

# Aspect-Oriented In-lined Reference Monitors \*

Kevin W. Hamlen    Micah Jones

University of Texas at Dallas  
hamlen@utdallas.edu, micah.jones1@student.utdallas.edu

## Abstract

An Aspect-Oriented, declarative, security policy specification language is presented, for enforcement by In-lined Reference Monitors. The semantics of the language establishes a formal connection between Aspect-Oriented Programming and In-lined Reference Monitoring wherein policy specifications denote *Aspect-Oriented security automata*—security automata whose edge labels are encoded as pointcut expressions. The prototype language implementation enforces these security policies by automatically rewriting Java bytecode programs so as to detect and prevent policy violations at runtime.

**Categories and Subject Descriptors** D.3.1 [*Programming Languages*]: Formal Definitions and Theory; D.4.6 [*Operating Systems*]: Security and Protection—access controls; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—denotational semantics

**General Terms** Languages, Security

**Keywords** Aspect-Oriented Programming, In-lined Reference Monitors, Object-Oriented Programming, Runtime Verification, security automata

## 1. Introduction

Over the past 15 years, *In-lined Reference Monitors* (IRM's) [26] have emerged as a powerful, flexible method of enforcing security policies over untrusted, mobile code. In an IRM framework, untrusted codes (e.g., binary executables) are automatically rewritten according to a security policy specification before they are executed. The rewriting process inserts dynamic security checks into the untrusted code to detect and prevent impending policy violations at runtime. The resulting *self-monitoring* program can be safely executed without additional security checks imposed by the operating system or hardware.

Independently of this research, Aspect-Oriented Programming (AOP) [18] has emerged as an elegant paradigm for expressing cross-cutting code transformations at the source level. An *aspect* consists of source code fragments (*advice*), and *pointcut* specifications that dictate where these fragments should be injected throughout the code. At compile-time an *aspect weaver* merges the aspects

with the rest of the code to create a single executable. Aspects are useful because they allow programmers to consolidate code associated with a given cross-cutting concern. For example, the requirement that all file operations must be logged could be implemented by an aspect that injects the appropriate logging operation just before every file operation.

Numerous authors (c.f., [27, 28, 6, 7]) have observed that AOP lends itself to the implementation of security enforcement mechanisms like IRM's. For example, the Java-MOP [5] runtime verification system is implemented atop AspectJ [19], an AOP extension for Java. Similarly, the Polymer system [4] implements IRM's for Java through what is essentially an aspect-weaving process. The approach is quite powerful, having been shown to enforce a large class of important security properties that includes both safety and some liveness properties [23]. However, one disadvantage of representing security policies as aspects is that full aspects contain non-declarative code fragments (the advice) that can be difficult for policy-writers to write and reason about correctly. Aspect-weaving is a complex process that can result in surprising and unforeseen runtime behavior once the aspects are merged into the rest of the code; so it can be unclear whether a given aspect actually enforces the higher-level security concern it was intended to implement.

We present the design and implementation of a *purely declarative*, Aspect-Oriented security policy specification language for In-lined Reference Monitoring. In our analysis we define a formal denotational semantics for our language that merges the semantics of AOP languages and that of IRM's. The result is a language in which policies denote *Aspect-Oriented security automata*—security automata whose edge labels are encoded as pointcut expressions. Each policy in our language therefore encodes a property that can be said to be true or false of rewritten code apart from any original, unmodified code from which it may have been derived. The property modeled by a policy is the acceptance condition of the automaton it denotes.

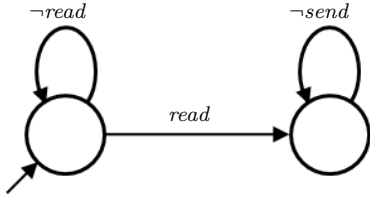
The existence of a formal denotational semantics for the language is useful because it provides a means of formally proving that untrusted code satisfies a specified security policy. This provides the necessary theoretical foundation whereby a certifying compiler (c.f., [24]) or certifying In-lined Reference Monitoring system (c.f., [17, 2]) can generate a proof of policy-adherence for the code it produces. Code-recipients can use such proofs to independently verify that the code is safe to execute even when the code-producer is not trusted. A denotational semantics also facilitates the stronger objective of formally verifying that a program-rewriting system always produces policy-adherent code.

The purely declarative nature of our language means that policies define *what* security property to enforce without overspecifying *how* it is to be enforced. For any given policy there will typically be many possible rewriting strategies that enforce it. This flexibility affords IRM implementations the freedom to choose an optimal rewriting algorithm based on architectural details, the results of program analyses, and other information that becomes available at

\* This research was made possible by AFOSR YIP award number FA9550-08-1-0044 and a grant from Texas Enterprise Funds.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PLAS'08 June 8, 2008, Tucson, Arizona, USA.  
Copyright © 2008 ACM 978-1-59593-936-4/08/06...\$5.00



**Figure 1.** A security automaton prohibiting *send* after *read*

rewriting time. This also makes our language suitable for separate certification as discussed above. That is, a separate verifier could in principle examine rewritten code to determine whether it actually satisfies the security policy. We consider this to be important for building trustworthy IRM systems since it constitutes an extra level of redundancy for detecting and debugging incorrect policy specifications.

In making policies purely declarative we do not exclude the possibility that some rewriter implementations might also accept additional input from the policy-writer suggesting how the policy should be enforced. For example, policy-writers might suggest remedial actions (e.g., premature termination, roll-back, etc.) expressed as imperative code fragments to be executed in the event that a policy violation would have otherwise occurred. However, this information is not trusted and therefore remains separate from the policy. Hence, if the implementation of roll-back includes an operation that constitutes a violation of the security policy, this flaw in the rewriting algorithm can be detected and rejected by a verifier.

The remainder of the paper is structured as follows. Section 2 describes the syntax of our policy specification language, called SPoX, and informally describes its semantics. A formal denotational semantics for SPoX is provided in Section 3 along with discussion of various rewriting strategies that can be employed to enforce SPoX policies. Section 4 discusses our prototype implementation of a Java IRM framework for SPoX. Section 5 discusses related work and Section 6 concludes.

## 2. Language Syntax

SPoX (Security Policy XML) is an XML-based security policy specification language suitable for enforcement by IRM’s. A SPoX specification defines a *security automaton* [26]—a finite- or infinite-state machine that accepts all and only those *event sequences* that satisfy a security policy. For example, the security automaton in Figure 1 encodes the policy that prohibits any network *send* operation after a file *read* operation. Such a policy might be used to prevent untrusted code from leaking files over the network. Observe that the transitions of the automaton are labeled with predicates that denote sets of security-relevant events—program operations that change the security state of the program. Any program operation satisfying the label of an outgoing edge from the current state causes the automaton to transition to the destination state. If no outgoing edge label satisfies the next operation to be executed, the operation is a policy violation and the automaton rejects.

A grammar for the core language of SPoX is given in Figure 2. SPoX specifications are lists of edge declarations, each consisting of three parts:

- *Pointcut expressions* identify sets of related security-relevant events that programs might exhibit at runtime. These serve as edge labels for the automaton.
- *Security-state variable* declarations abstract the security state of an arbitrary program. These serve as node labels for the automaton.

$n \in \mathbb{N}$	natural numbers	$c \in C$	class names
$sv \in SV$	state variables	$md \in MD$	method names
$iv \in IV$	iteration variables	$fd \in FD$	field names
$id \in ID$	object identifiers		
$pol ::= edg^*$			<b>policies</b>
$edg ::=$	<code>&lt;edge&gt;pcd ep* &lt;/edge&gt;</code>		<b>edges</b>
	<code>  &lt;forall var="iv" from="a1" to="a2"&gt;edg&lt;/forall&gt;</code>		edgesets
			iteration
$pcd ::=$	<code>&lt;call&gt;c.md&lt;/call&gt;</code>		<b>pointcuts</b>
	<code>  &lt;get&gt;c.fd&lt;/get&gt;   &lt;set&gt;c.fd&lt;/set&gt;</code>		method calls
	<code>  &lt;arg num="n" obj="id"&gt;vp&lt;/arg&gt;</code>		field accesses
	<code>  &lt;and&gt;pcd<sub>1</sub> pcd<sub>2</sub>&lt;/and&gt;</code>		stack args
	<code>  &lt;or&gt;pcd<sub>1</sub> pcd<sub>2</sub>&lt;/or&gt;</code>		conjunction
	<code>  &lt;not&gt;pcd&lt;/not&gt;</code>		disjunction
	<code>  &lt;cfld&gt;pcd&lt;/cfld&gt;</code>		negation
			temporal ops
$ep ::=$	<code>&lt;nodes [obj="id" var="sv"&gt;</code>		<b>edge endpoints</b>
	<code>  a<sub>1</sub>, a<sub>2</sub></code>		
	<code>&lt;/nodes&gt;</code>		
$vp ::=$	<code>&lt;true /&gt;   &lt;isnull /&gt;</code>		<b>value predicates</b>
$a ::=$	<code>n   iv   a<sub>1</sub>+a<sub>2</sub>   a<sub>1</sub>-a<sub>2</sub></code>		<b>arithmetic</b>
	<code>  a<sub>1</sub>*a<sub>2</sub>   a<sub>1</sub>/a<sub>2</sub>   (a)</code>		

**Figure 2.** SPoX core language syntax

- *Security-state transitions* describe how events cause the security automaton’s state to change at runtime. These define the transition relation for the automaton.

In the following paragraphs we define the language of edge labels (pointcuts), state labels (security-state variables), and transition relations supported by SPoX, along with an informal description of their semantics. A more formal treatment of the denotational semantics of SPoX along with strategies for enforcing SPoX policies are given in Section 3.

**Pointcuts.** SPoX expresses security automaton edge labels as pointcut designator expressions in the style of Aspect Oriented Programming (AOP) [18]. A pointcut designator defines a set of program-states that constitute security-relevant events, where a program-state consists of the complete runtime memory image of the program including the stack, code, and program-counter (i.e., the next instruction to be executed). For easy machine parsing, SPoX expresses pointcut expressions in an XML syntax. For example, the pointcut expression

```

<and>
  <call>File.renameTo</call>
  <not><arg num="1" obj="x"><isnull /></arg></not>
</and>

```

denotes the set of all program-states in which the next instruction to be executed is a call to the `renameTo` method of a Java `File` object, and the first argument being passed to the method is non-null.

The core language of pointcut expressions in Figure 2 includes support for method calls, field accesses, inspection of stack arguments, boolean operators, and the `cfld` temporal operator from AspectJ [19]. (Informally, a program state satisfies `<cfld>p</cfld>` if its call stack contains a frame satisfying

*p.*) The full SPoX pointcut language includes all relevant<sup>1</sup> pointcut operators from AspectJ including support for constructors, static initializers, exceptions, lexical scoping (e.g., “within”), and other temporal operators (e.g., “cflowbelow”). In addition to AspectJ pointcuts, the full language has an `<instr>` tag that can be used to identify any individual Java bytecode instruction as a security-relevant operation. The core language supports predicates that test the nullity of stack arguments; the full language additionally supports integer comparisons and string regular-expression matching.

**Security states.** A *security state* in a SPoX policy is a set of security-state variables and their integer values. These sets are dynamic in size; state transitions can add or remove state variables as well as change the value of existing state variables. Security-state variables come in two varieties:

- *Global security-state variables* are members of every security state; they cannot be added or removed by state transitions.
- *Instance security-state variables* describe the security state of an individual runtime object. Such variables are added to the security state by program operations that create a new instance of a security-relevant object, and they are removed from the security state by program operations that destroy a security-relevant object.

Instance security-state variables allow SPoX specifications to express policies that include per-object security properties. For example, a policy could require that each `File` object can be read at most ten times by defining an instance security-state variable associated with each `File` object and defining state transitions that increment the object’s security-state variable each time that individual `File` object is read (up to ten times). Global security-state variables allow SPoX specifications to express policies that include instance-independent security properties. For example, a policy could require that at most ten `File` objects may be created during the lifetime of the program by defining a global security-state variable that gets incremented each time any `File` object is created (up to ten times).

Security-state variables are not program variables; they are meta-variables declared by the SPoX specification purely for the purpose of defining the structure of a security automaton that encodes the policy to be enforced. However, rewriters might implement the security policy by reifying these meta-variables into the untrusted code and tracking their values at runtime. For example, a rewriter might add a new global runtime variable for each global security-state variable, and might add a new object field for each instance security-state variable. The rewriter could then add new runtime operations that update these new program variables whenever security-relevant operations occur, and might consult their values to decide whether an impending operation is a policy violation.

To simplify the core language syntax in Figure 2 we leave security-state variable declarations implicit. That is, for all variable names  $sv \in SV$  the policy implicitly declares a global security-state variable with name  $sv$  and an instance security-state variable with name  $sv$  associated with every class in  $C$ . Only a few of these meta-variables actually appear in any given SPoX policy, so the rewriter described above need only reify a small subset of these into rewritten programs. Our full-scale SPoX policy language makes security-state variable declarations explicit to help policy-writers detect mistyped variable names, typing errors, and other inconsistencies in the policy specification.

The start state of a SPoX security automaton is the state that assigns 0 to all global security-state variables and that has no in-

stance security-state variables (since no objects yet exist at program start). Each security-relevant program operation changes the current security state by adding or removing a finite set of zero or more instance security-state variables (corresponding to the finite set of security-relevant objects the operation creates or destroys at runtime). Security-relevant operations change the values of existing security-state variables (described in more detail below). Thus, each security state consists of a countably infinite set of security-state variables and their values, and the total number of security states in the automaton is at most countably infinite.

**Security-state Transitions.** Each edge in a security automaton is modeled as a triple  $(q_0, p, q_1)$ , where  $q_0$  is a source state,  $q_1$  is a destination state, and  $p$  is an edge label. In SPoX, edge labels are pointcut expressions and states are sets of security-state variables and their values. Since SPoX security automata have a potentially infinite number of states, we allow a single `<edge>` element to introduce a possibly infinite set of edges to the automaton. For example, the following SPoX fragment introduces an edge from every state in which global variable  $g$  has value 3 to a corresponding state in which  $g$  has value 4 (and all other security-state variables are unchanged):

```
<edge>p<nodes var="g">3,4</nodes></edge>
```

The effect is that program operations matching pointcut expression  $p$  will change the security state of variable  $g$  from 3 to 4.

To refer to instance security-state variables, pointcut expressions declare *object identifiers* associated with the security-relevant arguments of operations that match the expression. For example, the pointcut expression given by

```
<and>
  <call>File.renameTo</call>
  <and><arg num="0" obj="x"><true /></arg>
  <arg num="1" obj="y"><true /></arg></and>
</and>
```

declares two identifiers  $x$  and  $y$  that refer (respectively) to the `File` object whose `renameTo` method is about to be invoked and the object that is being passed as its first argument. One could then write

```
<nodes obj="x" var="v">0,1</nodes>
<nodes obj="y" var="v">0,0</nodes>
```

to specify that when the  $v$  security-state variable of both objects is 0, then the  $v$  security-state variable of the object whose method was invoked should change to 1, but that of the other object should remain unchanged. Note that `<nodes>` elements in an `<edge>` element are conjunctive. That is, a transition is introduced for each pair of states that satisfy *all* `<nodes>` elements given.

The security automata corresponding to many realistic security policies have repetitive, redundant structure. For example, the security automaton that permits at most 1000 `read` operations consists of 1001 states with edges from one to the next, each labeled `read`. To allow policy-writers to elegantly specify such structure, SPoX introduces a third kind of variable for iteration. As an example, the following fragment introduces the 1000 transitions described above:

```
<forall var="i" from="0" to="999">
  <edge>read
  <nodes var="g">i,i+1</nodes></edge>
</forall>
```

Here,  $i$  is an *iteration variable* that ranges from 0 to 999. Security-state variables and iteration variables can appear in simple arithmetic expressions (constants, addition, subtraction, multiplication, and division) to define the source and destination states of the transitions.

<sup>1</sup>The relevant AspectJ pointcut operators are those that do not concern advice.

$o \in Obj$	objects
$v ::= o \mid null$	values
$jp ::= \langle \rangle \mid \langle k, v^*, jp \rangle$	join points
$k ::= call\ c.md \mid get\ c.fd \mid set\ c.fd$	join kinds

**Figure 3.** Join points

$q \in Q = (SV \uplus (Obj \times SV)) \rightarrow \mathbb{N}$	security states
$S \in SM = (SV \uplus (ID \times SV)) \rightarrow \mathbb{N}$	state-variable maps
$I \in IM = IV \rightarrow \mathbb{N}$	iteration var maps
$b \in Bnd = ID \rightarrow Obj$	bindings
$r \in OBnd = Bnd \uplus \{Fail\}$	optional bindings
$\mathcal{P} : pol \rightarrow (\Upsilon \times 2^Q \times \Upsilon \times ((Q \times JP) \rightarrow 2^Q))$	policy denotations
$\mathcal{ES} : edg \rightarrow IM \rightarrow 2^{(JP \rightarrow OBnd) \times SM \times SM}$	edgeset denotations
$\mathcal{PC} : pcd \rightarrow JP \rightarrow OBnd$	pointcut denotations
$\mathcal{EP} : s \rightarrow IM \rightarrow (SM \times SM)$	endpoint constraints
$\mathcal{A} : a \rightarrow IM \rightarrow \mathbb{N}$	arithmetic

$$\begin{aligned}
\mathcal{P}[\langle edg_1 \dots edg_n \rangle] &= (Q, \{q_0\}, JP, \delta) \\
&\text{where } q_0 = (SV \uplus (Obj \times SV)) \times \{0\} \\
&\text{and } \delta(q, jp) = \{q[S'[b] \mid (f, S, S') \in \cup_{1 \leq i \leq n} \mathcal{ES}[\langle edg_i \rangle], \\
&\quad f(jp) = b, S'[b] \sqsubseteq q\} \\
\mathcal{ES}[\langle forall\ var="iv" from="a_1" to="a_2">edg</forall>]I &= \\
&\quad \cup_{\mathcal{A}[a_1]I \leq j \leq \mathcal{A}[a_2]I} \mathcal{ES}[\langle edg \rangle](I[j/iv]) \\
\mathcal{ES}[\langle edge>pcd\ ep_1 \dots ep_n</edge>]I &= \\
&\quad \{(\mathcal{PC}[\langle pcd \rangle], \sqcup_{1 \leq j \leq n} S_j, \sqcup_{1 \leq j \leq n} S'_j)\} \\
&\quad \text{where } \forall j \in \mathbb{N}. (1 \leq j \leq n) \Rightarrow ((S_j, S'_j) = \mathcal{EP}[ep_j]I) \\
\mathcal{PC}[\langle pcd \rangle]jp &= match-pcd(pcd)jp \\
\mathcal{EP}[\langle nodes\ var="sv">a_1, a_2</nodes>]I &= \\
&\quad (\{(sv, \mathcal{A}[a_1]I)\}, \{(sv, \mathcal{A}[a_2]I)\}) \\
\mathcal{EP}[\langle nodes\ obj="id" var="sv">a_1, a_2</nodes>]I &= \\
&\quad (\{((id, sv), \mathcal{A}[a_1]I)\}, \{((id, sv), \mathcal{A}[a_2]I)\}) \\
\mathcal{A}[n]I &= n \quad \mathcal{A}[a_1+a_2]I = \mathcal{A}[a_1]I + \mathcal{A}[a_2]I \\
\mathcal{A}[iv]I &= I(iv) \quad \mathcal{A}[a_1-a_2]I = \mathcal{A}[a_1]I - \mathcal{A}[a_2]I \\
&\quad \mathcal{A}[a_1 * a_2]I = \mathcal{A}[a_1]I \cdot \mathcal{A}[a_2]I \\
&\quad \mathcal{A}[a_1/a_2]I = \mathcal{A}[a_1]I / \mathcal{A}[a_2]I
\end{aligned}$$

**Figure 4.** Denotational semantics for SPoX

### 3. Analysis

#### 3.1 Denotational Semantics

In this section we define a formal semantics for SPoX that unambiguously identifies what a policy specification means, and what it means for a program to satisfy a SPoX policy. We begin by defining *join points* in Figure 3. Following the operational semantics of AOP [30], a join point is a recursive structure that abstracts the control stack. Join point  $\langle k, v^*, jp \rangle$  consists of static information  $k$  found at the site of the current program instruction, dynamic information  $v^*$  consisting of the arguments about to be consumed by the instruction, and recursive join point  $jp$  modeling the rest of the control stack. The empty control stack is modeled by the empty join point  $\langle \rangle$ .

A SPoX security policy denotes a security automaton whose alphabet is the universe  $JP$  of all join points. We refer to such an automaton as an *Aspect-Oriented security automaton*. Such an automaton accepts or rejects (possibly infinite) sequences of join

$$\begin{aligned}
match-pcd(\langle call>c.md</call>)(call\ c.md, v^*, jp) &= \perp \\
match-pcd(\langle get>c.fd</get>)(get\ c.fd, v^*, jp) &= \perp \\
match-pcd(\langle set>c.fd</set>)(set\ c.fd, v^*, jp) &= \perp \\
match-pcd(\langle arg\ num="n" obj="id">vp</arg>)(k, v_0 \dots v_n \dots, jp) \\
&= \{(id, v_n)\} \text{ if } vp = \langle true \rangle \text{ or } (vp = \langle isnull \rangle \text{ and } v_n = null) \\
match-pcd(\langle and>pcd_1\ pcd_2</and>)jp &= \\
&\quad match-pcd(pcd_1)jp \wedge match-pcd(pcd_2)jp \\
match-pcd(\langle or>pcd_1\ pcd_2</or>)jp &= \\
&\quad match-pcd(pcd_1)jp \vee match-pcd(pcd_2)jp \\
match-pcd(\langle not>pcd</not>)jp &= \neg match-pcd(pcd) \\
match-pcd(\langle cflow>pcd</cflow>)(k, v^*, jp) &= \\
&\quad match-pcd(pcd) \vee match-pcd(\langle cflow>pcd</cflow>)jp \\
match-pcd(pcd)jp &= Fail \text{ otherwise} \\
b \vee r &= b \quad Fail \wedge r = Fail \quad \neg Fail = \perp \\
Fail \vee r &= r \quad b \wedge Fail = Fail \quad \neg b = Fail \\
&\quad b \wedge b' = b \sqcup b'
\end{aligned}$$

**Figure 5.** Matching pointcuts to join points

points. A formal denotational semantics is provided in Figure 4. We use  $\uplus$  for disjoint union,  $\Upsilon$  for the class of all countable sets,  $2^A$  for the power set of  $A$ ,  $\sqsubseteq$  and  $\sqcup$  for the partial order relation and join operation (respectively) over the lattice of partial functions, and  $\perp$  for the partial function whose domain is empty. For partial functions  $f$  and  $g$  we write  $f[g] = \{(x, f(x)) \mid x \in Dom(f) \setminus Dom(g)\} \sqcup g$  to denote the replacement of assignments in  $f$  with those in  $g$ . When  $S \in SM$  is a state-variable map and  $b \in Bnd$  is a binding of object identifiers to objects, we let  $S[b]$  denote the partial function that results from substituting  $b$  into the domain of  $S$ :

$$\begin{aligned}
S[b] &= \{((b(id), sv), n) \mid ((id, sv), n) \in S\} \sqcup \\
&\quad \{(sv, n) \mid (sv, n) \in S\}
\end{aligned}$$

Security automata [26] are modeled in the literature as tuples  $(Q, Q_0, E, \delta)$  consisting of a set  $Q$  of states, a set  $Q_0 \subseteq Q$  of start states, an alphabet  $E$  of events, and a transition function  $\delta : (Q \times E) \rightarrow 2^Q$ . Security automata are non-deterministic; the automaton accepts an event sequence if and only if there exists an accepting path for the sequence. In the case of Aspect-Oriented security automata,  $Q$  is the set of partial functions from security-state variables to values,  $Q_0 = \{q_0\}$  is the initial state that assigns 0 to all security-state variables,  $E = JP$  is the universe of join points, and  $\delta$  is defined by the set of edge declarations in the policy (discussed below).

Each edge declaration in a SPoX policy defines a set of source states and the destination state to which each of these source states is mapped when a join point occurs that *matches* the edge's pointcut designator. The process of matching a pointcut designator to a join point binds identifiers in the pointcut to runtime objects in the program state abstracted by the join point. Thus, pointcut designators denote mappings from join points to bindings. The denotational semantics in Figure 4 defines this matching process in terms of the *match-pcd* function from the operational semantics of AspectJ [30]. We adapt this definition to SPoX syntax in Figure 5.

Defining what it means for a program to satisfy a policy requires an operational semantics that defines what it means to execute a program. For this purpose we adopt Flatt, Krishnamurthi and Felleisen's small-step operational semantics of CLASSICJAVA, as defined in [13] (with two minor changes introduced below). CLASSICJAVA programs consist of a sequence of class and interface declarations followed by an entrypoint expression that models the

$P ::= \text{def}^*(\text{let } \text{input} = v \text{ in } e)$	programs
$\text{def} ::= \text{class } c \{ \dots \} \mid \dots$	class defs
$e ::= v \mid se \mid mc \mid \text{ret}_{c.md(v^*)} e$	expressions
$se ::= \text{new } c \mid \text{var}$	simple expr
$\quad \mid (e:c).fd \mid (e:c).fd = e$	
$\quad \mid \text{view } c e \mid \text{let } \text{var} = e \text{ in } e$	
$mc ::= e.md(e^*) \mid (\text{super} \equiv \text{this}:c).md(e^*)$	method calls
$\mathbf{E} ::= [] \mid (\mathbf{E}:c).fd$	eval contexts
$\quad \mid (\mathbf{E}:c).fd = e \mid (v:c).fd = \mathbf{E}$	
$\quad \mid \mathbf{E}.md(e^*) \mid v.md(v^* \mathbf{E} e^*)$	
$\quad \mid (\text{super} \equiv v:c).md(v^* \mathbf{E} e^*)$	
$\quad \mid \text{view } c \mathbf{E} \mid \text{let } \text{var} = \mathbf{E} \text{ in } e$	

**Figure 6.** Syntax of CLASSICJAVA with **ret** (CJR)

$$\begin{array}{c}
\frac{P \vdash_{CJ} \langle \mathbf{E}[se], S \rangle \hookrightarrow \langle \mathbf{E}[e'], S' \rangle}{P \vdash_{CJR} \langle \mathbf{E}[se], S \rangle \hookrightarrow \langle \mathbf{E}[e'], S' \rangle} \\
\frac{P \vdash_{CJ} \langle \mathbf{E}[o.md(v^*)], S \rangle \hookrightarrow \langle \mathbf{E}[e'], S' \rangle \quad S(o) = \langle c, F \rangle}{P \vdash_{CJR} \langle \mathbf{E}[o.md(v^*)], S \rangle \hookrightarrow \langle \mathbf{E}[\text{ret}_{c.md(v^*)} e'], S' \rangle} \\
\frac{P \vdash_{CJ} \langle \mathbf{E}[(\text{super} \equiv \text{this}:c).md(v^*)], S \rangle \hookrightarrow \langle \mathbf{E}[e'], S' \rangle}{P \vdash_{CJR} \langle \mathbf{E}[(\text{super} \equiv \text{this}:c).md(v^*)], S \rangle \hookrightarrow \langle \mathbf{E}[\text{ret}_{c.md(v^*)} e'], S' \rangle} \\
\frac{P \vdash_{CJR} \langle \mathbf{E}[\text{ret}_{c.md(v^*)} v], S \rangle \hookrightarrow \langle \mathbf{E}[v], S \rangle}{P \vdash_{CJR} \langle \mathbf{E}[e], S \rangle \hookrightarrow \langle \mathbf{E}[e'], S' \rangle} \\
\frac{P \vdash_{CJR} \langle \mathbf{E}[\text{ret}_{c.md(v^*)} e], S \rangle \hookrightarrow \langle \mathbf{E}[\text{ret}_{c.md(v^*)} e'], S' \rangle}{P \vdash_{CJR} \langle \mathbf{E}[\text{ret}_{c.md(v^*)} e], S \rangle \hookrightarrow \langle \mathbf{E}[\text{ret}_{c.md(v^*)} e'], S' \rangle}
\end{array}$$

**Figure 7.** Operational semantics of CJR in terms of those for CLASSICJAVA

program’s main method. Small-step judgment  $P \vdash_{CJ} \langle e, S \rangle \hookrightarrow \langle e', S' \rangle$  asserts that in program  $P$ , expression  $e$  in store  $S$  evaluates to new expression  $e'$  and new store  $S'$ . Stores  $S : \text{Obj} \rightarrow (c \times F)$  map objects to class-tagged field records. A partial syntax is provided in Figure 6 for the reader’s convenience; for the full syntax and semantics the reader is invited to consult [13].

The syntax in Figure 6 differs from that in [13] in two important respects. First, to model program input we adopt the convention that the entrypoint expression must have the form  $(\text{let } \text{input} = v \text{ in } e)$ , where  $\text{input}$  is a reserved variable name and value  $v$  is the input supplied to the program. Thus, in our treatment each program  $P$  actually denotes the equivalence class of CLASSICJAVA programs obtained by substituting value  $v$  with any other value of equivalent type. Second, we introduce the expression  $\text{ret}_{c.md(v^*)} e$ , which indicates that subexpression  $e$  is the (partially reduced) body of method  $md$  of class  $c$  that was called with values  $v^*$  as parameters. These **ret** expressions make method-returns explicit. They do not affect expression evaluation but they make it possible to recover the runtime call-stack from a partially reduced expression, which is necessary for matching expressions to pointcut designators in our analysis.

Hereafter we refer to our modified language as CJR (CLASSICJAVA with **ret**). Figure 7 defines the small-step operational semantics of CJR in terms of those for CLASSICJAVA.

Some (but not all) CJR configurations  $c = \langle e, S \rangle$  are join points. In the context of SPoX policies, we consider join points to be abstractions of security-relevant program states. Function  $J(c, \langle \rangle)$  yields the join point that abstracts configuration  $c$  (or the empty join point  $\langle \rangle$  if  $c$  is not security-relevant). We lift  $J$  to configuration sequences  $\chi$  and to sets  $X$  of configuration sequences

$$\begin{array}{l}
J : (e \times jp) \rightarrow jp \\
J(\langle \mathbf{E}[(o:c).fd], S \rangle, jp) = \langle \text{get } c.f.d, S(o), jp \rangle \\
J(\langle \mathbf{E}[(o:c).fd = v], S \rangle, jp) = \langle \text{set } c.f.d, S(o), jp \rangle \\
J(\langle \mathbf{E}[o.md(v^*)], S \rangle, jp) = \langle \text{call } c.md, o v^*, jp \rangle \\
\quad \text{where } S(o) = \langle c, F \rangle \\
J(\langle \mathbf{E}[(\text{super} \equiv o:c).md(v^*)], S \rangle, jp) = \langle \text{call } c.md, o v^*, jp \rangle \\
J(\langle \mathbf{E}[\text{ret}_{c.md(v^*)} e], S \rangle, jp) = J(\langle e, S \rangle, \langle \text{call } c.md, v^*, jp \rangle) \\
J(\langle e, S \rangle, jp) = \langle \rangle \text{ for all other } e
\end{array}$$

**Figure 8.** Mapping partially reduced CJR expressions to join points

in the obvious ways:  $J(c_1 c_2 \dots) = J(c_1, \langle \rangle) J(c_2, \langle \rangle) \dots$  and  $J(X) = \{J(\chi) \mid \chi \in X\}$ .

Armed with these definitions we are finally able to formally define what it means for a CJR program to satisfy a SPoX policy:

**Definition 1** (Executions). *Let  $P = \text{def}^*(\text{let } \text{input} = v \text{ in } e)$  be a well-typed CJR program. An execution  $\chi$  of  $P$  is a finite or countably infinite sequence of configurations  $\langle e_0, S_0 \rangle \langle e_1, S_1 \rangle \dots$  such that  $e_0 = e$ ,  $S_0 = \{(\text{input}, v_0)\}$  where  $v_0$  has the same type<sup>2</sup> as  $v$ , and for all  $i < \text{length}(\chi) - 1$ ,  $P \vdash \chi_i \hookrightarrow \chi_{i+1}$  holds. Furthermore, if  $\chi$  is finite then there exists no configuration  $\langle e_n, S_n \rangle$  satisfying  $P \vdash \chi_{\text{length}(\chi)-1} \hookrightarrow \langle e_n, S_n \rangle$ . We denote the set of all executions of  $P$  by  $X_P$ .*

**Definition 2** (Policy-adherence). *A CJR program  $P$  satisfies SPoX policy  $pol$  if and only if  $J(X_P) \subseteq L(\mathcal{P}[\![pol]\!])$  holds, where  $L(A)$  denotes the language accepted by security automaton  $A$ .*

The above asserts that program  $P$  satisfies policy  $pol$  if and only if every execution of  $P$  is accepted by the Aspect-oriented security automaton that  $pol$  denotes. Thus, executing  $P$  will never result in a security violation.

### 3.2 Policy Enforcement

Most (but not all) SPoX policies can be enforced by an IRM system through the insertion of dynamic security checks into the untrusted code. Dynamic checks are required in general because many SPoX policies are not statically decidable. In particular, SPoX policies that involve predicates on runtime values (e.g., those with  $\langle \text{arg} \rangle$ ) will typically not be statically decidable since the general problem of deciding whether an arbitrary runtime value will satisfy an arbitrary predicate is equivalent to the halting problem. However, we argue in this section that SPoX policies are dynamically decidable, and we sketch a simple algorithm for inserting runtime checks into untrusted code to detect policy violations before they occur. This algorithm is the basis for our prototype implementation of SPoX.

A means of *detecting* impending policy violations before they occur is not always sufficient for an In-lined Reference Monitor to *enforce* the policy, however. The IRM can still fail if the decision algorithm for detecting policy violations commits a security violation when executed as part of the untrusted code. For example, the SPoX policy  $\langle \text{and} \rangle p \langle \text{not} \rangle p \langle \text{not} \rangle \langle \text{and} \rangle$  (where  $p$  is any pointcut) is unsatisfiable, rejecting all program states. Therefore there is no code that a rewriter could insert that would not itself violate the security policy. Unsatisfiable policies are a trivial example of this problem but there are more realistic policies that present similar

<sup>2</sup>Formally, there exists a type  $\tau$  such that CLASSICJAVA typing judgments [13]  $P, \{ \} \vdash_e v \Rightarrow v' : \tau$  and  $P, \{ \} \vdash_e v_0 \Rightarrow v'_0 : \tau$  both hold.

```

G(edg1 ... edgn)jp =
  guard: do {
    ES(edg1)jp ... ES(edgn)jp
    System.exit(1);
  } while (false);
ES(<forall var="iv" from="a1" to="a2">edg</forall>)jp =
  if (a1 <= a2)
    for (int iv=a1; iv<=a2; ++iv) { ES(edg)jp }
ES(<edge>pcd ep1 ... epn</edge>)jp =
  b = match-pcd(pcd)jp;
  if ((b!=Fail) && EP(ep1) && ... && EP(epn)) {
    EP'(ep1) ... EP'(epn)
    break guard;
  }
EP(<nodes var="sv">a1, a2</nodes>) =
  (SecurityState.sv == a1)
EP(<nodes obj="id" var="sv">a1, a2</nodes>) =
  (b(id).sv == a1)
EP'(<nodes var="sv">a1, a2</nodes>) =
  SecurityState.sv = a2;
EP'(<nodes obj="id" var="sv">a1, a2</nodes>) =
  b(id).sv = a2;

```

**Figure 9.** Java pseudo-code for a rewriting algorithm for SPoX

difficulties. For example, the policy

```

<edge>
  <not><arg num="1" obj="x"><true /></arg></not>
  <nodes var="g">0, 0</nodes>
</edge>

```

rejects any program whose evaluation stack grows to more than one element. This restriction disallows even basic arithmetic operations, prohibiting effective detection of policy violations through runtime checks.

In what follows we make the simplifying assumption that code inserted as part of the detection algorithm is not security-relevant. In practice, a certifying In-lined Reference Monitoring system can check this assumption by using the denotational semantics in Section 3.1 to verify code produced by the rewriter. Rewritten code that fails verification is rejected to prevent a security violation. Our prototype implementation discussed in §4 checks this conservatively by statically verifying that no rewriter-inserted operations satisfy any pointcut expression in the policy; thus, no inserted operations are security-relevant. More precise (but still conservative) verification algorithms are obviously possible (e.g., see [17, 2]); we leave the development of such a system to future work. To simplify the discussion, we also limit our attention in this section to policies that model deterministic security automata. Non-deterministic automata could be modeled by tracking sets of states at runtime instead of individual states.

As outlined in Section 2, an IRM can track a program’s security state at runtime by reifying security-state variables into the untrusted code. In particular, consider the following (non-optimized) rewriting procedure:

1. Inject a new `SecurityState` class into the untrusted code with a static field for each global security-state variable and an instance field for each instance security-state variable.
2. Rewrite each instruction that manipulates an object of type  $c \in C$  to instead manipulate an object pair of type  $c \times \text{SecurityState}$ , where the first member of the pair is the original object and the second member models the object’s security state. This expands each original instruction into a *chunk* of one or more rewritten instructions.

3. To each chunk, prepend an instruction sequence that first computes  $jp = J(\langle e, S \rangle, \langle \rangle)$ , where  $J$  is defined in Figure 8 and  $\langle e, S \rangle$  is the current program state; followed by instruction sequence  $G(\text{pol})jp$ , where  $\text{pol}$  is the policy and  $G$  is defined in Figure 9.

4. Finally, rewrite all static jumps in the original program to target the beginning of whichever chunk contains their destination addresses. (The only computed jumps in Java are method returns, which need not be rewritten.)

The rewriting procedure described above enforces a security policy by inserting a runtime security check before each program operation. When an impending security violation is detected, the program is prematurely terminated. This is only one method of rewriting untrusted code to enforce SPoX policies; clearly many other approaches also exist. For example, instead of premature termination, rewriters could implement other remedial actions such as event suppression, checkpointing with roll-back, or specific corrective operations specified by advice external to the policy.

The simple rewriting algorithm presented here does not produce particularly efficient code, but the code it produces can be significantly optimized through partial evaluation. For example, for many policies it can be statically determined that most instructions in the untrusted code are not security-relevant (e.g., they match no pointcut expression in the policy). For those instructions the code defined in Figure 9 partially evaluates to an empty instruction sequence. Thus, in practice a rewriter typically only inserts a few runtime security checks around the few program operations that might be security-relevant.

The code in Figure 9 can be optimized further by replacing the for-loops with a more efficient integer linear programming algorithm. In particular, each set of  $n$  nested `<forall>` elements in a SPoX policy that surround  $m$  `<nodes>` elements defines a rational polytope  $T \subseteq \mathbb{Q}^n$  with  $2(n + m)$  linear constraints. To decide whether the current runtime security state matches the source state of any edge defined by this structure it suffices to decide whether feasible region  $T$  contains an integer lattice point. (See [3] for a summary of efficient algorithms for computing this.) In the common case where each `<forall>` and `<nodes>` element refers to at most one iteration variable, polytope  $T$  is a box, and therefore the problem can be trivially decided with  $2n$  integer inequality tests and no loops.

The simple rewriting algorithm outlined in this section might fail when applied to Java code that is self-modifying or multi-threaded. Self-modifying code can be supported by adding the rewriter to the load path of the Java virtual machine, so that it can transform any modified code at runtime before it is first executed. Multi-threaded code can be supported by making each chunk produced by the rewriting algorithm atomic. Each of these solutions might have an adverse effect on performance so is worthy of further study. Our prototype implementation described in the next section leaves these enhancements to future work.

## 4. Implementation

Our implementation of SPoX is an application that rewrites Java bytecode programs in accordance with a SPoX policy specification by in-lining runtime security checks into the untrusted code. We implemented our rewriter in Java using Apache’s BCEL API [14] to read and write bytecode binaries. Discounting library code, the rewriter currently consists of about 4000 lines of Java source code.

The rewriter is still in development and does not yet support some features of SPoX, but already supports most core features including method call and field access monitoring, dynamic tests of argument values (`<arg>`), boolean operators, and global state-variables. Simple for-all-iteration is supported, but instance state-

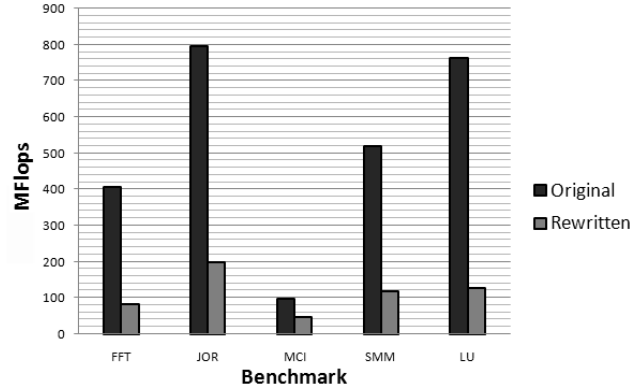
variables and temporal operators are part of current development work.<sup>3</sup> The implementation also supports features not covered in the core fragment of SPoX presented in this paper. This includes class, method, and field names specified as AspectJ-style regular expressions, and support for an `<instr>` element that allows individual Java bytecode instructions to be identified as security-relevant operations.

As a preliminary evaluation of our prototype we enforced security policies on two applications: RSSOwl and the SciMark benchmark suite. RSSOwl is a popular open-source RSS aggregator program, and is sufficiently large and complex to be a useful test of our implementation’s rewriting speed. It also makes extensive use of socket and file operations, allowing us to enforce some realistic example policies. SciMark is a benchmarking suite that focuses mainly on numerical computations. We used it to assess the runtime efficiency of rewritten programs.

For RSSOwl we enforced a policy that prevents new socket output streams from being acquired after any file in the `Windows` directory has been opened. A formal statement of the policy is given in the Appendix. Our rewriter enforced this policy by inserting dynamic checks before all constructor calls to classes whose fully qualified names begin with `java.io.File`. The dynamic checks included a regular expression match that tests whether the first argument passed to the method denotes a filename in the `Windows` directory. Guard instructions were also inserted before all calls to `java.net.Socket.getOutputStream`, forcing premature termination if such a call occurs after a restricted file has been accessed. There are obviously ways for malicious code to circumvent this simplistic policy (for example, by calling one of the Java runtime library methods that accepts filenames as a non-constructor argument), but this policy is short, easy to read, and works well for the RSSOwl program.

The original RSSOwl (v1.2.4) JAR file was approximately 5MB in size, and rewriting introduced no measurable increase in size. (In fact, the size decreased by about 65K, which we speculate is due to BCEL’s binary-writing algorithm producing a slightly more compact representation of some of the meta-data in the original binary.) Rewriting the application on an Intel 2.66GHz Core 2 Duo processor with 4GB of RAM took approximately 17.6 seconds at a rate of about 300 K/sec. Of that time, only about 8.9 seconds (51% of the total time) were spent analyzing and transforming the underlying bytecode. The remaining 8.7 seconds can be attributed to I/O operations, including unpacking and repacking the JAR’s contents. The runtime performance of the rewritten code could not be measured formally because RSSOwl requires user interaction when executed; however we did not observe any noticeable performance overhead due to the inserted security checks. Likewise, no behavioral change to the application was observed except in the event of a policy violation—reading a file from the `Windows` directory and then attempting to access the network resulted in premature termination of the application as intended.

To measure the runtime overhead introduced by the rewriter we applied a security policy to the SciMark benchmark suite and measured the performance of each benchmark before and after rewriting. SciMark includes five processor-intensive mathematical routines: Fast Fourier Transform (FFT), Jacobi Successive Over-Relaxation (JOR), Monte Carlo integration (MCI), sparse matrix multiplication (SMM), and dense LU matrix factorization (LU). The policy we applied prohibits a program from performing more than  $n$  floating point multiplication operations during its lifetime, where we set  $n$  to an unreachably high number to prevent the application from violating the policy and terminating prematurely.



**Figure 10.** Performance overhead from enforcing worst-case security policies on SciMark benchmarks

The rewriter enforced this policy by inserting runtime security checks around every `dmul` instruction in the untrusted code.

Figure 10 graphs the runtime overhead introduced by our rewriter by comparing the number of MFlops before and after rewriting for each benchmark. Overall we observed an average 76% reduction in performance with individual benchmarks ranging from 50% to 80%. We view these statistics as worst-case scenarios, since this particular policy was designed to force the rewriter to insert many dynamic checks within the innermost loops of intensive numerical calculations. For less pathological policies that we tried (e.g., monitoring method invocations), the runtime overhead introduced was so small as to be unmeasurable. The SciMark JAR file was 34K in size prior to rewriting and dropped to 31K afterward. (See the discussion above for a possible explanation for the size reduction.) Our rewriter loaded and transformed the entire suite in less than 0.5 seconds.

The preliminary experiments reported above are admittedly artificial and intended only to demonstrate the feasibility of implementing the core AOP features presented in this paper as part of an IRM system. Future work will involve more realistic case studies toward a more practical evaluation of the usefulness of the language for enforcing real security policies.

## 5. Related Work

Aspect-Oriented Programming was introduced by Kiczales et. al. [18] as a way of elegantly representing cross-cutting concerns at the source language level. A cross-cutting concern is a program feature whose implementation cuts across many different modules in a traditional language. A canonical example is that of logging, which traditionally requires inserting a call to the logging module from every module that performs a logged operation. In contrast, an Aspect-Oriented program implements each cross-cutting concern as a single aspect, where an aspect consists of a pointcut specifying which operations throughout the rest of the code should be modified by the aspect, and advice that provides code to be inserted around each such operation. Later work established a formal operational semantics for AOP in terms of the lambda calculus [29] and monadic theory [30].

AOP has been implemented as an extension to numerous languages. Two of the most widely used AOP extensions to Java are Hyper/J [25] and AspectJ [19]. More recently, aspectual variants of functional languages, such as AspectML [7], have been proposed and are a subject of current development. Functional AOP languages are particularly promising in the context of security en-

<sup>3</sup> AspectJ [19] implementations exist for both of these features, which we are porting to SPoX.

forcement because they can express provably effect-free advice [6], which is guaranteed not to contain security-relevant operations for certain common security policies.

In-lined Reference Monitors have existed in various guises since the days of the earliest compilers and operating systems; however it is only in the past decade that a formal theory of In-lined Reference Monitoring has been established. Schneider’s [26] seminal work on the subject was the first to propose In-lined Reference Monitors as a means of implementing *safety policies*—policies that prevent some “bad event” from happening. It showed that safety policies could be modeled as security automata. Later work [16, 22] showed that the general technique of program-rewriting could also be leveraged to enforce even broader classes of policies, including important classes of liveness policies.

The SASI system [11] was one of the earliest systems to implement these ideas. SASI enforces safety policies by rewriting x86 machine code programs and Java bytecode programs in accordance with a security policy expressed in PSLang. PSLang [10] is a simple imperative language that allows policy-writers to identify security-relevant instructions and the code that the rewriter should inject around each. Code specified in the policy and rewritten code produced by the rewriter are both trusted.

Several other Java In-lined Reference Monitoring systems followed, including Naccio [12], Java-MAC [20], Java-MOP [5], and Polymer [4]. Naccio is a source-to-source translator that enforces resource bound policies over untrusted C programs. Policies in Naccio are specified as a mixture of declarative and imperative code, with separate files for defining system resources, safety policies, and security-relevant operations specific to each architecture. Java-MAC and Java-MOP both enforce safety policies through Java bytecode rewriting. Java-MAC policies are specified in MEDL/PEDL—a policy language that defines security-state variables, security-relevant events (method calls or field accesses), and state changes effected by events. Java-MOP extends this approach with multiple specification logic engines (e.g., one for LTL [9] and another for JML [21]) and an implementation built atop AspectJ [19]. Its use of formal specification logics made it one of the first of these systems to have a formally defined denotational semantics. Polymer focuses on enforcing composable security policies in Java through an extensible collection of higher-level policy combinators. Policies in Polymer are specified in an AspectJ-like imperative language whose semantics are operational rather than denotational.

Mobile [17, 15] is a certifying In-lined Reference Monitoring system for the Microsoft .NET framework. It rewrites .NET CLI binaries according to a declarative security policy specification, producing a proof of policy-adherence in the form of typing annotations in an effect-based type system (c.f. [8]). These proofs are verified by a trusted type-checker to guarantee policy-adherence of rewritten code. We conjecture that Mobile’s policy specification language is equivalent in power to SPoX; but unlike SPoX, Mobile policies denote types in Mobile’s type system. Our work improves upon this by formally linking SPoX policies to AOP semantics and security automata, thereby establishing a closer connection between a policy specification and the underlying security property it encodes.

ConSpec [1] is a simplification of PSLang that restricts the non-declarative subset of PSLang to effect-free operations. The result is a language whose denotational semantics, like our work, is based on security automata and is suitable for formal certification [2]. We estimate that ConSpec has essentially the same expressive power as SPoX except that it lacks a rich language for expressing security-relevant events; security-relevant events in ConSpec are limited to method calls and are declared with method prototypes. In contrast, the central theme of our work is the incorporation of an AOP-style

pointcut language that includes boolean and temporal operators, field access predicates, value predicates, and predicates that can identify any instruction as security-relevant. This increases the expressive power of the language and promotes policy code-reuse, because individual state-update expressions can match more diverse collections of events (see §3.1).

## 6. Conclusion

We presented the design and implementation of SPoX: a purely declarative, Aspect-Oriented, security policy specification language. Our formal denotational semantics for the language unambiguously defined the security policy denoted by a SPoX specification as an Aspect-Oriented security automaton. The operational semantics of AspectJ and CLASSICJAVA were then leveraged to define what it means for a Java program to satisfy such a policy. Our prototype implementation enforced SPoX policies by automatically rewriting Java bytecode programs according to a SPoX specification. Preliminary evaluation of the prototype displayed good performance for many common cases, but high overhead for certain unusual cases such as policies that place controls on numerical operations in benchmark programs.

Our current implementation omits several important features of SPoX, such as complete support for iteration variables and instance security-state variables. Future work will therefore involve completing the implementation of these missing features for a more comprehensive evaluation of the tool. The current implementation also skips many obvious opportunities for optimizing bytecode during the rewriting process. Investigating these opportunities is the subject of current research.

The denotational semantics and definition of policy-adherence provided in this paper lays the foundation necessary for formally proving policy-adherence of rewritten programs, or even proving that a particular rewriting algorithm suffices to enforce a given class of supported security policies. Future work should therefore carry out such a proof, either by formally verifying a rewriter implementation or through the construction of a certifying rewriting system for SPoX in the style of a certifying compiler.

Although the declarative nature of our language helps policy-writers avoid many common mistakes that arise when policies are expressed as code, there is still much room for error when writing SPoX specifications. Part of our development work involves devising type systems and other analysis tools to help policy-writers verify that a given specification actually encodes the security policy they intend. Future work might also consider compiling SPoX policies from higher-level specifications such as checkboxes or flowcharts for a more user-friendly policy-development environment.

Finally, recent work on Aspect-Oriented functional languages provides a welcome opportunity for researchers to implement SPoX policies in languages more amenable to formal verification than C or Java. Such implementations could allow SPoX to be safely extended with more powerful runtime checking features and provide stronger guarantees to users that the instrumentation process is correct.

## Acknowledgments

The authors would like to thank Gopal Gupta for his comments and critiques of early drafts of this paper, and the anonymous reviewers for their contributions to the related work section.

## References

- [1] Irem Aktug and Katsiaryna Naliuka. ConSpec: A formal language for policy specification. In *Proc. of the 1st International*

- Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197-1 of Lecture Notes in Theoretical Computer Science, pages 45–58, Dresden, Germany, September 2007.
- [2] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *Proc. of the 15th International Symposium on Formal Methods (FM'08)*, Turku, Finland, May 2008. To appear.
- [3] Alexander Barvinok and James E. Pommersheim. An algorithmic theory of lattice points in polyhedra. *New Perspectives in Algebraic Combinatorics*, 38:91–147, 1999.
- [4] Lujo Bauer, Jay Ligatti, and David Walker. Composing security policies with Polymer. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 305–314, Chicago, Illinois, June 2005.
- [5] Feng Chen and Grigore Roşu. Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proc. of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, Edinburgh, Scotland, United Kingdom, April 2005.
- [6] Daniel S. Dantas and David Walker. Harmless advice. In *Proc. of the 8th ACM Symposium on Principles of Programming Languages (POPL)*, pages 383–396, Charleston, South Carolina, January 2006.
- [7] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, June 2008. To appear.
- [8] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *Proc. of the 18th European Conference on Object-Oriented Programming (ECOOP)*, pages 465–490, Oslo, Norway, June 2004.
- [9] E. Allen Emerson. *Handbook of Theoretical Computer Science*, chapter on Temporal and Modal Logic, pages 995–1072. Elsevier and MIT Press, 1990.
- [10] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, Ithaca, New York, January 2004.
- [11] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. of the New Security Paradigms Workshop (NSPW)*, pages 87–95, Caledon Hills, Ontario, Canada, September 1999.
- [12] David Evans and Andrew Twynman. Flexible policy-directed code safety. In *Proc. of the 20th IEEE Symposium on Security and Privacy*, pages 32–45, Oakland, California, May 1999.
- [13] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proc. of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 171–183, San Diego, California, January 1998.
- [14] Apache Software Foundation. Byte code engineering library, 2006. <http://jakarta.apache.org/bcel/>.
- [15] Kevin W. Hamlen. *Security Policy Enforcement by Automated Program-rewriting*. PhD thesis, Cornell University, Ithaca, New York, August 2006.
- [16] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 28(1):175–205, January 2006.
- [17] Kevin W. Hamlen, Greg Morrisett, and Fred B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. of the 1st ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 7–15, June 2006.
- [18] Gregor Kiczales, John Lamping, Anurag Medhdekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proc. of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241, pages 220–242, Jyväskylä, Finland, June 1997.
- [19] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proc. of the 15th European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, pages 327–355, Budapest, Hungary, June 2001.
- [20] Moonjoo Kim, Mahesh Viswanathan, Sampath Kannan, Insup Lee, and Oleg V. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, March 2004.
- [21] Gary T. Leavens and Albert L. Baker. Enhancing the pre- and postcondition technique for more expressive specifications. In *FM'99 – Formal Methods: World Congress on Formal Methods in Development of Computer Systems*, pages 1087–1106, Toulouse, France, September 1999.
- [22] Jarred Ligatti, Lujo Bauer, and David Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2): 2–16, February 2005.
- [23] Jay Ligatti, Lujo Bauer, and David Walker. Enforcing non-safety security policies with program monitors. In *Proc. of the 10th European Symposium on Research in Computer Security (ESORICS)*, pages 355–373, Milan, Italy, September 2005.
- [24] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proc. of the ACM Conference on Programming Language Design and Implementation (PLDI)*, pages 333–344, Montreal, Quebec, Canada, June 1998.
- [25] Harold Ossher and Peri Tarr. Hyper/J<sup>TM</sup>: Multi-dimensional separation of concerns for Java<sup>TM</sup>. In *Proc. of the 23rd International Conference on Software Engineering (ICSE)*, pages 729–730, Toronto, Ontario, Canada, May 2001.
- [26] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1): 30–50, February 2000.
- [27] Viren Shah and Frank Hill. An aspect-oriented security framework. In *Proc. of the DARPA Information Survivability Conference and Exposition*, volume 2, pages 143–145, April 2003.
- [28] John Viega, J.T. Bloch, and Pravir Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2), February 2001.
- [29] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proc. of the 8th International Conference on Functional Programming (ICFP)*, pages 127–139, Uppsala, Sweden, August 2003.
- [30] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):890–910, September 2004.

## Appendix

The following is a SPoX specification for the RSSOwl policy described in Section 4, which prohibits network sends after accessing any file in the Windows directory:

```
<?xml version="1.0"?>
<policy>
  <state name="s" />
  <edge>
    <and>
      <call>java.io.File*</call>
      <arg num="1">
        <streq>[A-Za-z]*:\\WINDOWS\\. *</streq>
      </arg>
    </and>
    <nodes var="s">0,1</nodes>
  </edge>
  <edge>
    <call>java.net.Socket.getOutputStream</call>
    <nodes var="s">1,#</nodes>
  </edge>
</policy>
```

Some syntax in this specification extends the SPoX core language fragment described in this paper. Of particular note is the use of `<state>` elements to explicitly declare security-state variable names, the value predicate `<streq>` for runtime regular expression matching on strings, and the reserved security-state label `#` for explicit policy violations (i.e., no security automaton edge).