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 that 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 composable, 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 composable policies on Java applications. We formalize the central workings of Polymer by defining an unambiguous semantics for our language. Using this formalization, we state and prove an uncircumventability theorem, which guarantees that monitors will intercept all security-relevant actions of untrusted software.

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 that 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 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.

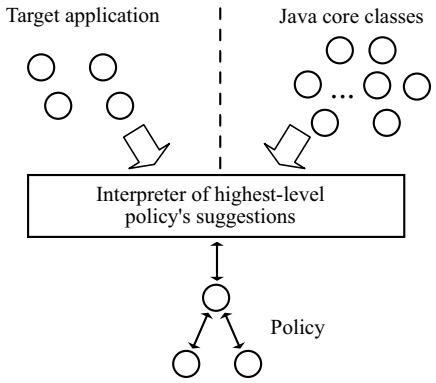

Fig. 1. A secure Polymer application

(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 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. We also used the combinators to design complex policies for text-editor and media-player applications.
(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. The formal semantics serves as the unambiguous, canonical definition of the Polymer language. We prove that our language is type safe, a necessary property for protecting the program monitor's

state from untrusted applications, and that applications cannot circumvent monitoring code before executing security-relevant methods.

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] (and prove so for the formal Polymer calculus). 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 in greater detail how to write policies using Polymer. We present more examples of Polymer policies (e.g., in the new Section 3.3) and 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 discuss, in the new Section 3.4, the ways in which Polymer facilitates, and at times hinders, the design and implementation of policies in practice. Our conclusions are based on our experience using Polymer and are substantiated by the unabbreviated code examples we show throughout the article.
—We present the complete semantics and type-safety proof for the Polymer language in Section 4 and the appendices.
—We prove an additional property of the formal Polymer language, uncircumventability of monitors, in Section 4.5.
—We also make numerous minor revisions to the text throughout, including the addition of a brief discussion of future work in Section 6.2.

Roadmap. Section 2 introduces 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 introduces the Polymer language. We begin with the basic concepts (Section 2.1) and show how to program simple policies (Section 2.2). Then, we

```
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

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 wildcards: * matches any one constraint (e.g., any return type or any single parameter type), and .. matches zero or more parameter types. For example, the pattern

$$\langle \text{public void java.io.*.}\langle \text{init} \rangle (\text{int, ..}) \rangle$$

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

Action patterns appear in two places. First, the action declaration file is a set of action patterns. During the instrumentation process, every action that matches an action pattern in the action declaration file is instrumented. Second, policies use action patterns in aswitch statements to determine 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).

We distinguish between *irrelevant* and *OK* suggestions for 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. XACML also distinguishes between its NotApplicable and Permit responses to security-relevant method invocations [**?**].

Figure 3 shows our Sug class. The class contains convenience methods for dynamically 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

```
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

```
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

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 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

```
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

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.

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 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

```
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()

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 abstraction, called platform interfaces, that supports a similar sort of separation between concrete and

```
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

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 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 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,

```
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;
  }
}
```

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

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

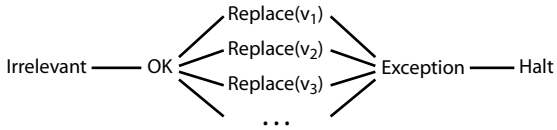

Fig. 10. Lattice ordering of Polymer suggestions' semantic impact

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. In related work, Krishnan provides formal semantics for several of our combinators [Krishnan 2005].

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 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

```
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);
    }
  }
}
```

Fig. 11. A conjunctive policy combinator

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

```
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.isIrr())
         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

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 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 sets 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 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 commonly used 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

Implementing and using Polymer has been instrumental in confirming and refining its design. In this section we describe Polymer's implementation (Section 3.1) and a fully implemented email-client policy (Section 3.2). We then discuss additional case-study policies (Section 3.3) and summarize ways in which Polymer facilitates and sometimes hinders policy design and implementation (Section 3.4).

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 for a particular action declaration file.

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. Thus, 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 (outlined in Figure 13) to secure untrusted email clients that use the JavaMail API. 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 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, and the Audit, Conjunction, TryWith, Dominates, and AutoUpdate superpolicies in Section 2.3. The left branch of the policy hierarchy (shaded in Figure 13) describes a generic policy

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

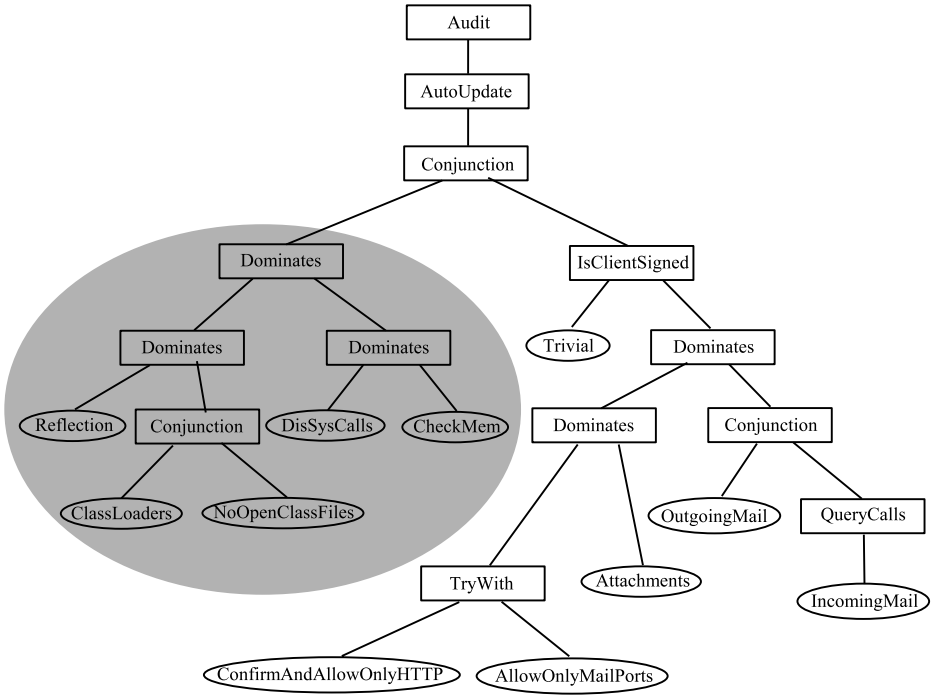

Fig. 13. Email-client policy hierarchy

that we include in all of our high-level Polymer policies. This branch 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, 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

```
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

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. In the email-client policy, QueryCalls is needed above IncomingMail (shown in Figure 8) so that the invocation of getSubject in IncomingMail's spamifySubject method does not get intercepted by IncomingMail, which would cause a recursive invocation of spamifySubject.

—OutgoingMail logs all mail being sent, pops up a window confirming the recipients of messages (to prevent a malicious client from quietly sending mail on the user's behalf), backs up every outgoing message by sending a BCC to polydemo@cs.princeton.edu, and automatically appends contact information to textual messages.

—IncomingMail was shown in an abbreviated form in Figure 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.

The entire email-client policy is specified in 1539 lines of Polymer code (excluding empty lines and comments). To get an idea of how many lines of code need to be written to specify a new policy, it is instructive to consider the size of the reusable components of the email policy: the combinators (described in Section 2.3) are specified in 410 lines; the shaded anti-tampering branch (which includes the Dominates and Conjunction combinators) in 356 lines; and the abstract actions for opening files and network connections in 225 lines.

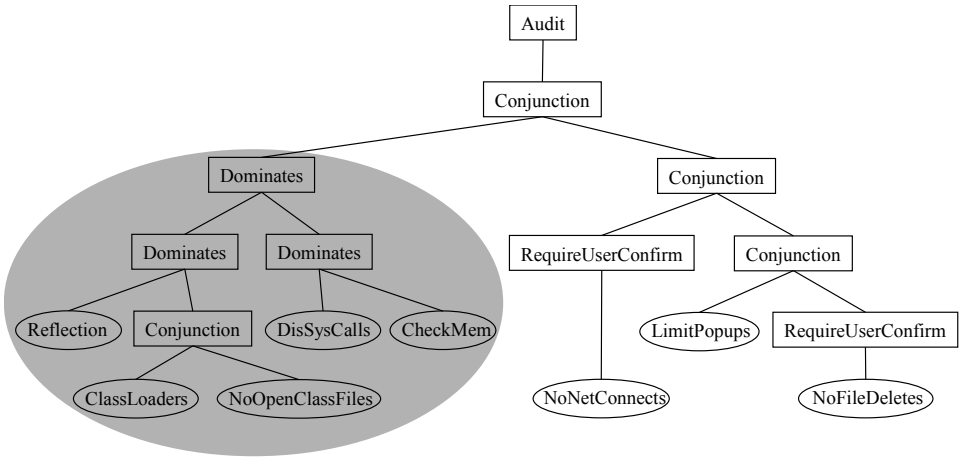

Fig. 15. Text-editor policy hierarchy

3.3 Additional Case Studies

We next describe two additional case-study policies, which are fully designed but not implemented, and highlight the effect of complete mediation on their designs. As we will see, complete mediation is a powerful tool required when enforcing some policies; however, policy designers must take care to consider complete mediation to ensure that their designs are consistent.

Text-editor Policy. Many reasonable restrictions could be placed on untrusted text editors; we design a policy that makes the following restrictions.

—To protect privacy, all network connections must be confirmed with the user.

—To prevent the editor from maliciously operating on the file system, all file deletions must be confirmed with the user.

—To prevent "window spam", the editor application may never have more than four windows open simultaneously.

Figure 15 contains a policy hierarchy designed to enforce these text-editor restrictions. The hierarchy contains the same generic, shaded branch as the email-client hierarchy of Figure 13; as described in Section 3.2, this branch is a policy that prevents the target application from circumventing the monitor via reflection, etc. The rest of the policy hierarchy specifies text-editor-specific restrictions as follows.

—NoNetConnects and NoFileDeletes suggest halting the target application before any network connections or file deletions occur.

—RequireUserConfirm is a policy modifier that returns whatever suggestions its subpolicy returns, except when both of the following conditions hold: (1) the subpolicy suggests halting the application or raising a security exception; and (2) the user then confirms, in a pop-up window, that the action triggering the HaltSug or ExnSug is in fact OK to execute. In this case, RequireUserConfirm OKs the action instead of halting or raising a security exception.

—LimitPopups limits to four the number of windows open at any time. If a fifth window is attempted to be opened, LimitPopups halts the target application.

In addition to reusing the entire shaded branch of the email-client policy and the Conjunction and Audit combinators, the text-editor policy also borrows the abstract actions for opening files and network sockets that we developed for the email-client policy. These two abstract actions are likely to be useful for a wide range of policies, even when—as is the case for the email-client and text-editor policies—policies place different constraints on the behaviors that those abstract actions describe.

The text-editor policy is a good illustration of the idea, discussed in Section 2.1, that policies become composeable when they contain effectless query methods. The NoNetConnects and NoFileDeletes policies do not immediately halt the application's execution in their query methods; this effect gets postponed via HaltSugs, enabling the RequireUserConfirm superpolicy to analyze the suggestions of the NoNetConnects and NoFileDeletes subpolicies. RequireUserConfirm can safely disobey suggestions of NoNetConnects and NoFileDeletes because neither subpolicy executes observable, effectful computation in generating those suggestions. If NoNetConnects and NoFileDeletes were modified to directly halt the target application in their query methods, then the policies would not be composeable with the RequireUserConfirm superpolicy because RequireUserConfirm would have no way to undo the effects of its subpolicies when disobeying their suggestions.

Another interesting aspect of effect composition in the text-editor policy relates to the window-pop-up effects. The two RequireUserConfirm policies may insert actions to pop up windows, while the LimitPopups policy may forbid those same pop-ups and require application termination when they occur. We want LimitPopups to behave this way in order to prevent a text editor from annoying users by, for example, spawning a hundred threads, each of which opens a network socket to trigger the RequireUserConfirm policy to open a confirmation window. Hence, complete mediation, particularly the ability to monitor even policy code, is necessary to enforce the desired policy (if that policy is to reuse the RequireUserConfirm subpolicy as a component). Other major run-time policy-enforcement systems, including PoET/PSLang [Erlingsson and Schneider 2000] and Naccio [Evans and Twyman 1999; Evans 2000], treat policy code as unmonitorable. Because these other enforcement systems do not adhere to complete mediation, they cannot enforce the text-editor policy as described above.

Media-player Policy. As a final policy example, we consider a media-player policy hierarchy designed to restrict untrusted media players in the following ways.

—To protect privacy, the media player is not permitted to "phone home", that is, to make a connection to any network operated by the media player's creators. Some media-player applications shipped with audio CDs have recently been shown to contain spyware that phones home to report every use of the DRM-protected CD [Halderman and Felten 2006].

—To prevent malicious access to the file system, the media player is not permitted to write to files, delete files, or open any file with a nonmedia file type (e.g., an executable file).

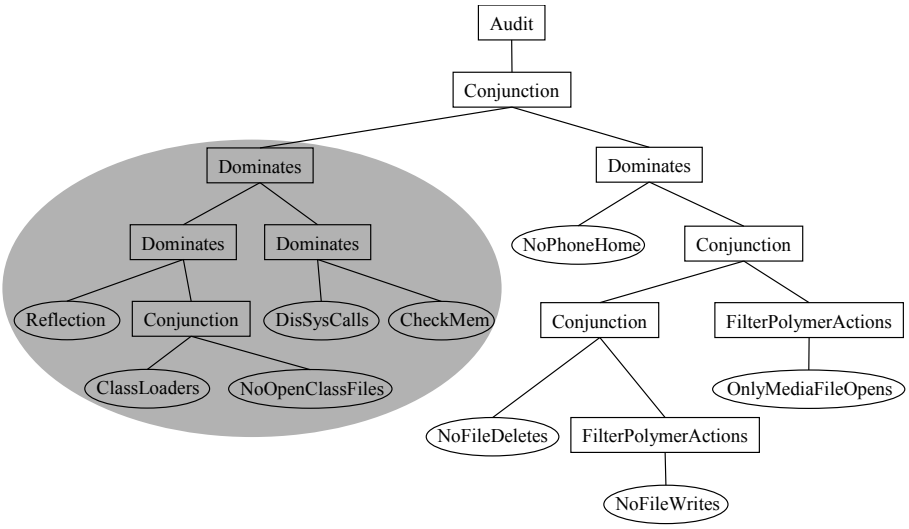

Fig. 16. Media-player policy hierarchy

Figure 16 outlines the policy hierarchy designed to enforce these media-player restrictions. Again, the hierarchy specifies the same generic, shaded branch as the earlier policy-hierarchy examples. The other high-level branch specifies restrictions on the media player's access to network sockets and the file system.

—NoPhoneHome restricts applications from making network connections directly to IP addresses assigned to the application's provider (though it does not prevent connections to IP addresses of entities that may be in collusion with the provider). The policy obtains the application provider's IP addresses through a DNS search of the provider's name, which the policy obtains either from the application's signed JAR file or from the provider's name passed as a parameter to the policy's constructor.

—NoFileDeletes and NoFileWrites respond to applications' attempts to delete and write files by raising security exceptions.

—OnlyMediaFileOpens allows applications to open only those files that have fixed media types, such as mpeg and mp3 (these types are supplied as a whitelist to the policy's constructor). The policy raises security exceptions in response to nonmedia-file opens.

—FilterPolymerActions automatically OKs all actions invoked by Polymer policies, enabling its subpolicy to only monitor actions invoked by application and library code (but not Polymer code). Similar to the QueryCalls policy described in Section 3, FilterPolymerActions performs stack inspection to determine whether actions were invoked by Polymer code. FilterPolymerActions defers to its subpolicy for suggestions and state updates in response to actions not invoked from Polymer code.

—Finally, the NoPhoneHome policy is composed with the media-player file-access-control policies using a Dominates combinator as an efficient form of Conjunction (as described in Section 2.3).

This media-player policy example illustrates an important policy-design concept: When incorporating a policy module into a global policy specification, one must consider whether the module being inserted is *consistent* with the other modules. Inconsistencies occur when one module forbids actions required by another module. Imprecise specifications of forbidden actions typically cause inconsistencies; rather than an imprecise (and incorrect) specification that all of a certain type of action should be forbidden, the actual intended policy is often that those actions should be forbidden only when not invoked from (or allowed by) another policy. Essentially, policy designers must be mindful of whether actions forbidden to be invoked by applications (and library) code should be allowed to be executed by Polymer code.

For example, without including the FilterPolymerActions policy in the media-player example, our NoFileWrites and OnlyMediaFileOpens policies would be globally inconsistent with the Audit policy because Audit requires opening and writing to a log file (which has a non-media file type), while NoFileWrites and OnlyMediaFileOpens prevent those actions. We resolve the global inconsistency simply by inserting FilterPolymerActions above the NoFileWrites and OnlyMediaFileOpens policies, so that Polymer code may open and write to non-media-type files.

Having to consider and resolve these sorts of inconsistencies is the price of using a system that adheres to complete mediation. Although obeying complete mediation makes the Polymer system very expressive, permitting policies like the text-editor policy described above to be enforced, Polymer policy designers must consider, for every policy included in a policy design, whether Polymer-invoked actions should be trusted. Our experience indicates that it is not difficult for a policy designer to get into the habit of considering inconsistencies during policy design and resolving them by inserting the appropriate filters.

Finally, we note that a knowledgeable policy designer could optimize the media-player policy hierarchy by coalescing duplicate FilterPolymerActions policies to a single parent node, as shown in Figure 17. This is an optimization because it causes FilterPolymerActions to execute only once for every two executions in the original hierarchy of Figure 16. The optimization is safe because it never over-filters; that is, no harm is done by filtering Polymer actions to the NoFileDeletes policy in this case because the rest of the policy never attempts to delete files anyway.

3.4 Experiential Observations

This subsection describes our experiences designing the case-study Polymer policies of Sections 3.2 and 3.3 and implementing the email-client case-study policy of Section 3.2. We discuss the ways in which Polymer facilitates and sometimes hinders policy design and implementation.

Tasks Polymer Facilitates. As expected, we found Polymer most useful in modularizing and composing policies. The ability to write modules of isolated policy concerns and to reuse existing policies made complex-policy design and implementation tractable. An example of such reuse at a high level is the shaded anti-tampering policy that we found useful to include as part of all our case studies. An example of low-level reuse are the Conjunction and Dominates combinators, each of which we typically used multiple times in constructing any policy.

In our case studies we also took advantage of Polymer's ability to specify new

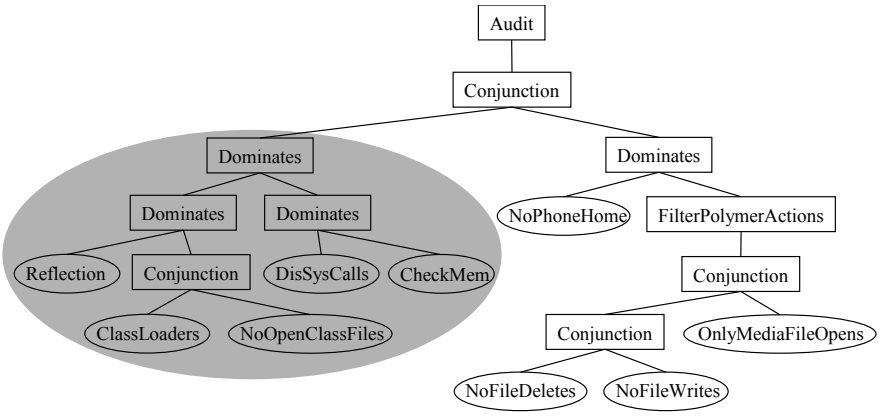

Fig. 17. Optimized version of media-player policy hierarchy shown in Figure 16

combinators. Although we commonly used the more standard Conjunction and Dominates combinators, we also found it necessary to define new combinators such as TryWith, AutoUpdate, and RequireUserConfirm. The need for these combinators would have been hard to predict until considering specific policies, yet once implemented they appeared applicable to other policies as well.

Beyond policy modularity and composability, Polymer was designed for expressivity to facilitate specifying a rich set of policies. Polymer provides complete mediation for policies that require monitoring of policy code (e.g., the text-editor policy of Figure 15) and ReplSugs for policies that require replacing calls to security-relevant methods with precomputed return values (e.g., the IncomingMail policy of Figure 8). We found both of these provisions essential for implementing practical policies.

Finally, more minor features such as pattern matching for actions and many-many mappings from concrete to abstract actions also helped make policy specification convenient, as the code examples we have presented throughout this article show.

Tasks Polymer Hinders. In our experience, Polymer hinders policy design and implementation in two major ways. First, designers must take care to avoid the sorts of policy inconsistencies described in Section 3.3. Having to consider when to use policy filters places an additional burden on policy designers, but we believe the increased policy expressiveness provided by complete mediation outweighs the inconvenience of having to consider appropriate policy filters during policy design.

The second major hindrance introduced by Polymer is simply the price of universal policy composability, that is, having to implement policies in terms of effectless query methods. In our experience, withholding effects from query methods requires care and can lead to relatively convoluted policy logic. For instance, the DisSysCalls policy of Figure 6 can be written more simply and straightforwardly, as shown in Figure 18. Tools like PoET/PSLang [Erlingsson and Schneider 2000] and Naccio [Evans and Twyman 1999] let users specify policies similarly, but this prevents them from being safely composed. For example, the policy in Figure 18 is not immediately composable because combinators cannot disobey its suggestions to halt

```
public class DisSysCalls extends Policy {
  public Sug query(Action a) {
    aswitch(a) {
      case ⟨* java.lang.Runtime.exec(..)⟩:
        System.err.println("Illegal exec method called");
        System.err.println("About to halt target");
        return new HaltSug(this, a);
    }
    return new IrrSug(this, a);
  }
}
```

Fig. 18. Simple but not-immediately-composable version of the policy in Figure 6

until after its halting error messages have already been printed. Again, we believe the benefits of having modular, composable policies outweighs the inconvenience of requiring policies to contain effectless query methods.

4. FORMAL SEMANTICS OF THE POLYMER LANGUAGE

We next precisely define the Polymer language by providing a complete, formal semantics for the language. This semantics—as opposed to the particular Java implementation discussed in Sections 2 and 3—serves as the unambiguous, canonical definition of the Polymer language. Hence, the semantics is a core contribution of this article and, as far as we are aware, is the first complete formal semantics for a rich run-time policy-specification language. The semantics highlights the simplicity of the core features of the Polymer language, and we hope that the semantics will be useful in defining future policy-specification languages, as it handles non-trivial challenges (such as how to model the separation of security-relevant and security-irrelevant methods) applicable to many policy-specification languages.

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 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 of the target language protects the program monitor's state and code from being tampered with by the untrusted application).

4.1 Syntax

Figure 19 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

Types	τ	::=	$\text{Bool} \mid (\vec{\tau}) \mid \tau\ \text{Ref} \mid \tau_1 \to \tau_2 \mid \text{Poly} \mid \text{Sug} \mid \text{Act} \mid \text{Res}$
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. 19. Abstract syntax for the Polymer calculus

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 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 patterns p includes variable patterns x as well as patterns for matching constructors. Ordinary, unmonitored functions are executed via the usual function application command $(e_1\ e_2)$.

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

[2] As 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 $(\lambda x{:}\tau.e_2)e_1$.

takes an action to execute as an argument. Each suggestion will be given a unique interpretation in the operational semantics.

4.2 Static Semantics

Figure 20 presents the most interesting rules for the language's static semantics. The more standard rules are shown in Figures 23 and 24 in Appendix A. 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.

$$\begin{array}{lll} \text{Label stores} & S & ::= \cdot \mid S, l : \tau \\ \text{Variable contexts} & C & ::= \cdot \mid C, x : \tau \mid C, f : \tau \end{array}$$

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 20 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 20 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 20) 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 20), which give types to a library of monitored functions $\vec{F}$ and types to locations in memory M.

The static semantics for case expressions and pattern matching, as well as the remaining straightforward rules for standard expressions, is shown in Appendix A.

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-reduction can occur. Our contexts specify a left-to-right, call-by-value evaluation order, as shown in Figure 21.

We specify execution through a pair of judgments, one for top-level evaluation (shown in the top of Figure 22) and one for basic reductions (shown in the bottom of Figure 22 and in Figures 25 and 26 in Appendix B). 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. We also define a standard multi-step relation $\mapsto^*$ as the reflexive, transitive closure of the top-level single-step relation.

Execution of many of the constructs is relatively straightforward. One exception

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

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

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

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

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

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

$$\frac{\begin{array}{c} \vdash \vec{F} : C \qquad C \vdash M : S \\ S; C \vdash e_{\text{pol}} : \text{Poly} \qquad S; C \vdash e_{\text{app}} : \tau \end{array}}{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$$

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

$$\frac{\begin{array}{c} \vec{F} = \text{fun} f_1(x_1{:}\tau_1){:}\tau_1'\{e_1\}, ..., \text{fun} f_n(x_n{:}\tau_n){:}\tau_n'\{e_n\} \\ C = f_1 : \tau_1 \to \tau_1', ..., f_n : \tau_n \to \tau_n' \\ \forall i \in \{1..n\} \;.\; \cdot\,; C, x_i : \tau_i \vdash e_i : \tau_i' \end{array}}{\vdash \vec{F} : C}$$

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

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

Fig. 20. Static semantics (rules for policies, suggestions, actions, and programs)

Evaluation contexts

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

Fig. 21. Evaluation contexts

is execution of function application, the rules for which are given in the bottom half of Figure 22. For ordinary functions, we use the usual capture-avoiding substitution. Monitored functions, on the other hand, may only be executed if they are wrapped up as actions and then invoked using the invk command. The invk command applies the query method to discover the suggestion the current policy makes and then interprets the suggestion. Notice, for instance, that to respond to the irrelevant suggestion (irrs), the application simply proceeds to execute the body

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

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

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

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

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

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

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

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

```
  let s = v_query(act(f, v)) in
  case s of
     irrs       ⇒  let x = v in result(e:τ2)
   | oks        ⇒  v_acc(act(f, v), s);
                   let x = v in let r = result(e:τ2) in v_res r;  r
   | repls(r)   ⇒  v_acc(act(f, v), s);  r
   | exns       ⇒  v_acc(act(f, v), s);  raise exn
   | inss(a)    ⇒  v_acc(act(f, v), s);  v_res(invk a);  invk act(f, v)
   | _          ⇒  abort
```

Fig. 22. Dynamic semantics (policy and target steps; beta steps for functions)

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 straightforward and appear in Figures 25 and 26 in Appendix B.

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 uncovers 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 protected only 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 Language Properties

We have proven two important properties of the Polymer language: it is type safe, and well-typed programs cannot circumvent monitoring code.

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. The theorems are stated below; proofs appear in Appendix C.

THEOREM 4.1 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* $\vdash (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}}) : \tau$.

Definition 4.2 *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 4.3 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* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (\vec{F}, M', e'_{\text{pol}}, e'_{\text{app}})$.

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 is also useful for proving that well-typed programs obey complete mediation—that control always properly flows to the monitoring code before any security-sensitive application method executes. We discuss such *uncircumventability* of monitoring code next.

Uncircumventability. Unlike in the Java implementation of Polymer, policies written in our formal language do not have to rely on auxiliary policies to guarantee that monitors will intercept all security-relevant method calls (auxiliary policies are necessary in the Java implementation because of Java's class-loading and reflection capabilities). We have shown that programs written in the Polymer calculus adhere to the principle of complete mediation by proving the following uncircumventability theorem (the proof is in Appendix D).

THEOREM 4.4 UNCIRCUMVENTABILITY. *If* $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ *and* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto^* (\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)])$ *then there exist functions* v_{query}, v_{acc}, *and* v_{res} *and context* E' *such that:*

(1) $v_{\text{pol}} = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$,
(2) $(\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)]) \mapsto (\vec{F}, M', v_{\text{pol}}, E'[v_{\text{query}}\ (\text{act}(f, v))])$, *and*

(3) *for all programs* P *such that* $(\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)]) \mapsto P$, *it must be the case that* $P = (\vec{F}, M', v_{\text{pol}}, E'[v_{\text{query}}\ (\text{act}(f, v))])$.

This uncircumventability theorem states that well-typed programs, after having evaluated the top-level policy value, *always* execute the top-level policy's query method before any security-relevant method. In other words, the program's policy value always gets to interpose and judge how monitored methods should execute before those monitored methods actually begin executing. Uncircumventability follows directly from the language's semantics and type safety. Although uncircumventability is required for enforcement systems to be secure and effective, as far as we are aware, Polymer's is the first uncircumventability theorem proven for a policy-specification language.

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 run-time 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. XACML supports the creation of new combinators (called "combining algorithms") [**?**], but, like the other related systems, it does not facilitate safe composition of general policies because it does not constrain the use of effects. For example, none of the cited enforcement systems, even if extended to obey complete mediation, could directly implement the RequireUserConfirm combinators of the text-editor policy in Figure 15 because the application-halting effects of the NoNetConnects and NoFileDeletes policies could

not be undone by the RequireUserConfirm policies. In contrast, Polymer makes the NoNetConnects and NoFileDeletes policies combinable with the RequireUserConfirm policies by requiring that all policies make suggestions in an effectless query method invocable from *any* superpolicy. In general, no combinator that sometimes disobeys its subpolicies' suggestions is directly implementable in previous work on similar policy-specification languages, although such combinators are implementable in Polymer.

Semantics of Policy-specification Languages. Of the implemented languages cited above, PoET/Pslang [Erlingsson and Schneider 2000; Erlingsson 2003] and Naccio [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 composability, 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 several complex case-study applications. Finally, we have defined a simple Polymer calculus, which formally defines and distills the key novelties of the Polymer language, and have proven soundness and uncircumventability properties of the calculus. 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.

Acknowledgements

Many thanks to the anonymous ACM TOSEM reviewers for their helpful and prompt comments. We are grateful to Greg Morrisett for valuable feedback on earlier versions of this paper. We are also thankful to Frank Piessens, Bart De Win, and other members of the Katholieke Universiteit Leuven community for insightful discussions that honed our understanding of complete mediation's effects on Polymer (cf. Section 3.3).

This research was supported in part by National Science Foundation grants CNS-0716343, CNS-0716216, CCR-0238328, and CCR-0306313, by ARDA grant NBCHC030106, by the Army Research Office through grant DAAD19-02-1-0389, and by a Sloan Fellowship.

REFERENCES

Agesen, O., Freund, S. N., and Mitchell, J. C. 1997. Adding type parameterization to the Java language. In *Object Oriented Programing: 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. Mext-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 retrospective. 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.

Halderman, J. A. and Felten, E. W. 2006. Lessons from the Sony CD DRM episode. In *Proceedings of the 15th USENIX Security Symposium*. 77–92.

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 Brazell, 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 specificational approach to high level program monitoring and measuring. *IEEE Trans. Softw. Eng. 18,* 11, 969–978.

Ligatti, J. 2006. Policy enforcement via program monitoring. Ph.D. thesis, Princeton University.

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.

OASIS. 2005. *eXtensible Access Control Markup Language (XACML) version 2.0*. `http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

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.

A. ADDITIONAL RULES OF STATIC SEMANTICS

Figure 23 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'.

Figure 24 contains the remaining, straightforward rules for the static semantics of standard expressions.

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

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

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

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

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

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

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

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

Fig. 23. Static semantics (rules for case expressions)

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

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

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

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

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

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

Fig. 24. Static semantics (standard rules)

B. ADDITIONAL RULES OF DYNAMIC SEMANTICS

Figure 25 presents the rules for evaluating case expressions and pattern matching. The rules 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. Figure 26 presents the language's standard beta-reduction rules.

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

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

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

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

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

$$\frac{}{\text{oks} \sim \text{oks} : \cdot} \qquad \frac{}{\text{halts} \sim \text{halts} : \cdot} \qquad \frac{}{\text{irrs} \sim \text{irrs} : \cdot} \qquad \frac{}{\text{exns} \sim \text{exns} : \cdot}$$

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

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

Fig. 25. Dynamic semantics (beta steps for case expressions)

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

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

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

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

Fig. 26. Dynamic semantics (standard beta steps)

C. PROOF OF TYPE SAFETY

We state below the lemmas and theorems in our proof of the Polymer language's type safety and provide the proof technique for each. Details for nontrivial proof cases appear in Ligatti's thesis [Ligatti 2006].

LEMMA C.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 $S; C, x : \tau' \vdash e : \tau$. □

LEMMA C.2 STORE SUBSTITUTION. *If* $C \vdash M : S$ *and* $S(l) = \tau$ *and* $S; C \vdash v : \tau$ *then* $C \vdash [l \to v]M : S$.

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

LEMMA C.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 $S; C \vdash e : \tau$. □

LEMMA C.4 INVERSION OF TYPING. *Every typing rule is invertible. That is, if the conclusion of any typing rule (in Figure 20, and Figures 23, and 24 in the Appendix) 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* $C(f) = \tau_1 \to \tau_2$ *and* $C \vdash p : (\tau_1; C')$.

PROOF. Immediate by inspection of the typing rules. □

LEMMA C.5 CANONICAL FORMS. *If* $S; C \vdash v : \tau$ *then*

—$\tau = \text{Bool}$ *implies* $v = \text{true}$ *or* $v = \text{false}$

—$\tau = (\tau_1, ..., \tau_n)$ *implies* $v = (v_1, ..., v_n)$

—$\tau = \tau'$ Ref *implies* $v = l$

—$\tau = \tau_1 \to \tau_2$ *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 19). □

Definition C.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 C.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 C.6 (well-typed context) and Lemma C.1 (variable substitution). □

Lemma C.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. □

Definition C.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 $x \in \mathrm{dom}(C')$ there exists v such that $v/x \in V$ and $S; C \vdash v : C'(x)$.

Lemma C.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 $v \sim p : V$. □

Lemma C.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 C.1 (variable substitution). □

Lemma C.12 Basic Decomposition. *If* $S; C \vdash e : \tau$ *and* $\vdash \vec{F} : C$ *and* $C \vdash M : S$ *and* $v_{\mathrm{pol}} = \mathrm{pol}(v_{\mathrm{query}}, v_{\mathrm{acc}}, v_{\mathrm{res}})$ *then either:*

—e *is a value* v, *or*

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

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

Lemma C.13 Policy Decomposition. *If* $\vdash (\vec{F}, M, e_{\mathrm{pol}}, e_{\mathrm{app}}) : \tau$ *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.*
  —$(\vec{F}, M, \mathrm{Triv}, e) \rightarrow_\beta (M', e')$, *for some* M' *and* e', *where* Triv *is the trivial policy defined in Figure 22*
  —$e = \mathrm{raise\ exn}$ *and* $E \neq E'[\mathrm{try}\ E''\ \mathrm{with}\ e'']$
  —$e = \mathrm{abort}$.

Proof. Immediate by Lemma C.4 (inversion of $\vdash (\vec{F}, M, e_{\mathrm{pol}}, e_{\mathrm{app}}) : \tau$) and Lemma C.12 (basic decomposition). □

Lemma C.14 Application Decomposition. *If* $\vdash (\vec{F}, M, v_{\mathrm{pol}}, e_{\mathrm{app}}) : \tau$ *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.*
  —$(\vec{F}, M, v_{\mathrm{pol}}, e) \rightarrow_\beta (M', e')$, *for some* M' *and* e'
  —$e = \mathrm{raise\ exn}$ *and* $E \neq E'[\mathrm{try}\ E''\ \mathrm{with}\ e'']$
  —$e = \mathrm{abort}$.

Proof. Immediate by Lemma C.4 (inversion of $\vdash (\vec{F}, M, v_{\mathrm{pol}}, e_{\mathrm{app}}) : \tau$), Lemma C.5 (canonical forms for $S; C \vdash v_{\mathrm{pol}} : \mathrm{Poly}$), and Lemma C.12 (basic decomposition). □

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

THEOREM C.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. □

Definition C.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 C.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 C.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 $e =$ raise exn (where $E \neq E'[\text{try } E'' \text{ with } e'']$) or $e =$ abort or $(\vec{F}, M, \text{Triv}, e) \rightarrow_\beta (M', e')$. When $e_{\text{pol}} = E[e]$ such that $e =$ raise exn or $e =$ 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 C.14 (application decomposition) implies that either $e_{\text{app}} = v_{\text{app}}$ or $e_{\text{app}} = E[e]$ such that $e =$ raise exn (where $E \neq E'[\text{try } E'' \text{ with } e'']$) or $e =$ 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. □

D. PROOF OF UNCIRCUMVENTABILITY

We state and prove the uncircumventability of Polymer policies after generalizing preservation from single- to multi-step transitions.

THEOREM D.1 MULTI-STEP PRESERVATION. *If* $\vdash P : \tau$ *and* $P \mapsto^* P'$ *then* $\vdash P' : \tau$.

PROOF. By induction on the derivation of $P \mapsto^* P'$, using Theorem C.16. □

THEOREM D.2 UNCIRCUMVENTABILITY. *If* $\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau$ *and* $(\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) \mapsto^* (\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)])$ *then there exist functions* v_{query}, v_{acc}, *and* v_{res} *and context* E' *such that:*

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

(2) $(\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)]) \mapsto (\vec{F}, M', v_{\text{pol}}, E'[v_{\text{query}}\ (\text{act}(f, v))])$, *and*

(3) *for all programs* P *such that* $(\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)]) \mapsto P$, *it must be the case that* $P = (\vec{F}, M', v_{\text{pol}}, E'[v_{\text{query}}\ (\text{act}(f, v))])$.

PROOF. By Theorem D.1, $\vdash (\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)]) : \tau$, so by the sole rule deriving $\vdash P : \tau$, we have $S; C \vdash v_{\text{pol}} : \text{Poly}$ for some label store S and variable context C. Because $S; C \vdash v_{\text{pol}} : \text{Poly}$, Canonical Forms (Lemma C.5) implies that $v_{\text{pol}} = \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}})$.

To prove obligation (2), let $P' = (\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)])$. Because $\vdash P' : \tau$, Theorem C.18 (progress) implies that either P' is finished or $P' \mapsto P''$ for some program P''. By Definition C.17, P' is not finished, so $P' \mapsto P''$. By inspection of the rules deriving $P' \mapsto P''$, we must have $(\vec{F}, M', v_{\text{pol}}, \text{invk act}(f, v)) \rightarrow_\beta (M'', e')$ for some M'' and e', but only the beta reduction for invk expressions enables this transition. By the definition of the beta reduction for invk expressions, then, we have $M'' = M'$ and $e' = \text{Wrap}(v_{\text{pol}}, F_i, v)$, where F_i is a monitored function. Let C denote the case expression in the definition of Wrap given in the beta-reduction rule for invk expressions. Using the definitions of Wrap, let-expression abbreviations, and the rule for top-level program execution based on application-level beta reductions, we obtain $(\vec{F}, M', v_{\text{pol}}, E[\text{invk act}(f, v)]) \mapsto (\vec{F}, M', v_{\text{pol}}, E'[v_{\text{query}}\ (\text{act}(f, v))])$, where $E' = E[\lambda s.C\ [\,]\ ]$. Because no other transitions are possible from P', proof obligation (3) is satisfied, that is, for all P such that $P' \mapsto P$, $P = (\vec{F}, M', v_{\text{pol}}, E'[v_{\text{query}}\ (\text{act}(f, v))])$. □