Types and Effects for Non-interfering Program Monitors*

Lujo Bauer, Jarred Ligatti and David Walker

Department of Computer Science
Princeton University
Princeton, NJ 08544

Abstract. A run-time monitor is a program that runs in parallel with an untrusted application and examines actions from the application's instruction stream. If the sequence of program actions deviates from a specified security policy, the monitor transforms the sequence or terminates the program. We present the design and formal specification of a language for defining the policies enforced by program monitors.
Our language provides a number of facilities for composing complex policies from simpler ones. We allow policies to be parameterized by values or other policies, and we define operators for forming the conjunction and disjunction of policies. Since the computations that implement these policies modify program behavior, naive composition of computations does not necessarily produce the conjunction (or disjunction) of the policies that the computations implement separately. We use a type and effect system to ensure that computations do not interfere with one another when they are composed.

1 Introduction

Any system designed to execute and interoperate with potentially malicious code should implement at least two different sorts of security mechanisms:

1. A safe language and sound type checker to statically rule out simple bugs.
2. A run-time environment that will detect, document, prevent and recover from those errors that cannot be detected beforehand.

Strong type systems such as the ones in the Java Virtual Machine [16] and Common Language Runtime [10, 11, 17] are the most efficient and most widely deployed mechanisms for ruling out a wide variety of potential security holes ranging from buffer overruns to misuse of system interfaces.

To complement static checking, secure run-time environments normally use auxiliary mechanisms to check properties that cannot be decided at compile time or link time. One of the ways to implement such run-time checks is with program monitors, which examine a sequence of program actions before they are

* This research was supported in part by a gift from Microsoft Research, Redmond; DARPA award F30602-99-1-0519; and NSF Trusted Computing grant CCR-0208601.

executed. If the sequence deviates from a specified policy, the program monitor transforms the sequence or terminates the program.

In this paper, we describe a new general-purpose language called Polymer that can help designers of secure systems detect, prevent and recover from errors in untrusted code at runtime. System architects can use Polymer to write program monitors that run in parallel with an untrusted application. Whenever the untrusted application is about to call a security-sensitive method, control jumps to the Polymer program which determines which of the following will occur:

- the application runs the method and continues with its computation,
- the application is terminated by the monitor,
- the application is not allowed to invoke the given method, but otherwise may continue with its computation, or
- the monitor performs some computation on behalf of the application before or after proceeding with any of the first three options (Figure 1).

This basic architecture has been used in the past to implement secure systems [6, 8, 9, 12, 19]. Previous work has shown that the framework effectively subsumes a variety of less general mechanisms such as access-control lists and stack inspection. Unfortunately, there has been a nearly universal lack of concern for precise semantics for these languages, which we seek to remedy in this paper.

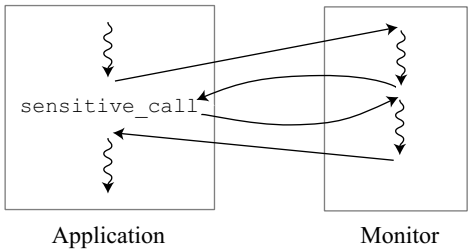

Fig. 1. Sample interaction between application and monitor: monitor allows application to make a sensitive call.

We improve upon previous work in a number of ways.

- We present a new, general-purpose language for designing run-time security policies. Policies are able to prevent actions from being executed, execute their own actions, and terminate the offending program. In our language, policies are first-class objects and can be parameterized by other policies or ordinary values. We provide several interesting combinators that allow complex policies to be built from simpler ones.

– We have defined a formal operational semantics for our language, which turns out to be a variant of the computational lambda calculus [18]. To our knowledge, this is the first such semantics for a general-purpose security monitoring language. It provides a tool that system architects can use to reason precisely about their security policies.
– We provide a type system, which we have proved sound with respect to our operational semantics. The type system includes a novel effect system that ensures that composed policies do not interfere with one another.

2 A Calculus for Composing Security Policies

In this section, we provide an informal introduction to our security policy language.

2.1 Simple Policies

Our monitoring language is derived from Moggi's computational lambda calculus [18]; consequently, the language constructs are divided into two groups: pure terms M and computations E. A computation runs in parallel with a *target program* and may have effects on the target's behavior. We call a suspended computation paired with an action set ($\{\texttt{actions} : A; \texttt{policy} : E\}$) a *policy*. A policy is a term that, when its suspended computation E is run, will intercept and manipulate *target actions* in the set A. We call this set of actions the *regulated set*. For the purposes of this paper, a target action is a function or method call that the target application wishes to execute. However, it is easy to imagine a variety of other sorts of target program actions, such as primitive operations like assignment, dereference, iteration, the act of raising particular exceptions, etc., that might also be considered actions that are regulated by a security policy. It should also be noted that the level of abstraction of program actions depends on the language and system interface of the target application (e.g., C system calls versus Java Core API method calls); these concerns, however, are orthogonal to the design of our calculus.

A First Example Consider the following policy, which enforces a limit on the amount of memory that an application can allocate for itself.

```
fun mpol(q:int).
{
  actions:  malloc();
  policy:
    next →
      case ⋆ of
        malloc(n) →
          if ((q-n) > 0) then
            ok; run (mpol (q-n))
          else
            halt
      end
    done → ()
}
```

A recursive policy, like the one above, is a recursive function (a term) with a policy as its body. The recursive function argument `q` is a memory quota that the application must not exceed. The only action manipulated by this policy is the `malloc` action. The computation defining the policy begins with the (`next` $\rightarrow$ E_1 | `done` $\rightarrow$ E_2) computation, which suspends the monitor until the target is about to execute the next action in the regulated set (i.e., the next malloc operation). At this point, the monitor executes E_1. If the program terminates before before executing another regulated action, E_2 will be executed to perform any sort of bookkeeping or application cleanup that is necessary. In this example, we assume no bookkeeping is necessary so the monitor simply returns `()` to indicate it is done.

The `ok` statement signals that the current action should be accepted, and `halt` is the terminal computation, which halts the target program.

A recursive call to a policy involves two steps. The first step is a function application (`mpol (q-n)`), which returns a policy (a suspended computation). To *run* the suspended computation, we use the `run` statement (`run (mpol (q-n))`). Sometimes, computations return interesting values (not just unit) in which case we write `let {x} = pol in E`. This is the monadic let, which executes its primary argument `pol`, binds the resulting value to `x` and continues the computation with `E`. We also use an ordinary let where convenient: `let x = M in E`.

Now that we have defined our recursive memory-limit policy, we can initialize it with a quota (`q0`) simply by applying our recursive function.

```
memLimit = mpol q0
```

The type of any policy is $\mathcal{M}(\tau)$ where τ is the type of the value that the underlying computation returns. Hence, the type of `memLimit` is $\mathcal{M}(\mathsf{unit})$.

A Second Example In this example, we restrict access to files by controlling the actions `fopen` and `fclose`. For simplicity, we assume that `fclose` takes a string argument rather than a file descriptor. The first argument to the policy is a function (`acl`) that returns true if the target is allowed access to the given file

in the given mode. The second argument is a list of files that the application has opened so far. The code below uses a number of list processing functions including cons (::), membership test (`member`), and element delete (`delete`).

```
fun fpol(acl:string->mode->bool, files:file list).
{
  actions:  fopen(), fclose();
  policy:
    let fcloses fs = {... fclose f ...} in
    next →
      case ⋆ of
        fopen(s,m) →
          if (acl s m) then
            ok; run (fpol acl (s::files))
          else
            run (fcloses files); halt
        fclose(s) →
          if (member files s) then
            ok; run (fpol acl (delete files s))
          else
            sup; run (fpol acl files)
      end
    done →
      run (fcloses files)
}
```

The main additional statement of interest in this policy is the `sup` statement. We view an attempt to close a file that that has not been opened by the application a benign error. In this case, we do not terminate the application, we simply *suppress* the action and allow the application to continue (if it is able to do so). In practice, the `sup` expression also throws a security exception that may be caught by the target.

A second point of interest is the fact that our file-system policy is written so that if the target terminates, it will close any files the target has left open. It uses an auxiliary computation `fcloses` to close all the files in the list.

Once again, we must initialize our policy with appropriate arguments.

```
fileAccess = fpol (acl0,[]).
```

2.2 Composing Policies

One of the main novelties of our language is that policies are first-class values. As a result, functions can abstract over policies and policies may be nested inside other policies. Moreover, we provide a variety of combinators that allow programmers to synthesize complex policies from simpler ones.

Parallel Conjunctive Policies A resource-management policy might want to enforce policies on a variety of different sorts of resources, all defined independently of one another. We use the conjunctive combinator $M_1 \wedge M_2$ to create such a policy. For example, the following policy controls both file access and limits memory consumption.

```
RM = fileAccess ∧ memLimit
```

When this policy is run, target actions are streamed to both `fileAccess` and `memLimit`. Actions such as `malloc`, which are not relevant to the `fileAccess` policy, are ignored by it and automatically deemed OK. The same is true of actions that are not relevant to `memLimit`. The two computations may be seen as running in parallel and if either policy decides to halt the target then the target will be stopped.

The result of a parallel conjunctive policy is a pair of values, one value being returned from each of the two computations. Hence, our resource manager has type $\mathcal{M}(\mathsf{unit} \times \mathsf{unit})$.

Closely related to the parallel conjunctive policy is the trivial policy $\top$, which immediately returns (). The trivial policy is the identity for the parallel conjunctive policy. In other words, $M \wedge \top$ accepts exactly the same sequences of program actions as M.

Higher-order Policies Since policies are ordinary values, we can parameterize policies by other policies. For example, rather than fix a particular resource management policy once and for all, a system designer might prefer to design a generic resource manager that is composed of a file-access policy and a memory limit policy.

```
genericRM = λfa:M(unit).λml:M(unit).{let {x} = fa ∧ ml in ()}
```

The generic resource manager above abstracts two policies and returns another policy that runs the two policies in conjunction, discards their results and returns unit. We can apply the generic resource manager to the two policies we created above.

```
strictRM = genericRM fileAccess memLimit
```

However, we might need a different policy for a different application. For instance, for a more trusted application, we might choose not to limit memory, but still control file access. In this case, we use the trivial policy instead of `memLimit`.

```
laxRM = genericRM fileAccess ⊤
```

Parallel Disjunctive Policies A parallel disjunctive policy $M_1 \vee_\tau M_2$ accepts a sequence of operations and returns a result as soon as either M_1 or M_2 would accept the sequence of operations and return. Both policies must agree to halt the target in order to stop it. As in the conjunctive policy, target actions that are not in the regulated set of one of the policies are simply passed over by that policy and implicitly accepted. A disjunctive policy $M_1 \vee_\tau M_2$ has type $\mathcal{M}(\tau)$ when $\tau = \tau_1 + \tau_2$, M_1 has type $\mathcal{M}(\tau_1)$ and M_2 has type $\mathcal{M}(\tau_2)$.

There are several uses for disjunctive policies. At the most basic level, a disjunctive policy can serve to widen an existing policy. For example, suppose we have already implemented a policy for controlling arbitrary, untrusted applications (`untrustedPol`). Later, we might wish to develop a second policy for more trusted applications that authenticate themselves first (`authenticatedPol`). By using disjunction we allow applications either to authenticate themselves and gain further privileges or to use the untrusted policy.

```
widenedPol = untrustedPol ∨τ authenticatedPol
```

It is likely possible to rewrite `untrustedPol` so that it grants extra privileges when a user authenticates himself. However, modular design principles suggest we should leave the code of the initial policy alone and create a separate module (policy) to handle the details of authentication and the extended privileges.

Disjunctive policies also provide a convenient way to create *Chinese wall policies* [5]. A Chinese wall policy allows the target to choose from one of many possible policies. However, when one policy is chosen the others become unavailable. For example, when designing a browser policy, we might expect two different sorts of applets. One sort of applet acts like a proxy for a database or service situated across the net. This kind of applet needs few host system resources other than network access. It takes requests from a user and communicates back to the online database. In particular, it has no use for the file system. Another sort of applet performs tasks for the host system and requires access to host data. In order to allow both sorts of applets to run on the host and yet to protect the privacy of host data, we can create a Chinese wall policy which allows either file-system access or network access but not both.

In the code below, we implement this policy. The patterns `File.*` and `Network.*` match all functions in the interface `File` and `Network` respectively. We assume the policies `filePolicy` and `networkPolicy` have been defined earlier.

```
fileNotNetwork =
{
  actions:  File.*, Network.*;
  policy:
    next →
      case ⋆ of
        File.* → run (filePolicy)
        Network.* → halt
      end
    done → ()
}


networkNotFile =
{
  actions:  File.*, Network.*;
  policy:
    next →
      case ⋆ of
        File.* → halt
        Network.* → run (networkPolicy)
      end
    done → ()
}
ChineseWall = fileNotNetwork ∨τ networkNotFile
```

Like conjunction, disjunction has an identity: $\bot$ is the unsatisfiable policy, which halts immediately regardless of any program actions. The policy $M \vee_\tau \bot$ accepts the same sequences of actions as M.

2.3 Interfering Policies

Composition of policies can sometimes lead to policies that are ill-defined or simply wrong. For example, consider the conjunction of two file-system policies, `liberalFilePolicy` and `stricterFilePolicy`. The first policy okays each file-system action while the second policy suppresses some of the file-system actions. What should the result be when one policy suppresses an action and another concurrently allows (and potentially requires) it to occur?

A similar problem would occur if we attempted to compose our original file-system policy `fileAccess` with a logging policy `logPolicy` that stores the sequence of all actions that occur in a system in order to detect suspicious access patterns and to uncover mistakes in a policy. Our original `fileAccess` itself performs certain actions on behalf of the target, including closing target files. If the logging policy operates concurrently with the file-access policy, it cannot detect and log the actions performed by `fileAccess`.

We propose a twofold solution to such problems. First, we use a type and effect system to forbid ill-defined or interfering policies such as the ones considered above. Second, we provide an alternative set of combinators that allow programmers to explicitly sequence policies rather than having them execute in parallel. This gives programmers necessary flexibility in defining policies.

Types and Effects Our type and effect system gives policies refined types with the form $\mathcal{M}^{A_r}_{A_e}(\tau)$. The set of actions A_r includes all the actions regulated by the policy. The second set A_e specifies the effect of the policy. In other words, it specifies the actions that may be suppressed or initiated on behalf of the program.

These refined types give rise to a new typing rule for parallel conjunctive policies. In the following rule, the context Γ maps variables to their types in the usual way.

$$\frac{\Gamma \vdash M_1 : \mathcal{M}^{A_1}_{A_2}(\tau_1) \qquad \Gamma \vdash M_2 : \mathcal{M}^{A_3}_{A_4}(\tau_2) \qquad A_1 \cap A_4 = A_2 \cap A_3 = \emptyset}{\Gamma \vdash M_1 \wedge M_2 : \mathcal{M}^{A_1 \cup A_3}_{A_2 \cup A_4}(\tau_1 \times \tau_2)}$$

The constraint in the rule specifies that the effects of one of the policies must not overlap with the set of actions regulated by the other policy. A similar rule constrains the policies that may be composed using parallel disjunction.

$$\frac{\Gamma \vdash M_1 : \mathcal{M}^{A_1}_{A_2}(\tau_1) \qquad \Gamma \vdash M_2 : \mathcal{M}^{A_3}_{A_4}(\tau_2) \qquad A_1 \cap A_4 = A_2 \cap A_3 = \emptyset}{\Gamma \vdash M_1 \vee_{\tau_1+\tau_2} M_2 : \mathcal{M}^{A_1 \cup A_2}_{A_3 \cup A_4}(\tau_1 + \tau_2)}$$

Rules for typing other terms and rules for typing computations are explained in Section 3.

Sequential Combinators Sequential combinators allow programmers to explicitly order the execution of effectful policies that apply to the same set of target actions. The sequential conjunctive policy $M_1 \triangle M_2$ operates as follows. The policy M_1 operates on the target action stream, creating an output stream that may contain new actions that M_1 has injected into the stream and may be missing actions that M_1 has suppressed. The policy M_2 acts as it normally would on the output of M_1. Since this is a conjunctive policy, if either policy decides to terminate the application then the application will be terminated. The sequential disjunctive policy $M_1 \bigtriangledown_\tau M_2$ is similar: M_2 operates on the output of M_1. In this case, however, both M_1 and M_2 must decide to terminate the target in order for the target to be stopped. If one policy signals halt, the disjunction continues to operate as if that policy has no effect on the target.

The typing rules for sequential combinators (shown below) are much more liberal then the typing rules for parallel combinators. By explicit sequencing of operations, the programmer determines how the conflicting decisions should be resolved.

$$\frac{\Gamma \vdash M_1 : \mathcal{M}^{A_1}_{A_2}(\tau_1) \quad \Gamma \vdash M_2 : \mathcal{M}^{A_3}_{A_4}(\tau_2)}{\Gamma \vdash M_1 \triangle M_2 : \mathcal{M}^{A_1 \cup A_3}_{A_2 \cup A_4}(\tau_1 \times \tau_2)}$$

$$\frac{\Gamma \vdash M_1 : \mathcal{M}^{A_1}_{A_2}(\tau_1) \quad \Gamma \vdash M_2 : \mathcal{M}^{A_3}_{A_4}(\tau_2)}{\Gamma \vdash M_1 \bigtriangledown_{\tau_1+\tau_2} M_2 : \mathcal{M}^{A_1 \cup A_3}_{A_2 \cup A_4}(\tau_1 + \tau_2)}$$

Because sequential operators accept a wider range of policies than parallel ones, they can be used to implement any policy that can be implemented with parallel combinators. Parallel combinators, however, ensure the often-desirable property that the two policies being composed do not interfere with each other.

3 Formal Semantics

This section describes the syntax and formal semantics of our calculus.

3.1 Syntax

The syntax of our formal language differs slightly from the syntax used in the previous section. First, we use the metavariable a to range over actions and consider them to be atomic symbols rather than decomposable into class name, method name and arguments. Second, we write the regulated set for a policy using superscript notation: $\{E\}^A$ is the simple policy with the regulated set A and computation E.

There are also a number of differences in the computations. Our `acase` instruction chooses a control flow path based upon whether the current action belongs to an arbitrary subset A of the current possible actions. If we want to store or manipulate the current action, we use the primitive $x \ \rightarrow E$ to bind the current action to the variable x, which may be used in E (intuitively, this takes the place of pattern matching). To invoke one of the atomic program actions, we explicitly write $\texttt{ins}(a)$. Finally, for each of the policy combinators discussed in the previous section, we add a corresponding computation. Each of these computations is superscripted with the regulated sets for their subcomputations. Figure 2 presents a formal syntax for our language.

3.2 Static Semantics

We specify the static semantics for the language using three main judgments.

Subtyping: $\vdash \tau_1 \leq \tau_2$ The rules for subtyping are mostly standard. Unit, pairs, sums and function types have their usual subtyping rules. We say the type of actions $\mathsf{act}(A)$ is covariant in A since $\mathsf{act}(A)$ is a subtype of $\mathsf{act}(A')$ when $A \subseteq A'$. Policy types are covariant in their return type and effect set but invariant in their regulated set. In other words, it is safe for policies to appear to have a larger

(Types)	τ	$::=$	$\mathsf{act}(A) \mid \tau_1 \to \tau_2 \mid \mathsf{unit}$
		$\mid$	$\tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mathcal{M}_{A_2}^{A_1}(\tau)$
(Behaviors)	β	$::=$	$\cdot \mid \mathtt{ins}(a) \mid \mathtt{sup}(a) \mid \mathtt{acc}(a)$
(Terms)	M	$::=$	x (variable)
		$\mid$	a (action)
		$\mid$	$\mathtt{fun}\ f{:}\tau\ (x).M$ (recursive function)
		$\mid$	$M_1\ M_2$ (application)
		$\mid$	$()$ (unit)
		$\mid$	$\langle M_1, M_2 \rangle$ (pairing)
		$\mid$	$\pi_1\ M \mid \pi_2\ M$ (first/second projections)
		$\mid$	$\mathtt{inl}_\tau(M_1) \mid \mathtt{inr}_\tau(M_2)$ (left/right injections)
		$\mid$	$\mathtt{case}\ M_1\ (x \to M_2 \mid x \to M_3)$ (case)
		$\mid$	$\{E\}^A$ (simple policy)
		$\mid$	$\top$ (trivially satisfiable policy)
		$\mid$	$M_1 \wedge M_2$ (parallel-conjunctive policy)
		$\mid$	$M_1 \triangle M_2$ (sequential-conjunctive policy)
		$\mid$	$\bot$ (unsatisfiable policy)
		$\mid$	$M_1 \vee_\tau M_2$ (parallel-disjunctive policy)
		$\mid$	$M_1 \triangledown_\tau M_2$ (sequential-disjunctive policy)
(Values)	v	$::=$	$x \mid a \mid \mathtt{fun}\ f{:}\tau\ (x).M \mid ()$
		$\mid$	$\langle v_1, v_2 \rangle \mid \mathtt{inl}_\tau(v_1)$
		$\mid$	$\mathtt{inr}_\tau(v_2) \mid \{E\}^A$
(Computations)	E	$::=$	M (return)
		$\mid$	$\mathtt{let}\ \{x\} = M\ \mathtt{in}\ E$ (let)
		$\mid$	$\mathtt{ok}; E$ (accept action)
		$\mid$	$\mathtt{sup}; E$ (suppress action)
		$\mid$	$\mathtt{ins}(M); E$ (call action)
		$\mid$	$(\mathtt{next} \to E_1 \mid \mathtt{done} \to E_2)$ (next action)
		$\mid$	$x\ \to E$ (bind action)
		$\mid$	$\mathtt{acase}\ (\star\ \subseteq A)\ (E_1 \mid E_2)$ (action case)
		$\mid$	$\mathtt{case}\ M\ (x \to E_1 \mid x \to E_2)$ (case)
		$\mid$	$\mathtt{any}$ (trivial computation)
		$\mid$	$E_1 \wedge^{A_1, A_2} E_2$ (parallel-conjunctive computation)
		$\mid$	$E_1 \triangle^{A_1, A_2} E_2$ (sequential-conjunctive computation)
		$\mid$	$\mathtt{halt}$ (terminal computation)
		$\mid$	$E_1 \vee_\tau^{A_1, A_2} E_2$ (parallel-disjunctive computation)
		$\mid$	$E_1 \triangledown_\tau^{A_1, A_2} E_2$ (sequential-disjunctive computation)

Fig. 2. Syntax

effect than they actually do, but they must regulate the set that they claim to regulate.

$$\frac{A \subseteq A'}{\vdash \mathsf{act}(A) \leq \mathsf{act}(A')} \qquad \text{(Sub-Act)}$$

$$\frac{A_2 \subseteq A_2' \quad \vdash \tau \leq \tau'}{\vdash \mathcal{M}_{A_2}^{A_r}(\tau) \leq \mathcal{M}_{A_2'}^{A_r}(\tau')} \qquad \text{(Sub-Monad)}$$

Term Typing: $\Gamma \vdash M : \tau$ The term typing rules contain the ordinary introduction and elimination rules for functions, unit, pairs and sums. The treatment of variables is also standard. The basic rule for actions gives an action a the singleton type $\mathsf{act}(\{a\})$. When this rule is used in conjunction with the subsumption rule, an action may be given any type $\mathsf{act}(A)$ such that $a \in A$. The non-standard typing rules for terms are given below.

$$\frac{}{\Gamma \vdash a : \mathsf{act}(\{a\})} \qquad \text{(S-Act)}$$

$$\frac{\Gamma; \diamond \vdash^{A_r} E : \tau, A_e}{\Gamma \vdash \{E\}^{A_r} : \mathcal{M}_{A_e}^{A_r}(\tau)} \qquad \text{(S-Sus)}$$

$$\frac{}{\Gamma \vdash \top : \mathcal{M}_{\emptyset}^{\emptyset}(\mathsf{unit})} \qquad \text{(S-Top)}$$

$$\frac{\Gamma \vdash M_1 : \mathcal{M}_{A_2}^{A_1}(\tau_1) \quad \Gamma \vdash M_2 : \mathcal{M}_{A_4}^{A_3}(\tau_2) \quad A_1 \cap A_4 = A_2 \cap A_3 = \emptyset}{\Gamma \vdash M_1 \wedge M_2 : \mathcal{M}_{A_2 \cup A_4}^{A_1 \cup A_3}(\tau_1 \times \tau_2)} \qquad \text{(S-ParCon)}$$

$$\frac{\Gamma \vdash M_1 : \mathcal{M}_{A_2}^{A_1}(\tau_1) \quad \Gamma \vdash M_2 : \mathcal{M}_{A_4}^{A_3}(\tau_2)}{\Gamma \vdash M_1 \triangle M_2 : \mathcal{M}_{A_2 \cup A_4}^{A_1 \cup A_3}(\tau_1 \times \tau_2)} \qquad \text{(S-SeqCon)}$$

$$\frac{}{\Gamma \vdash \bot : \mathcal{M}_{\emptyset}^{\emptyset}(\tau)} \qquad \text{(S-Bot)}$$

$$\frac{\Gamma \vdash M_1 : \mathcal{M}_{A_2}^{A_1}(\tau_1) \quad \Gamma \vdash M_2 : \mathcal{M}_{A_4}^{A_3}(\tau_2) \quad A_1 \cap A_4 = A_2 \cap A_3 = \emptyset}{\Gamma \vdash M_1 \vee_{\tau_1+\tau_2} M_2 : \mathcal{M}_{A_3 \cup A_4}^{A_1 \cup A_2}(\tau_1 + \tau_2)} \qquad \text{(S-ParDis)}$$

$$\frac{\Gamma \vdash M_1 : \mathcal{M}_{A_2}^{A_1}(\tau_1) \quad \Gamma \vdash M_2 : \mathcal{M}_{A_4}^{A_3}(\tau_2)}{\Gamma \vdash M_1 \bigtriangledown_{\tau_1+\tau_2} M_2 : \mathcal{M}_{A_2 \cup A_4}^{A_1 \cup A_3}(\tau_1 + \tau_2)} \qquad \text{(S-SeqDis)}$$

Elementary policies (rule (S-Sus)) are given the type $\mathcal{M}_{A_e}^{A_r}(\tau)$ when the suspended computation regulates the actions in A_r, has effect A_e and produces a value of type τ. The trivial policy (rule (S-Top)) makes its decisions based upon no regulated actions, has no effect and simply returns unit. The terminal policy

(rule (S-Bot)) also makes its decision based upon no regulated actions, has no effect, but instead of returning a value, it immediately calls for termination of the target. Since the terminal policy never returns, we allow its return type to be any type τ.

Rules (S-ParCon) and (S-SeqCon) give types to the two conjunctive policies. In each case, the type of the resulting computation involves taking the union of the regulated sets and the union of the effects since a conjunctive policy makes its decisions based on the regulated actions of both policies and potentially has the effects of either policy. These combinators return a pair of values, which is reflected in the type of the conjunctive combinator. The parallel conjunction is constrained so that the regulated set of one conjunct is disjoint from the effect of the other and vice versa. This constraint prevents one conjunct from inserting or suppressing actions that should be regulated by the other conjunct. Typing for the sequential conjunction is more liberal. It allows one policy to supersede another regardless of the effects of either policy. The rules for the disjunctive combinators ((S-ParDis) and (S-SeqDis)) are analogous to their conjunctive counterparts except that disjunctions return sums rather than pairs.

Computation Typing: $\Gamma; B \vdash^{A_r} E : \tau, A_e$ The basic judgment for typing computations may be read "Computation E produces a value with type τ and has effect A_e in Γ when run against a target whose next action is in B." B ranges over non-empty sets A or the symbol $\diamond$, which represents no knowledge about the next action. The next action might not even exist, as is the case when the target has terminated. We maintain this set of possible next actions so that we know what actions to consider as possible effects of a suppress statement and what actions may be bound to a variable in a `bind` statement. We do not consider computation judgments to be valid unless either $B \subseteq A_r$ or $B = \diamond$. We define $B \sqcup A_r$ to be B if $B \subseteq A_r$ and $\diamond$ otherwise. Finally, set intersect and set minus operators $\cap_\diamond$ and $\setminus_\diamond$ act like standard set operators, except that instead of returning $\emptyset$ they return $\diamond$.

The computation typing rules are given below.

$$\frac{\Gamma \vdash M : \tau}{\Gamma; B \vdash^{A_r} M : \tau, \emptyset} \quad \text{(SE-Ret)}$$

$$\frac{\Gamma \vdash M : \mathcal{M}_{A_2}^{A'_r}(\tau') \quad \Gamma, x{:}\tau'; \diamond \vdash^{A_r} E : \tau, A \quad A'_r \subseteq A_r}{\Gamma; B \vdash^{A_r} \texttt{let}\ \{x\} = M \texttt{in}\ E : \tau, A \cup A_2} \quad \text{(SE-Let1)}$$

$$\frac{\Gamma; \diamond \vdash^{A_r} E : \tau, A \quad B \neq \diamond}{\Gamma; B \vdash^{A_r} \texttt{ok}; E : \tau, A} \quad \text{(SE-Acc)}$$

$$\frac{\Gamma; \diamond \vdash^{A_r} E : \tau, A \quad B \neq \diamond}{\Gamma; B \vdash^{A_r} \texttt{sup}; E : \tau, A \cup B} \quad \text{(SE-Sup)}$$

$$\frac{\Gamma \vdash M : \mathsf{act}(A') \quad \Gamma; B \vdash^{A_r} E : \tau, A}{\Gamma; B \vdash^{A_r} \texttt{ins}(M); E : \tau, A \cup A'} \quad \text{(SE-Ins)}$$

$$\frac{\Gamma; A_r \vdash^{A_r} E_1 : \tau, A \quad \Gamma; \diamond \vdash^{A_r} E_2 : \tau, A}{\Gamma; B \vdash^{A_r} (\texttt{next} \to E_1 \mid \texttt{done} \to E_2) : \tau, A} \quad \text{(SE-Next)}$$

$$\frac{\Gamma, x{:}\mathsf{act}(B); B \vdash^{A_r} E : \tau, A \quad B \neq \diamond}{\Gamma; B \vdash^{A_r} x \ \to E : \tau, A} \quad \text{(SE-Bind)}$$

$$\frac{\begin{array}{c}\Gamma; B \cap_\diamond A' \vdash^{A_r} E_1 : \tau, A \qquad \Gamma; B \setminus_\diamond A' \vdash^{A_r} E_2 : \tau, A \\ A' \subseteq A_r \qquad B \neq \diamond\end{array}}{\Gamma; B \vdash^{A_r} \texttt{acase}\ (\star \ \subseteq A')\ (E_1 \mid E_2) : \tau, A} \quad \text{(SE-Acase)}$$

$$\frac{\begin{array}{c}\Gamma \vdash M : \tau_1 + \tau_2 \\ \Gamma, x{:}\tau_1; B \vdash^{A_r} E_1 : \tau, A \qquad \Gamma, x{:}\tau_2; B \vdash^{A_r} E_2 : \tau, A\end{array}}{\Gamma; B \vdash^{A_r} \texttt{case}\ M\ (x \to E_1 \mid x \to E_2) : \tau, A} \quad \text{(SE-Case)}$$

$$\frac{}{\Gamma; B \vdash^{A_r} \texttt{any} : \mathsf{unit}, \emptyset} \quad \text{(SE-Any)}$$

$$\frac{\begin{array}{c}\Gamma; B \sqcup A_1 \vdash^{A_1} E_1 : \tau_1, A_3 \qquad \Gamma; B \sqcup A_2 \vdash^{A_2} E_2 : \tau_2, A_4 \\ A_1 \cap A_4 = A_2 \cap A_3 = \emptyset \qquad A_1 \cup A_2 = A_r\end{array}}{\Gamma; B \vdash^{A_r} E_1 \wedge^{A_1, A_2} E_2 : \tau_1 \times \tau_2, A_3 \cup A_4} \quad \text{(SE-ParCon)}$$

$$\frac{\Gamma; B \sqcup A_1 \vdash^{A_1} E_1 : \tau_1, A_3 \quad \Gamma; \diamond \vdash^{A_2} E_2 : \tau_2, A_4 \quad A_1 \cup A_2 = A_r}{\Gamma; B \vdash^{A_r} E_1 \triangle^{A_1, A_2} E_2 : \tau_1 \times \tau_2, A_3 \cup A_4} \quad \text{(SE-SeqCon1)}$$

$$\frac{}{\Gamma; B \vdash^{A_r} \texttt{halt} : \tau, \emptyset} \quad \text{(SE-Halt)}$$

$$\frac{\begin{array}{c}\Gamma; B \sqcup A_1 \vdash^{A_1} E_1 : \tau_1, A_3 \qquad \Gamma; B \sqcup A_2 \vdash^{A_2} E_2 : \tau_2, A_4 \\ A_1 \cap A_4 = A_2 \cap A_3 = \emptyset \qquad A_1 \cup A_2 = A_r\end{array}}{\Gamma; B \vdash^{A_r} E_1 \vee^{A_1, A_2}_{\tau_1 + \tau_2} E_2 : \tau_1 + \tau_2, A_3 \cup A_4} \quad \text{(SE-ParDis)}$$

$$\frac{\Gamma; B \sqcup A_1 \vdash^{A_1} E_1 : \tau_1, A_3 \quad \Gamma; \diamond \vdash^{A_2} E_2 : \tau_2, A_4 \quad A_1 \cup A_2 = A_r}{\Gamma; B \vdash^{A_r} E_1 \bigtriangledown^{A_1, A_2}_{\tau_1 + \tau_2} E_2 : \tau_1 + \tau_2, A_3 \cup A_4} \quad \text{(SE-SeqDis1)}$$

$$\frac{\Gamma; B' \vdash^{A_r} E : \tau', A' \quad (B' = \diamond \ \texttt{or} \ B \subseteq B') \quad \vdash \tau' \leq \tau \quad A' \subseteq A}{\Gamma; B \vdash^{A_r} E : \tau, A} \quad \text{(SE-Sub)}$$

Terms have no effects, so they are well typed with respect to any next action (SE-Ret).

The `let` rule (SE-Let1) requires M to be a policy with a regulated set that is a subset of the current computation's regulated set. When this policy returns, we will have no information regarding the next action because the suspended policy may have accepted or suppressed an arbitrary number of actions. As a result, we check E in a context involving $\diamond$.

Rules (SE-Acc) and (SE-Sup) have similar structure. In both cases, we must be sure that the target has produced some action to be accepted or suppressed (i.e., $B \neq \diamond$). The main difference between the two rules is that we record the effect of the suppression, whereas acceptance has no effect. The rule for invoking actions (SE-Ins) adds A' to the effect of the computation when the action called belongs to the set A' (in other words, when the action has type $\mathsf{act}(A')$).

The next/done construct adds A_r to the context for checking E_1 and $\diamond$ for checking E_2 since we only take the first branch when we see an action in the regulated set and we take the second branch when there are no more actions (rule (SE-Next)). Rule (SE-Acase) takes the first or second branch depending upon whether the current action is in the set A_1. We refine the context in each branch to reflect the information that we have about the current action.

Rule (SE-ParCon) places several constraints on parallel conjunction of computations. Since the next action could be in the regulated set of the conjunction but not in the regulated sets of both E_1 and E_2, E_1 and E_2 must both be well typed either with respect to a subset of their regulated sets or with respect to $\diamond$. This is ensured by typing the subcomputations with respect to $B \sqcup A_1$ and $B \sqcup A_2$. In addition, there is not allowed to be a conflict between the regulated actions of one subcomputation and the effects of the other. Finally, the regulated set of the conjunction must be the union of the regulated sets of the subcomputations.

The first rule for sequential conjunction (SE-SeqCon1) is similar, with two exceptions. First, there is no constraint on the regulated and effect sets of the subcomputations. Second, E_2 must be well typed with respect to $\diamond$ because we cannot make any assumption about what the next action will be (it may be an action emitted by E_1, or E_1 may suppress all actions until the target has finished executing).

The rules for the disjunctive operators (SE-ParDis and SE-SeqDis1) are identical to their conjunctive counterparts except that they have sum types rather than pair types.

The subsumption rule for computations (SE-Sub) is invariant in regulated sets, covariant in type and effect sets, and contravariant in the type of the next action. It is always OK to consider that a computation has more effects than it actually does. In addition, a computation typed with respect to the possible next actions B' continues to be well typed even if more information about the next action is available.

3.3 Operational Semantics and Safety

We have defined a formal operational semantics and proven the safety of our language using progress and preservation. This result not only guarantees that the ordinary sorts of errors do not occur during evaluation but also rules out various policy conflicts (such as one computation in a parallel conjunction accepting a target action while the other computation suppresses it). The proof is quite long and detailed, but well worth the effort: it helped us catch numerous errors in preliminary versions of our system. Please see our technical report for details [2].

4 Discussion

4.1 Related Work

The SDS-940 system at Berkeley [6] was the first to use code rewriting to enforce security properties. More recently, the advent of safe languages such as Java, Haskell, ML, Modula, and Scheme, which allow untrusted applications to interoperate in the same address space with system services, has led to renewed efforts to design flexible and secure monitoring systems. For example, Evans and Twyman's Naccio system [9] allows security architects to declare *resources*, which are security-relevant interfaces, and to attach *properties*, which are bundles of security state and checking code, to these resources. Erlingsson and Schneider's SASI language [7] and later Poet and Pslang system [8] provide similar power. Grimm and Bershad [12] describe and evaluate a flexible mechanism that separates the access-control mechanism from policy in the SPIN extensible operating system. Finally, the Ariel project [19] allows security experts to write boolean constraints that determine whether or not a method can be invoked.

A shortcoming of all these projects is a lack of formal semantics for the proposed languages and systems. Without a formal semantics, system implementers have no tools for precise reasoning about their systems. They also do not provide a general set of primitives that programmers can use to explicitly construct complex policies from simpler ones.

A slightly different approach to program monitoring is taken by Lee et al. [14, 15] and Sandholm and Schwarzbach [21]. Rather than writing an explicit program to monitor applications as we do, they specify the safety property in which they are interested either in a specialized temporal logic (Lee et al.) or second-order monadic logic (Sandholm and Schwarzbach).

Many monitoring systems may be viewed as a specialized form of aspect-oriented programming. Aspect-oriented languages such as AspectJ [13] allow programmers to specify *pointcuts*, which are collections of program points and *advice*, which is code that is inserted at a specified pointcut. Wand et al. [23] give a denotational semantics for these features using monadic operations. Conflicting advice inserted at the same pointcut is a known problem in aspect-oriented programming. AspectJ solves the problem by specifying a list of rules that determine the order in which advice will be applied. We believe that our language, which allows explicit composition of policies and makes it possible to statically check composed policies for interference, is a more flexible approach to solving this problem.

Theoretical work by Alpern and Schneider [1, 22] gives an automaton-theoretic characterization of safety, liveness, and execution monitoring (EM) policies. EM policies are the class of policies enforceable by a general-purpose program monitor that may terminate the target, but may not otherwise modify target behavior. This class of program monitors (called security automata) corresponds precisely to our effect-free monitors, and consequently, as pointed out by Schneider, they are easily composed. We have previously extended Schneider's work by defining a new class of automata [3, 4], the *edit automata*, which are able to insert and

suppress target actions as well as terminate the target. Edit automata more accurately characterize practical security monitors that modify program behavior. We proved that under certain realistic assumptions such automata are strictly more powerful than security automata.

4.2 Current and Future Work

In order to confirm that our policy calculus is feasible and useful, we have developed a practical implementation of it [2]. Polymer, our language for writing policies, implements most of the policy calculus but uses Java, rather than a lambda calculus, as the core language. Enforcing a policy on a target application involves compiling the Polymer policy into Java bytecode and instrumenting the target to call the policy prior to executing any security-relevant instructions. For simplicity, the target programs we currently consider are Java source programs, but many of the techniques we use can also be extended to handle Java bytecode. We have not yet implemented static checking of effects.

Our immediate concern is to acquire more experience applying our tool to enforcing security policies on realistic applications. We are interested both in testing our tool on untrusted mobile programs as well as using it to make programs and services written by trusted programmers more robust. As an example of the latter application, we intend to follow Qie et al. [20] and use our tool to control resource consumption and to help prevent denial of service in Web servers.

Rather than having an external tool that rewrites Java programs to enforce policies, we are working on internalizing the rewriting process within an extension to the Java language. We hope to develop techniques that allow programmers to dynamically rewrite programs or parts of programs and to update or modify security policies without necessarily bringing down the system. We believe the idea of policies as first-class objects will be crucial in this enterprise.

We plan to implement a mechanism that would allow policy writers to group related methods into "abstract" actions (e.g., constructors for `java.io.FileInputStream`, `java.io.FileWriter`, etc., could be referred to by the abstract action `FileOpenAction(String filename)`). The independence of these abstract actions from the underlying system would allow policies to be ported from one system to another by changing only the definitions of the abstract actions.

We are also investigating additional combinators that could be added to our language. In particular, we are interested in developing fixed-point combinators that extend our sequential operators. These combinators would iteratively combine two policies without restricting their effects or requiring that one supersedes the other. We would also like to make it possible for policy writers to develop their own combinators.

Acknowledgments

The authors would like to thank Dan Wallach for suggesting the Chinese wall policy as a good example of a disjunctive policy. We are also grateful to Dan Grossman for commenting on a draft of this paper.

References

1. Bowen Alpern and Fred Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.
2. Lujo Bauer, Jarred Ligatti, and David Walker. A calculus for composing security policies. Technical Report TR-655-02, Princeton University, 2002. Forthcoming.
3. Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. In *Foundations of Computer Security*, Copenhagen, Denmark, July 2002.
4. Lujo Bauer, Jarred Ligatti, and David Walker. More enforceable security policies. Technical Report TR-649-02, Princeton University, June 2002.
5. David Brewer and Michael Nash. The Chinese wall security policy. In *IEEE Symposium on Security and Privacy*, pages 206–214, Oakland, May 1989.
6. P. Deutsch and C. A. Grant. A flexible measurement tool for software systems. In *Information Processing*, pages 320–326, 1971. Appeared in the proceedings of the IFIP Congress.
7. Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, pages 87–95, Caledon Hills, Canada, September 1999.
8. Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, Oakland, California, May 2000.
9. David Evans and Andrew Twyman. Flexible policy-directed code safety. In *IEEE Security and Privacy*, Oakland, CA, May 1999.
10. Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. In *ACM Symposium on Principles of Programming Languages*, London, UK, January 2001.
11. John Gough. *Compiling for the .NET Common Language Runtime*. Prentice Hall, 2001.
12. Robert Grimm and Brian Bershad. Separating access control policy, enforcement and functionality in extensible systems. *ACM Transactions on Computer Systems*, pages 36–70, February 2001.
13. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of AspectJ. In *European Conference on Object-oriented Programming*. Springer-Verlag, 2001.
14. Moonjoo Kim, Mahesh Viswanathan, Hanene Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *European Conference on Real-time Systems*, York, UK, June 1999.
15. Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. Run-time assurance based on formal specifications. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, June 1999.
16. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

17. Erik Meijer and John Gough. A technical overview of the Common Language Infrastructure. `http://research.microsoft.com/~emeijer/Papers/CLR.pdf`.
18. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
19. R. Pandey and B. Hashii. Providing fine-grained access control for Java programs through binary editing. *Concurrency: Practice and Experience*, 12(14):1405–1430, 2000.
20. Xiaohu Qie, Ruoming Pang, and Larry Peterson. Defensive programming: Using an annotation toolkit to build DoS-resistant software. Technical Report TR-658-02, Princeton University, July 2002.
21. Anders Sandholm and Michael Schwartzbach. Distributed safety controllers for web services. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 270–284. Springer-Verlag, 1998.
22. Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and Systems Security*, 3(1):30–50, February 2000.
23. Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Workshop on Foundations of Aspect-Oriented Languages*, 2002.