program configurations,

$$\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau \quad (21)$$

□

REFERENCES

- AGESEN, O., FREUND, S. N., AND MITCHELL, J. C. 1997. Adding type parameterization to the Java language. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*. Apache Software Foundation 2003. *Byte Code Engineering Library*. Apache Software Foundation. <http://jakarta.apache.org/bcel/>.
- BAUER, L., APPEL, A. W., AND FELTEN, E. W. 2003. Mechanisms for secure modular programming in Java. *Software—Practice and Experience* 33, 5, 461–480.
- BAUER, L., LIGATTI, J., AND WALKER, D. 2003. Types and effects for non-interfering program monitors. In *Software Security—Theories and Systems. Next-NSF-JSPS International Symposium, ISSS 2002, Tokyo, Japan, November 8-10, 2002, Revised Papers*, M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, Eds. Lecture Notes in Computer Science, vol. 2609. Springer.
- BAUER, L., LIGATTI, J., AND WALKER, D. 2005a. Composing security policies with polymer. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*. Chicago.
- BAUER, L., LIGATTI, J., AND WALKER, D. 2005b. Polymer: A language for composing run-time security policies. <http://www.cs.princeton.edu/sip/projects/polymer/>.
- DAMIANOU, N., DULAY, N., LUPU, E., AND SLOMAN, M. 2001. The Ponder policy specification language. *Lecture Notes in Computer Science* 1995, 18–39.
- EDJLALI, G., ACHARYA, A., AND CHAUDHARY, V. 1998. History-based access control for mobile code. In *ACM Conference on Computer and Communications Security*. 38–48.
- ERLINGSSON, Ú. 2003. The inlined reference monitor approach to security policy enforcement. Ph.D. thesis, Cornell University.
- ERLINGSSON, Ú. AND SCHNEIDER, F. B. 1999. SASI enforcement of security policies: A retro-ACM Journal Name, Vol. V, No. N, February 2007.

- spective. In *Proceedings of the New Security Paradigms Workshop*. Caledon Hills, Canada, 87–95.
- ERLINGSSON, Ú. AND SCHNEIDER, F. B. 2000. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*. Oakland, CA.
- EVANS, D. 2000. Policy-directed code safety. Ph.D. thesis, Massachusetts Institute of Technology.
- EVANS, D. AND TWYMAN, A. 1999. Flexible policy-directed code safety. In *IEEE Security and Privacy*. Oakland, CA.
- HAVELUND, K. AND ROŞU, G. 2004. Efficient monitoring of safety properties. *International Journal on Software Tools for Technology Transfer (STTT)* 6, 2 (Aug.), 158–173.
- HINDMAN, B. AND GROSSMAN, D. 2006. Strong atomicity for java without virtual-machine support.
- IGARASHI, A., PIERCE, B., AND WADLER, P. 1999. Featherweight Java. In *ACM conference on Object-Oriented Programming, Systems, Languages and Applications*. Denver, CO, 132–146.
- JEFFERY, C., ZHOU, W., TEMPLER, K., AND BRAZZELL, M. 1998. A lightweight architecture for program execution monitoring. In *Program Analysis for Software Tools and Engineering (PASTE)*. ACM Press, 67–74.
- KICZALES, G., HILSDALE, E., HUGUNIN, J., KERSTEN, M., PALM, J., AND GRISWOLD, W. 2001. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag.
- KICZALES, G., IRWIN, J., LAMPING, J., LOINGTIER, J.-M., LOPES, C. V., MAEDA, C., AND MENDHEKAR, A. 1996. Aspect-oriented programming. *ACM Comput. Surv.* 28, 4es, 154.
- KIM, M., VISWANATHAN, M., BEN-ABDALLAH, H., KANNAN, S., LEE, I., AND SOKOLSKY, O. 1999. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*. York, UK.
- KRISHNAN, P. 2005. A monitoring policy calculus. Tech. Rep. CSA-05-01, Bond University.
- LIAO, Y. AND COHEN, D. 1992. A specification approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng.* 18, 11, 969–978.
- LIGATTI, J., BAUER, L., AND WALKER, D. 2005. Enforcing non-safety security policies with program monitors. In *10th European Symposium on Research in Computer Security (ESORICS)*. Milan, Italy.
- MCGRAW, G. AND FELTEN, E. W. 1999. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, Inc., New York, NY, USA.
- MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press.
- PETERSEN, A. 2003. Pooka: A Java email client. <http://www.suberic.net/pooka/>.

- ROBINSON, W. 2002. Monitoring software requirements using instrumented code. In *HICSS '02: Proceedings of the 35th Annual Hawaii International Conference on System Sciences (HICSS'02)-Volume 9*. IEEE Computer Society, Washington, DC, USA, 276.2.
- SALTZER, J. H. AND SCHROEDER, M. D. 1975. The protection of information in computer systems. In *IEEE 63, 9*. 1278–1308.
- SEN, K., VARDHAN, A., AGHA, G., AND ROSU, G. 2004. Efficient decentralized monitoring of safety in distributed systems. In *26th International Conference on Software Engineering (ICSE'04)*. 418–427.
- TUCKER, D. B. AND KRISHNAMURTHI, S. 2003. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. 158–167.
- WALKER, D., ZDANCEWIC, S., AND LIGATTI, J. 2003. A theory of aspects. In *ACM International Conference on Functional Programming*. Uppsala, Sweden.