Composing Security Policies with Polymer

Lujo Bauer
Carnegie Mellon University
lbauer@ece.cmu.edu

Jay Ligatti
Princeton University
jligatti@cs.princeton.edu

David Walker
Princeton University
dpw@cs.princeton.edu

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

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

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

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

When decomposing the security problem into parts, it is tempting to scatter access-control checks, resource-monitoring code, and other mechanisms across the many modules that implement these components. This is especially true when the enforcement of some property involves several low-level components drawn from otherwise logically different parts of the system. For instance, in order to implement a policy concerning data privacy, it may be necessary to consider the operation of a wide variety of system components including the file system and the network, as well as printers and other forms of I/O. Unfortunately, a scattered implementation of a policy is much more difficult to understand and verify than a centralized implementation—even finding all the pieces of a distributed policy can be problematic. Moreover, the distribution of the security policy and mechanism through a large body of code can make it more difficult to update a policy in response to security breaches and vulnerabilities. In the current security climate, where new viruses can spread across the Internet in minutes, speedy reaction to vulnerabilities is critical.

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

We call the decision procedure mentioned above a query method. This method returns one of six suggestions indicating that: (1) the action is irrelevant to the policy; the action is OK; the action should be reconsidered after some other code is inserted; the return value of the action should be replaced by a precomputed value; a security exception should be thrown instead of executing the action; or, the application should be halted. These objects are referred to as suggestions because there is no guarantee that the policy's desired reaction will occur when it is composed with other policies. Also for this reason, the query method should not have effects. State updates occur in other policy methods, which are invoked only when a policy’s suggestion is followed.

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

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

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

1.1 Related Work

Safe language platforms, such as the Java Virtual Machine (JVM) [14] and Common Language Runtime (CLR) [19], use stack inspection as the basis of their security monitoring systems. Unfortunately, while stack inspection can be effective in many situations, it has some serious drawbacks as well. First, stack inspection is just one algorithm for implementing access control, and, as explained by several researchers [9, 18] this algorithm is inherently partial. More recent systems make decisions based on the entire history of a computation and all the code that has had an impact on the current system state, not just the current control stack [1, 6, 7, 8, 9, 11, 12]. A second important flaw in the stack inspection model is that operations to enable privileges and perform access-control checks are scattered throughout the system libraries. Consequently, in order to understand the policy that is being enforced, one must read through arbitrary amounts of library code.

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

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

1. We have designed a new programming methodology that permits policies to be composed in meaningful and productive ways. A key innovation is the separation of a policy into an effectless method that generates suggestions (OK, halt, raise exception, etc.) and is safe to execute at any time, and effectful methods that update security state only under certain conditions.

2. We have written a library of first-class, higher-order policies and used them to build a large-scale, practical security policy that enforces a sophisticated set of constraints on untrusted email clients.

3. We have developed a formal semantics for an idealized version of our language that includes all of the key features of our implementation including first-class policies, suggestions, and application events. A formal semantics helps nail down corner cases and provides an unambiguous specification of how security policies execute—a crucial feature of any security mechanism, but particularly important as our security policies have imperative effects. We prove our language is type safe, a necessary property for protecting the program monitor from untrusted applications.

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

Our monitor language can also be viewed as an aspect-oriented programming language (AOPL) in the style of AspectJ [10]. The main high-level difference between our work and previous AOPLs is that our “aspects” (the program monitors) are first-class values and that we provide mechanisms to allow programmers to explicitly control the composition of aspects. Several researchers [19, 20] describe functional, as opposed to object-oriented, AOPLs with first-class aspect-oriented advice. However, they do not support aspect combinators like the ones we have developed here. In general, composing aspects is a known problem for AOPLs, and we hope the ideas presented here will suggest a new design strategy for general-purpose AOPLs.

2. POLYMER SYSTEM OVERVIEW

Similar to the designs of Naccio and PoET/Pslang, the Polymer system is composed of two main tools. The first is a policy compiler that compiles program monitors defined
3. POLYMER LANGUAGE

In this section, we describe the core features of the Polymer language. We begin with the basic concepts and show how to program simple policies. Then, we demonstrate how to create more complete policies by composing simpler ones.

3.1 Core Concepts

Polymer is based on three central abstractions: actions, suggestions, and policies. Policies analyze actions and convey their decisions by means of suggestions.

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

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

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

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

Action patterns appear in two places. First, the action declaration file is a set of action patterns. During the instrumentation process, every action that matches an action pattern in the action declaration file is instrumented. Second, policies use action patterns in aswitch statements to determine which security-sensitive action they are dealing with. aswitch statements are similar to Java's switch statements, as the following example shows.

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

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

Suggestions. Whenever the untrusted application attempts to execute a security-relevant action, the monitor suggests a way to handle this action (which we often call a trigger action because it triggers the monitor into making such a suggestion).

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

- An IrrSug suggests that the trigger action execute unconditionally because the policy does not reason about it.
- An OKSug suggests that the trigger action execute even though the action is of interest to the policy.
- An InsSug suggests that making a final decision about the target action be deferred until after some auxiliary code is executed and its effects are evaluated.
- A ReplSug suggests replacing the trigger action, which computes some return value, with a return value supplied by the policy. The policy may use InsSugs to compute the suggested return value.

Figure 1: A secure Polymer application

in Polymer the language into plain Java and then into Java bytecode. The second tool is a bytecode rewriter that processes ordinary Java bytecode, inserting calls to the monitor in all the necessary places. In order to construct a secure executable using these tools, programmers must perform the following series of steps:

1. Write the action declaration file, which lists all program methods that might have an impact on system security.

2. Instrument the system libraries specified in the action declaration file. This step may be performed independently of the specification of the security policy. The libraries must be instrumented before the Java Virtual Machine (JVM) starts up since the default JVM security constraints prevent many libraries from being modified or reloaded once the JVM is running.

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

4. Start the JVM with the modified libraries.

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

6. Execute the secured application.
public abstract class Policy {
    public abstract Sug query(Action a);
    public void accept(Sug s) {};
    public void result(Sug a, Object result, boolean wasException) {};
}

Figure 2: The parent class of all policies

• An ExnSug suggests that the trigger action not be allowed to execute, but also that the target be allowed to continue running. Whenever following an ExnSug, Polymer notifies the target that its attempt at invoking the trigger action has been denied by throwing a SecurityException that the target can catch before continuing execution.

• A HaltSug suggests that the trigger action not be allowed to execute and that the target be halted.

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

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

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

• accept is called to indicate to a policy that its suggestion is about to be followed. This gives the policy a chance to perform any bookkeeping needed before the 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 just suggested an InsSug). Thus, InsSugs allow a policy to delay making a decision about a trigger action until after executing another action.

A policy interface consisting of query, accept, and result methods is fundamental to the design of Polymer. We can compose policies by writing policy combinators that query other policies and combine their suggestions. In combining suggestions, a combinator may choose not to follow the suggestions of some of the queried policies. Thus, query

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

Figure 3: Policy that allows all actions

public class DisSysCalls extends Policy {
    public Sug query(Action a) {
        switch(a) {
            case java.lang.Runtime.execute(...) : return new HaltSug(this, a);
        }
        return new IrrSug(this);
    }
    public void accept(Sug s) {
        System.out.println("Illegal method called: " + s.getTrigger());
    }
}

Figure 4: Policy that disallows Runtime.execute methods

methods must not assume that their suggestions will be followed and should be free of effects such as state updates and I/O operations.

3.2 Simple Policies

To give a feel for how to write Polymer policies, we define several simple examples in this section; in Sections 3.3 and 4.2 we will build more powerful policies by composing the basic policies presented here using a collection of policy combinators.

We begin by considering the most permissive policy possible: one that allows everything. The Polymer code for this policy is given in Figure 3. Because the query method of Trivial always returns an IrrSug, it allows all trigger actions to execute unconditionally. To enable convenient processing of suggestions, every Sug constructor has at least one argument, the Policy making the Sug.

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

In practice, there can be many methods that correspond to a single action that a policy considers security relevant. For example, a policy that logs incoming email may need to observe all actions that can open a message. It can be cumbersome and redundant to have to enumerate all these methods in a policy, so Polymer makes it possible to group them into abstract actions.

Abstract actions allow a policy to reason about security-relevant actions at a different level of granularity than is offered by the Java core API. They permit policies to focus on regulating particular behaviors, say, opening email, rather than forcing them to individually regulate each of the actions that cause this behavior. This makes it easier to write more concise, modular policies. Abstract actions also
public class GetMail extends AbsAction {
    public boolean matches(Action a) {
        switch (a) {
        case <Message IMAPFolder getMessage(int)>:
            case <void IMAPFolder fetch(Message[], *)>:
                return true;
            ...
            return false;
        }
    }

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

Figure 5: Abbreviated abstract action for receiving email messages; signature is Message[] GetMail()

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

Naccio [8] contains alternative notions, called platform interfaces, that support a similar sort of separation between
congrete and abstract actions. It appears that our design is slightly more general, as our abstract actions allow pro-
grammers 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 6 logs all incoming email and prepends the string “SPAM:” to subject lines on messages
flagged by a spam filter. To log incoming mail, the policy first tests whether the trigger action matches the GetMail
abstract action (from Figure 5) using the keyword abs in an action pattern. Since query methods should not have
effects, the policy returns an OKMsg for each GetMail action; the policy logs the fetched messages in the result method.
Polymer triggers a done action when the application terminates; the policy takes advantage of this feature to insert an
action that closes the message log. If the InsMsg to close the log 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 IrMsg. 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 ReplMsg.

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

3.3 Policy Combinators

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

Conjunctive combinator. It is often useful to restrict an application’s behavior by applying several policies at once
and enforcing the most restrictive one. For example, a policy that disallows access to files can be used in combination with

Figure 6: Abbreviated policy that logs all incoming email and prepends the string “SPAM:” to subject
lines on messages flagged by a spam filter.
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 insertion, 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 takes the least upper bound of the two suggestions as given by the following lattice, which orders suggestions in terms of increasing semantic impact.

```
Irrelevant ———> OK
Replace(v1) ———> Exception
Replace(v2)
Replace(v3) ———> Halt
...
```

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. ReplaceSugs have more impact than OKSugs since they change the semantics of the application. ReplaceSugs containing different replacements are considered inequivalent; consequently, the “conjunction” of two ReplaceSugs is considered to be an EqnSug.

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 did after. For example, in the initial state the action might have been OK, but after the intervening insertions the second conjunct might suggest that the application be halted.

An abbreviated version of the conjunctive combinator is shown in Figure 7. The calls to SugUtils.getNewSug in the query method simply create new suggestions with the same type as the first parameter in these calls. Notice that the suggestion returned by the combinator includes the suggestions on which the combinator based its decision. This design makes it possible for the combinator’s accept and result methods to notify the appropriate subpolicies that their suggestions have been accepted and followed.

**Figure 7: Conjunctive policy combinator**

---

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

makes the same suggestion. Otherwise, the combinator defers judgment to the second subpolicy. A similar sort of combinator is the Dominates combinator, which always follows the suggestion of the first conjunct if that conjunct considers the trigger action security-relevant; otherwise, it follows the suggestion of the second conjunct. Note that if two subpolicies never consider the same action security-relevant, composing them with a Dominates combinator is equivalent to composing them with a Conjunction combinator, except the Dominates combinator is in general more efficient because it need not always query both subpolicies.

**Selectors.** Selectors are combinators that choose to enforce exactly one of their subpolicies. A selector can be used, for example, to enforce a weaker policy on the target application if the target is cryptographically signed; otherwise, the selector enforces a stronger policy. We show a use of such a selector in Section 4.2.

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

Another example of a policy modifier is our AutoUpdate superpolicy. This policy checks a remote site once per day to determine if a new policy patch is available. If so, it makes a secure connection to the remote site, downloads the updated policy, and dynamically loads the policy into the JVM as its new subpolicy. Policies of this sort, which de-
4. EMPirical Evaluation

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

4.1 Implementation

The principal requirement for enforcing the run-time policies we are interested in is that the flow of control of a running program passes to a monitor immediately before and after executing a security-relevant method. The kind of pre- and post-invocation control-flow modifications to bytecode that we use to implement Polymer can be done by tools like AspectJ [10]. Accordingly, we considered using AspectJ to insert into bytecode hooks that would trigger our monitor as needed. However, we wanted to retain precise control over how and where rewriting occurs to be able to make decisions in the best interests of security, which is not the primary focus of aspect-oriented languages like AspectJ. Instead, we used the Apache BCEL API [3] to develop our own bytecode rewriting tool.

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

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

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

Our system impacts target applications in two phases: before and during loading, when the application and the class libraries are instrumented by the bytecode rewriter; and during execution. The total time to instrument every method in all of the standard Java library packages (i.e., the 28742 methods in the 3948 classes in the java and javax packages of Sun’s Java API v.1.4.0) was 107 s, or 3.7 ms per instrumented method. This cost is reasonable because library instrumentation only needs to be performed once (rather than every time a target application is executed). The average time to load non-library classes into the JVM with our specialized class loader, but without instrumenting any methods, was 12 ms, twice as long as the VM’s default class loader required. The cost of transferring control to and from a Polymer program while executing a target is very low (approximately 0.62 ms); the run-time overhead is dominated by the computations actually performed by the policy. Hence the cost of monitoring a program with Polymer is almost entirely dependent on the complexity of the security policy.

4.2 Case Study: Securing Email Clients

To test the usefulness of Polymer in practice, we have written a large-scale policy to secure untrusted email clients that use the JavaMail API. The entire policy, presented in Figure 8, is approximately 1800 lines of Polymer code. We have extensively tested the protections enforced by the policy on an email client called Poika [16], without having to inspect or modify any of the approximately 30K lines of Poika source code. The run-time cost of enforcing the complex constraints specified by our policy is difficult to measure because the performance of the email client depends largely on interactions with the user; however, our experience indicates that the overhead is rarely noticeable.

The component policies in Figure 8 each enforce a modular set of constraints. The Trivial and Attachments policies were described in Section 3.2; the Conjunction, TryWith, Dominates, Audit, and AutoUpdate superpolicies were described in Section 3.3. The left branch of the policy hierarchy (shaded in Figure 8) contains a generic policy that we include in all of our high-level Polymer policies. This branch of policies ensures that a target cannot use class loading, reflection, or system calls maliciously and alerts the user when the memory available to the virtual machine is nearly exhausted. The nonshaded branch of the policy hierarchy contains 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.
- AllowOnlyIMAP allows only standard email socket connections (POP and IMAP).
- QueryCalls allows effectless security-sensitive actions invoked in the query method of its subpolicy to execute unconditionally. The implementation of this policy modifier inspects the dynamic call stack to deter-

1The tests were performed on a Dell PowerEdge 2650 with dual Intel Xeon 2.3 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.
mine whether a trigger action was invoked in the sub-policy’s `query` method.

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

- **IncomingMail** was shown in an abbreviated form in Figure 6. In addition to logging incoming mail and prompting “SPAM?” to the subject lines of email that fails a spam filter, this policy truncates long subject lines and displays a warning when a message containing an attachment is opened.

5. FORMAL SEMANTICS

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

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

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

Monitored functions \( \text{fun } f(x:	au_1); \tau_2 \{ e \} \) are syntactically separated from ordinary functions \( \lambda x:	au . e \). Moreover, we treat monitored function names \( f \) as a syntactically separate class of variables from ordinary variables \( x \). Monitored function names may only appear wrapped up as actions as in \( \text{act}(f, e) \). These actions are suspended computations that must be explicitly invoked with the command invk \( e \). Invoking an action causes the function in question to be executed and its result wrapped in a result constructor result(\( e: \tau \)). The elimination forms for results and most other objects discussed above is handled through a generic case expression and pattern matching facility. The class of patterns \( p \) includes variable patterns \( x \) as well as patterns for matching constructors. Ordinary, unmonitored functions are executed via the usual function application command \( (e_1 . e_2) \).

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

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

\[ \text{As usual, we treat expressions that differ only in the names of their bound variables as identical. We often write let } x = e_1 \text{ in } e_2 \text{ for } (\lambda x : \tau . e_2) e_1. \]
with type $\tau$ and binds variables with types given by $C'$.

We have worked hard to make the static semantics a simple but faithful model of the implementation. In particular, notice that all actions share the same type $\langle \text{Act} \rangle$ 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 $\text{Res}$. Case analysis is used to safely extract the proper value. This choice allows policy objects to process and react to arbitrary actions. To determine the precise nature of any action and give it a more refined type, the policy will use pattern matching. We have a similar design for action results and replacement values.

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

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

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

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

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

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

**Observations.** The semantics gives insight into some of the subtler elements of our implementation, which are important both to system users and to us as implementers.
\[
(F, M, e_{\text{pol}}, e_{\text{app}}) \mapsto (F, M', e'_{\text{pol}}, e'_{\text{app}})
\]

\[
(F, M, \text{Triv}, e) \rightarrow_\beta (M', e')
\]

\[
(F, M, E[e], e_{\text{app}}) \mapsto (F, M', E[e'], e_{\text{app}})
\]

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

\[
(F, M, v_{\text{pol}}, e) \mapsto (M', e')
\]

\[
(F, M, v_{\text{pol}}, E[e]) \mapsto (F, M, v_{\text{pol}}, E[e'])
\]

\[
(F, M, v_{\text{pol}}, e_{\text{app}}) \mapsto (M', e'_{\text{app}})
\]

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

\[
F_i \in \bar{F}, \quad F_i = \text{fun}f(x : \tau_1 : \tau_2 e)
\]

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

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

\[
\begin{align*}
\text{irs} & \Rightarrow \text{let } x = v \in \text{result}(e; \tau_2) \\
\text{oks} & \Rightarrow v_{\text{acc}}(\text{act}(f,v), s); \\
\text{reps}(r) & \Rightarrow \text{let } x = v \in \text{result}(e; \tau_2) \in v_{\text{res}} r; \quad r \\
\text{exns} & \Rightarrow v_{\text{acc}}(\text{act}(f,v), s); \quad \text{raise exn} \\
\text{ins}(a) & \Rightarrow v_{\text{acc}}(\text{act}(f,v), s); \quad v_{\text{res}}(\text{invk} a); \quad \text{invk act}(f,v) \\
\ast & \Rightarrow \text{abort}
\end{align*}
\]

\[\text{Figure 11: Dynamic semantics (selected rules)}\]

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

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

\[\text{6. SUMMARY}\]

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

\[\text{7. REFERENCES}\]


