Disambiguating Aspect-Oriented Security Policies[*]

Micah Jones
University of Texas at Dallas
micah.jones1@student.utdallas.edu

Kevin W. Hamlen
University of Texas at Dallas
hamlen@utdallas.edu

ABSTRACT

Many software security policies can be encoded as aspects that identify and guard security-relevant program operations. Bugs in these aspectually-implemented security policies often manifest as ambiguities in which aspects provide conflicting advice for a shared join point. The design and implementation of a detection algorithm for such ambiguities is presented and evaluated. The algorithm reduces advice conflict detection to a combination of boolean satisfiability, linear programming, and regular language non-emptiness. Case studies demonstrate that the analysis is useful for debugging aspect-oriented security policies for several existing aspectual security systems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.3.2 [**Programming Languages**]: Language Classifications—*Constraint and logic languages*; D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Languages, Security

Keywords

Aspect-oriented programming, declarative languages, reference monitors

1. INTRODUCTION

Aspect-oriented programming (AOP) is widely championed as a means of elegantly enforcing cross-cutting security concerns. Pointcuts allow aspects to conveniently identify security-relevant program operations, and advice associated with each pointcut can be leveraged to mandate runtime security checks, security state updates, and other interventionary code to prevent security-relevant operations from leading to policy violations.

However, expressing high-level security policies programmatically as aspects is often a difficult and potentially error-prone process. This is especially true when policies are intended to be generic, applying to a broad class of programs rather than just a few known programs. Correctly implementing generic, aspect-oriented security policies often requires highly non-trivial reasoning about how the aspect-weaving process might affect new, previously unseen, untrusted code. Unit testing tends to be an unreliable means of detecting errors in these policy encodings, since an incorrect aspectual policy implementation may correctly enforce the policy for most untrusted programs even if it permits policy-violating behavior or breaks policy-adherent behavior of a few unusual programs.

Two useful approaches to addressing the aspect-oriented security policy specification problem include eliminating side-effects from advice (often strengthening the pointcut language to compensate) [10, 11, 19] and synthesizing aspect-oriented policy implementations automatically from higher-level specification languages, such as LTL (cf., [7]) or TemporalZ [23]. This results in effect-free, purely declarative aspect-oriented languages that minimize the potential for undesired, effectful interactions with untrusted code.

However, policy errors are not limited to advice; they also arise in pointcuts. Pointcut errors tend to arise even in high-level specifications since the policy-writer must still somehow specify the set of security-relevant operations, and the language of operations is often an unfamiliar domain (e.g., Java bytecode instructions).

We have found that one of the most pernicious sources of error when writing complex, aspectual security policies is undesired *pointcut non-determinism.* Pointcut non-determinism arises when multiple pointcuts in an aspect-oriented security policy provide conflicting advice for a shared join point. As a trivial example, a policy that restricts file accesses might mandate different runtime security checks for calls to methods named `*Open*` than for calls to methods named `*Read*`. This policy might have unintended results when applied to a program that calls the `ReadOpenedFile` method, which matches both pointcuts. When pointcuts involve complex boolean expressions, regular expressions, class subtyping constraints, and mixtures of static and dynamic tests, even experts are prone to such mistakes.

We present the design and implementation of a pointcut analysis utility that automatically detects potential non-

[*]This research was supported by Young Investigator Award FA9550-08-1-0044 provided by the U.S. Air Force Office of Scientific Research.

AOSD March 15–19, Rennes and St. Malo, France
Copyright 2010 ACM 978-1-60558-958-9/10/03 ...$10.00.

determinism in pointcut libraries. Our tool targets the SPoX aspect-oriented policy specification language [19], which extends AspectJ [24] pointcuts with security state, instruction-level join point matching, and purely declarative dynamic inspection of runtime argument values. Other aspect-oriented security policy languages such as those supported by JavaMOP [7] have natural encodings in SPoX, permitting easy analysis of such policies using our utility. We have found this automated analysis invaluable for discovering bugs in policy specifications for real systems.

The detection algorithm is useful and practical in security contexts because it is program-agnostic, it is applied to a realistic aspect language, and it concerns specifications that encode program properties rather than program transformations. This last distinction is critical because protection systems that include formal verification require a property to verify, not a program transformation recipe. For example, proof-carrying code frameworks [30] and certifying IRM systems [20] both employ automated verifiers that expect security properties as input, against which untrusted code is checked. Traditional aspect-oriented programs that express transformations rather than properties are unsuitable for such purposes.

The remainder of the paper is structured as follows. Section 2 provides an overview of the SPoX language and its relationship to other AOP languages such as AspectJ. Section 3 details our algorithm for detecting pointcut non-determinism in SPoX policies. Our implementation of the algorithm in constraint-based Prolog is discussed in Section 4, including several case studies that illustrate how pointcut non-determinism bugs arise in practice and are detected by the analysis. Finally, related work and conclusions are summarized in Sections 5 and 6, respectively.

2. POLICY LANGUAGE

SPoX (Security Policy XML) [19] is a purely declarative yet powerful aspect-oriented policy specification language for constraining untrusted Java bytecode binaries. A SPoX specification denotes a *security automaton* [3]—a finite- or infinite-state machine that accepts all and only those *event sequences* that satisfy the security policy.

Security-relevant program *events* are specified in SPoX by pointcuts. The SPoX pointcut language is an extension of that used by AspectJ [24]. This allows policy-writers to develop policies that regard static and dynamic method calls and their arguments, object pointers, and lexical contexts, among other properties. SPoX supports all pointcuts from AspectJ that do not depend on advice, which allows it to specify a broad range of policies purely declaratively.

Prior work [22] has enforced SPoX policies as binary-level In-lined Reference Monitors (IRM's) [32]. An IRM system automatically instruments untrusted binaries with runtime security checks to produce *self-monitoring code*. The runtime checks preserve policy-adherent behavior while preventing policy-violating behavior. The instrumentation process is similar to aspect-weaving but is expressed as a binary-to-binary transformation rather than a compile-time, source-to-source or source-to-binary transformation. This allows SPoX policies to be enforced by code-consumers on binaries without source code.

In place of advice, SPoX policies include declarative specifications of how security-relevant events change the current security automaton state. The replacement of imperative advice with declarative state-transitions facilitates formal, automated reasoning about policies without the need to reason about arbitrary imperative code. State-transitions can be specified in terms of information gleaned from the current join point, such as method argument values, the call stack, and the current lexical scope. This allows advice typically encoded imperatively in most other aspect-oriented security languages to be declaratively encoded in SPoX policies. Typically this results in a natural translation from these other languages to SPoX, making SPoX an ideal target for our analysis.

$n \in \mathbb{Z}$	**integers**
$c \in C$	**class names**
$sv \in SV$	**state variables**
$iv \in IV$	**iteration vars**
$id \in ID$	**object identifiers**
$en \in EN$	**edge names**
$pn \in PCN$	**pointcut names**
$pol ::= np^* sd^* e^*$	**policies**
$np ::= \texttt{(pointcut name="}pn\texttt{"}\ pcd\texttt{)}$	**named pointcuts**
$sd ::= \texttt{(state name="}sv\texttt{"}\ [\texttt{"}c\texttt{"}]\texttt{)}$	**state declarations**
$e ::=$	**edges**
$\texttt{(edge name="}en\texttt{"}\ [\texttt{after}]\ pcd\ ep^*\texttt{)}$	edgesets
$\mid \texttt{(forall "}iv\texttt{" from}\ a_1\ \texttt{to}\ a_2\ e^*\texttt{)}$	iteration
$ep ::=$	**edge endpoints**
$\mid \texttt{(nodes}\ [\texttt{"}id\texttt{"}]\ \texttt{"}sv\texttt{"}\ a_1\texttt{,}a_2\texttt{)}$	state transitions
$\mid \texttt{(nodes}\ [\texttt{"}id\texttt{"}]\ \texttt{"}sv\texttt{"}\ a_1\texttt{,\#)}$	policy violations
$a ::= a_1\texttt{+}a_2 \mid a_1\texttt{-}a_2 \mid b$	**arithmetic**
$b ::= n \mid iv \mid b_1\texttt{*}b_2 \mid b_1\texttt{/}b_2 \mid \texttt{(}a\texttt{)}$	

Figure 1: Simplified SPoX syntax

The remainder of the section describes the syntax and semantics of SPoX. For readability we here use a simplified Lisp-style syntax; the implementation uses an XML-based syntax for easy parsing, portability, and extensibility. The language semantics are here described informally; a formal denotational semantics is provided in [19].

2.1 Security State

SPoX specifications (see Figures 1 and 2) are lists of security automaton edge declarations. Each edge declaration consists of three parts:

- *Pointcut expressions* (Figure 2) identify sets of related security-relevant events that programs might exhibit at runtime. These label the edges of the security automaton.

- *Security-state variable* declarations (*sd* in Figure 1) abstract the security state of an arbitrary program. The security state is defined by the set of all state variables and their integer[1] values. These label the automaton nodes.

[1] Binary operator / in Figure 1 denotes integer division.

$re \in RE$	**regular expressions**
$md \in MD$	**method names**
$fd \in FD$	**field names**
$pcd ::=$	**pointcuts**
(call mo^* rt $c.md$)	method calls
| (execution mo^* rt $c.md$)	callee executions
| (get mo^* $c.fd$)	field get
| (set mo^* $c.fd$)	field set
| (argval n ["id"] vp)	stack args (values)
| (argtyp n ["id"] c)	stack args (types)
| (target ["id"] [c])	object refs
| (withincode mo^* rt $c.md$)	lexical contexts
| (pointcutid "pn")	named pc refs
| (cflow pcd)	control flows
| (and pcd^*)	conjunction
| (or pcd^*)	disjunction
| (not pcd)	negation
$mo ::=$ public | private | $\cdots$	**modifiers**
$rt ::= c$ | void | $\cdots$	**return types**
$vp ::=$ (true)	**value predicates**
| (isnull)	object predicates
| (inteq n) | (intne n)	integer predicates
| (intle n) | (intge n)	
| (intlt n) | (intgt n)	
| (streq re)	string predicates

Figure 2: Simplified SPoX pointcut syntax

- *Security-state transitions* (e in Figure 1) describe how events cause the security automaton's state to change at runtime. These define the transition relation for the automaton.

Edges are specified by `edge` structures, each of which defines a (possibly infinite[2]) set of edges in the security automaton. Each `edge` structure consists of a pointcut expression and at least one `nodes` declaration. The pointcut expression defines a common label for the edges, while each `nodes` declaration imposes a transition pre-condition and post-condition for a particular state variable. The precondition constrains the set of source states to which the edge applies, and the post-condition describes how the state changes when an event matching the pointcut expression is exhibited.

As an example, Figure 3 shows a sample SPoX policy that requires programs to initialize instances of class `Clazz` before calling their methods. Line 1 declares a state variable `init` that tracks whether any given `Clazz` instance has been initialized. The edge at lines 3–6 transitions `init` from 0 to 1 when the object's constructor is invoked. The edge at lines 8–13 signals a policy violation when any of the object's other methods are called while `init` is 0. The reserved post-condition `#` in line 13 indicates that the security automaton should have no transition for this operation (hence it rejects).

```
1  (state name="init" "Clazz")
2
3  (edge name="initEdge"
4    (and (call "Clazz.new")
5         (target "x"))
6    (nodes "init" "x" 0,1))
7
8  (edge name="illegalMethodEdge"
9    (and (call "Clazz.*")
10        (not (call "Clazz.new"))
11        (not (withincode "Clazz.new"))
12        (target "x"))
13   (nodes "init" "x" 0,#))
```

Figure 3: `InitClazz` policy

Multiple `nodes` declarations in a single `edge` are interpreted conjunctively. For example, the following edge dictates that if an event matching pointcut `p` occurs when `s1` = 0 and `s2` = 1, then `s1` changes to 1 and `s2` remains invariant.

```
(edge name="edge1"
  (p)
  (nodes "s1" 0,1)
  (nodes "s2" 1,1))
```

In general, state variables come in two varieties:

- *Instance security-state variables* describe the security state of an individual runtime object. They can be thought of as hidden field members of security-relevant classes.
- *Global security-state variables* describe the state of the overall system. They can be thought of as hidden, global, program variables.

The `init` variable in Figure 3 is an instance security-state variable because it tracks whether any given object instance has been initialized. Such variables allow SPoX specifications to express per-object security properties. For example, a policy can require that each `File` object may be read at most ten times by defining an instance security-state variable associated with `File` objects and defining state transitions that increment each object's security-state variable each time that individual `File` object is read (up to ten times).

In contrast, global security-state variables allow SPoX specifications to express instance-independent security properties. For example, a policy can 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).

The security automata corresponding to many realistic security policies have repetitive, redundant structure. For example, a resource bound policy might implement a counter with automaton edges that increment the counter from i to $i + 1$ for each $i < b$ for some policy-specified bound b. To simplify such patterned repetition, SPoX includes iteration

[2] Infinity of the edge set arises from unboundedness of the set of objects; however, each object's automaton is finite.

variables, defined by forall structures. A forall may contain any number of edges and nested forall structures. For example, to describe a set of 10 edges, each of which increments the state variable s upon any event matching pointcut p, one could use the following SPoX fragment:

```
(forall "i" from 0 to 9
  (edge name="count"
    (p)
    (nodes "s" i,i+1)))
```

Iteration variables range over the integer lattice points of closed intervals. Thus, the above example allows state variable s to range from 0 to 10.

2.2 Pointcuts

A syntax for a subset of the SPoX pointcut language is given in Figure 2. SPoX pointcut expressions consist of all pointcuts available in AspectJ except for those that are specific to AspectJ's advice language.[3] This includes all regular expression operators available in AspectJ for specifying class and member names. Since SPoX policies are applied to Java bytecode binaries rather than to source code, the meaning of each pointcut expression is reflected down to the bytecode level. For example, the target pointcut matches any Java bytecode instruction whose *this* argument references an object of class c.

Instead of AspectJ's if pointcut (which evaluates an arbitrary, possibly effectful, Java boolean expression), SPoX provides a collection of effect-free *value predicates* that permit dynamic tests of argument values at join points. These are accessed via the argval predicate and include object nullity tests, integer equality and inequality tests, and string regular expression matching. Regular expression tests of non-string objects are evaluated by obtaining the toString representation of the object at runtime. (The call to the toString method itself is a potentially effectful operation and is treated as a matchable join point by the SPoX enforcement implementation. However, subsequent use of the returned string within injected security guard code is non-effectful.)

Predicates that match runtime arguments can also declare an identifier *id* to which nodes declarations refer in order to access the instance security state of security-relevant objects. For example, the following edge changes the s instance state variable of FileInputStream objects to 1 when they are closed.

```
(edge name="fileclose"
  (and (call void close)
       (target "x" java.io.FileInputStream))
  (nodes "x" "s" 0,1))
```

AspectJ pointcuts are further documented in [35], and SPoX extensions to them are documented in [19].

3. ANALYSIS

A SPoX policy is *non-deterministic* if it denotes a non-deterministic security automaton. Formally, we define policy non-determinism as follows.

Definition 1. A SPoX policy is non-deterministic if it denotes a security automaton in which there exists a state[4] $q \in (SV \times O) \rightharpoonup \mathbb{Z}$ and two edges (q, q_1) and (q, q_2) labeled with pointcuts pcd_1 and pcd_2 respectively, such that $q_1 \neq q_2$ and pcd_1 and pcd_2 match non-disjoint sets of join points.

IRM systems typically exhibit implementation-defined behavior when provided a non-deterministic policy specification as input. The implementation-defined behavior depends upon the order in which the IRM implements the runtime security checks that decide whether to perform each possible state transition at matching join points. The instrumentation (aspect-weaving) process does not detect the non-determinism statically because it cannot decide if multiple checks inserted at the same join point always result in mutually exclusive outcomes at runtime.

Two obvious solutions to this problem are to automatically determinize the security automaton prior to weaving it into untrusted code, or to automatically resolve conflicts by imposing a default ordering on conflicting edges. However, our experience indicates that these approaches often result in an automaton that does not reflect the intentions of the policy-writer. Instead, they often have the counterproductive effect of silently concealing policy design errors from the user. Non-determinism typically arises in practice when separate parts of a specification constrain event sets that the policy-writer expected to be disjoint, but that intersect at a few unusual join points or security states that the policy-writer did not adequately consider. Our goal is to bring these possible design errors to the attention of the policy-writer so that specification bugs can be manually corrected.

We therefore adopt the approach of detecting and rejecting non-deterministic policies automatically prior to enforcement. To support generic policies, the decision algorithm considers the universe of all possible untrusted target programs rather than a specific program to which the policy is to be applied. When non-determinism is detected, the algorithm yields a witness in the form of a security state and join point for which the non-determinism is exhibited. This allows policy-writers to understand and correct design flaws that may have led to the ambiguity.

At a high level the decision algorithm consists of two phases. In the first phase, state variable pre-conditions and post-conditions of every pair of edge declarations are compared to determine which edge pairs have common source states but distinct destination states. SPoX supports state transitions both immediately before and immediately after (via the after keyword in Figure 1) the matched join point. The SPoX language semantics adopt a point-in-time join point model [16] in which after-transitions are distinct from before-transitions of any following join point. Thus, an after edge can never conflict with a before edge in SPoX, and such edge pairs can be safely ignored when searching for potential non-determinism during the first phase.

In the second phase, pointcut labels of edge pairs identified in the first phase are compared to decide if their intersection is non-empty. If any labeled edge pair is identified as non-deterministic by both phases, a witness is then synthesized from a common source state from the first phase and

[3] For example, AspectJ's advicexecution() pointcut is omitted because SPoX lacks advice.

[4] Set O denotes the universe of object instances plus a *global meta-object* that models the storage context of global security state variables.

a common join point from the second phase. Both phases are discussed in detail in the following sections.

3.1 Security State Non-determinism

The problem of deciding whether there exist automaton edges (q, q_1) and (q, q_2) with $q_1 \neq q_2$ can be reduced to a linear programming problem. Each `edge` declaration in the policy defines its source (resp. destination) states in terms of pre-conditions (resp. post-conditions) that are expressed as integer equality constraints over state variables and iteration variables. Iteration variables are further constrained by inequality constraints imposed by the surrounding `forall` blocks that declare the iteration variables and the closed intervals over which their values range.

Thus, each set of n nested `forall` blocks that surround an `edge` containing m `nodes` elements defines a convex, rational polytope $T \subseteq \mathbb{Q}^n$ with $2(n+m)$ linear constraints. Each integer lattice point in feasible region T corresponds to a source state for an automaton edge defined by the `edge` declaration. The m post-condition constraints in such a structure define an affine transformation $f : T \to \mathbb{Q}^n$ that maps each source state to a destination state.

To decide whether an edge pair e_1, e_2 is potentially non-deterministic, we therefore adopt the following procedure.

1. Alpha-convert iteration variables to unique names. (Instance state variables are not renamed since the analysis must conservatively assume that all object references of similar type may alias.)

2. Compute feasible regions T_1 and T_2 for edges e_1 and e_2 by collecting the linear constraints encoded in relevant `forall` declarations and the pre-conditions of `nodes` declarations.

3. Compute affine transformations f_1 and f_2 by collecting the linear constraints encoded in the post-conditions of the `nodes` declarations.

4. Compute the set of common source states $T = T_1 \cap T_2$.

5. Compute extrema of objective function $m(q) = f_1(q) - f_2(q)$ over $q \in T$. If m has a maximum or minimum other than 0 at any $\bar{q} \in T$, then $\bar{q}$ is a common source state for which edges e_1 and e_2 lead to different destination states $f_1(\bar{q})$ and $f_2(\bar{q})$.

As an example, consider the following pair of edge declarations, which refer to the same instance state variable `s`:

```
(forall "i" from 0 to 10      (forall "i" from 0 to 3
  (argval 1 "x" p1)               (argval 2 "y" p2)
  (nodes "x" "s" i,i-3))          (nodes "y" "s" i*3,i+2))
```

Step 1 alpha-converts the iteration variables to unique names i_0 and i_1. Since objects `x` and `y` may alias (e.g., at a join point whose first and second arguments refer to the same object), instance state variable `s` is not alpha-converted. Step 2 then defines feasible region T_1 by linear constraints $(i_0 \geq 0) \wedge (i_0 \leq 10) \wedge (s = i_0)$ and feasible region T_2 by $(i_1 \geq 0) \wedge (i_1 \leq 3) \wedge (s = 3i_1)$. Transformations f_1 and f_2 are defined in Step 3 by $f_1(s) = s - 3$ and $f_2(s) = s/3 + 2$. Step 4 defines intersection $T = T_1 \cap T_2$ by the conjunction of the constraints defining T_1 and T_2.

Step 5 considers objective function $m(s) = s - 3 - (s/3 + 2)$, which is maximized in T at $(s, i_0, i_1) = (9, 9, 3)$ and minimized at $(0, 0, 0)$ (with values 1 and -5, respectively). Thus, non-determinism could potentially be exhibited when $\texttt{x.s} = \texttt{y.s} \in \{0, 9\}$ if there exists a program operation that matches both of pointcuts `p1` and `p2`. Disjointness of pointcut expressions is discussed in the next section.

```
(state name="c")
(forall "i" from 0 to maxint-1
  (edge name="cflowinc"
    (p)
    (nodes "c" i,i+1))
  (edge name="cflowdec" after
    (p)
    (nodes "c" i+1,i)))
```

Figure 4: SPoX fragment for (`cflow` p)

3.2 Pointcut Non-determinism

Pointcut disjointness is reducible to pointcut unsatisfiability. That is, pointcuts pcd_1 and pcd_2 match disjoint sets of join points if and only if pointcut (`and` pcd_1 pcd_2) is unsatisfiable. Pointcut satisfiability can, in turn, be reduced to boolean satisfiability (SAT). The remainder of the section describes an algorithm $\mathcal{S}$ that transforms a pointcut pcd into a boolean sentence $\mathcal{S}(pcd)$ such that sentence $\mathcal{S}(pcd)$ is satisfiable if and only if pointcut pcd is satisfiable.

Conjunction, disjunction, and negation in pointcut expressions can be translated directly to boolean conjunction, disjunction, and negation. The only remaining pointcut syntaxes are primitives (forms such as `call` that do not include nested pointcuts) and control flow operators such as `cflow`.

Control flow operators are removed from pointcuts prior to the analysis by translating them to an equivalent automaton encoding. For example, (`cflow` p) matches join points whose call stacks contain a frame matching pointcut p, which can be modeled by the policy fragment shown in Figure 4.[5] Control flow operators in edges are then replaced with equivalent `nodes` elements that stipulate an equivalent condition (e.g., $\texttt{c} \geq 1$).

This leaves the various pointcut primitives, such as `call`, `get`, and `argval`. For each unique primitive, a unique boolean variable is introduced. This results in a boolean sentence that is satisfiable if the original pointcut is satisfiable, but which might be satisfiable even if the original pointcut is unsatisfiable. For example, the pointcut

```
(and (call "File.open") (call "File.close"))
```

yields the sentence $a \wedge b$, which is satisfiable even though the original pointcut is not. To correct this, a constraint term must be added to the sentence for each pair of boolean variables that denote non-independent pointcut primitives. In this case the appropriate constraint term is $\neg(a \wedge b)$ since a and b denote primitives that cannot both be true for the same join point. A systematic approach to deriving constraint terms is provided later in the section.

Algorithm $\mathcal{S}$ can therefore be summarized as follows:

1. Construct a mapping $V : P \to B$ from pointcut primitives to unique boolean variables.

2. For each subset of pointcut primitives $S \subseteq P$, potentially generate a constraint term $c(S)$.

[5] In Figure 4, `c` is a thread-local instance state variable and is instantiated with a unique name for each cflow instance in the policy.

Table 1: Constraint generation cases

	call	exec	get	set	argv	targ	argt	with
call	CR							
exec	E	CR						
get	E	E	CR					
set	E	E	E	CR				
argv	C	C	E	C	RV			
targ	CR	CR	C	C	I	CR		
argt	C	C	E	C	T	I	CR	
with	I	E	I	I	I	I	I	CR

Legend:

I: independent (no constraint required)
E: mutually exclusive (use constraint $\neg(a \wedge b)$)
C: independent except for known classes
R: regular expression non-emptiness check
V: argval check
T: argval–argtyp compatibility check

3. Construct sentence $\mathcal{S}(pcd)$ as follows:

$$\mathcal{S}(pcd) = \mathcal{T}(pcd) \wedge \Big(\bigwedge_{S \subseteq P} c(S) \Big)$$

$$\begin{aligned} \text{where } \mathcal{T}(\texttt{and}\ pcd_1\ pcd_2) &= \mathcal{T}(pcd_1) \wedge \mathcal{T}(pcd_2) \\ \mathcal{T}(\texttt{or}\ pcd_1\ pcd_2) &= \mathcal{T}(pcd_1) \vee \mathcal{T}(pcd_2) \\ \mathcal{T}(\texttt{not}\ pcd_1) &= \neg\, \mathcal{T}(pcd_1) \\ \mathcal{T}(pcd) &= V(pcd) \text{ for } pcd \text{ primitive} \end{aligned}$$

Generating constraint terms $c(S)$ in Step 2 is the most difficult step in the reduction. Most of the necessary constraints can be generated by considering only pairs of pointcuts $S = \{p_1, p_2\}$ rather than larger sets. (The one exception involves regular expressions, and is described in greater detail below.) The possible cases for such pairs can be divided into the various possible syntactic forms for p_1 and p_2. These cases are summarized in Table 1. The columns and rows of the table are labeled with the four-letter prefixes of pointcut primitives listed in Figure 2.

Cells labeled I are always independent; no constraint term is required in these cases. For example, a `call` join point can potentially appear within any lexical scope, so `call` and `withincode` pointcuts are independent. Cells labeled E are mutually exclusive; the necessary constraint term is $\neg(\mathcal{T}(p_1) \wedge \mathcal{T}(p_2))$. For example, no join point is both a call instruction and a field-get instruction, so `call` and `get` are mutually exclusive.

Deriving appropriate constraints for cells labeled C requires a model of the trusted portion of the class hierarchy. Trusted classes typically consist of those implemented by the Java standard libraries or other system-level libraries. When one or both pointcut primitives name a known, trusted class, the constraint generator performs a subclass test over the trusted class model; otherwise the two pointcuts are independent. For example, two `argtyp` pointcuts (`argt`) are typically independent since in an arbitrary untrusted program any type could be a subtype of any other type. However, if one pointcut names a standard library class c that is declared final and the other uses a type pattern that does not match any superclass of c, then the two are mutually exclusive.

A regular expression non-emptiness test is required for cells marked R in Table 1, and is the only case that requires consideration of sets $S \subseteq P$ that are larger than size 2. For each set of pointcuts S that place regular expression constraints upon the same join point component (e.g., the same instruction argument), the constraint generator must decide whether there exists a string that satisfies all regular expressions in S and none in $P - S$. If so, no constraint is generated for S; otherwise a mutual exclusion constraint is generated. For example, consider the pointcut fragment

```
(and (call "x*") (call "xx*") (call "xy"))
```

which initially reduces to the sentence $a \wedge b \wedge c$ before constraints are added. Regular expressions `xx*` and `xy` are both subsets of `x*`, and `xy` is disjoint from `xx*`. The algorithm above therefore represents these with a conjunction of three constraints: $\neg(b \wedge \neg a) \wedge \neg(c \wedge \neg a) \wedge \neg(a \wedge b \wedge c)$.

The space of subsets $S \subseteq P$ that must be considered is potentially exponential in the size of P, but in practice the space can be significantly pruned through memoizing. That is, if any subset S has an empty intersection, then no supersets of S need be considered further. Thus, working upward from small subsets to larger ones tends to result in a smaller number of regular expression emptiness sub-problems that must be solved.

The cell of Table 1 marked V concerns the special case of two `argval` predicates. Two `argval` predicates are always independent unless they regard the same argument index n and both contain value predicates that are relevant to the same type of data (object, integer, or string). String predicates reduce to regular expression non-emptiness problems, described above. Integer predicates result in implication, bi-implication, or mutual exclusion constraints. For example, if p_1 contains `(inteq 3)` and p_2 contains `(intlt 4)`, then $c(\{p_1, p_2\}) = (\mathcal{T}(p_1) \Rightarrow \mathcal{T}(p_2))$.

Finally, the table cell marked T concerns the special case of an `argval` and an `argtyp` primitive. These are always independent if they refer to different runtime arguments, they are mutually exclusive if they refer to differing types, and otherwise the `argval` predicate implies the `argtyp` predicate. For example, if $p_1 =$ `(argval 1 (inteq 3))` and $p_2 =$ `(argtyp 1 int)` then we obtain the constraint $\mathcal{T}(p_1) \Rightarrow \mathcal{T}(p_2)$ because any join point satisfying p_1 also satisfies p_2.

Once boolean sentence $\mathcal{S}(pcd)$ has been constructed, it is delivered to a SAT-solving engine. The SAT-solver yields a satisfying assignment of boolean variables if one exists. From such an assignment it is trivial to recover a witness join point that lies in the intersection of the two original pointcuts. This facilitates disambiguation of the flawed policy specification by providing the human expert with an example target program fragment for which the policy is ambiguous. In the next section, we discuss several case studies in which our analysis algorithm discovered and reported undesired non-determinism in real policies.

4. CASE STUDIES

We implemented our non-determinism detection tool using a combination of Java and SWI-Prolog. The Java component consists of a SPoX parsing library (approximately 5000 lines of Java code) and an analysis engine that includes a Prolog-generating back-end (approximately 4200 lines of Java code). It extracts relevant information from the policy, including automaton transitions and pointcut expressions, and uses this to generate Prolog predicates that model the extracted policy information.

The Prolog half of the implementation is the heart of the analysis engine, and consists of dynamically generated code. It decides whether any pair of pointcut-labeled edges in the policy are non-deterministic using the algorithm described in Section 3. State variable non-determinism is decided through the use of a Prolog-based linear constraint solver. Pointcut non-determinism is decided by submitting the boolean sentence derived by algorithm $\mathcal{S}$ of Section 3.2 to a C implementation of the MiniSat SAT-solving engine [15]. If the sentence is satisfiable then the policy is non-deterministic, and the analysis tool identifies the conflicting portions of the specification.

In this section we discuss six policy scenarios in which our analysis tool discovered policy bugs through non-determinism detection. For each case study, we discuss the origins of the policy, the design flaw that led to unintended non-determinism, and how we removed the error. Runtime statistics for all experiments are summarized at the end of the section.

4.1 Filesystem API Protocols

An important class of software security policies are those that prescribe protocols for accessing system API's. For example, the JavaMOP documentation [18] includes a policy that prevents writing to a file that is not already open. Such a policy can be naturally encoded in SPoX as a 2-state automaton, where the *open* operation transitions the automaton from the *closed* to *opened* state, *write* operations are only permitted in the *opened* state, and the *close* operation transitions the automaton back to the *closed* state.

In practice such protocols can be significantly more complex. For example, a natural extension to the example above makes the set of acceptable operations contingent upon the mode in which the file has been opened. Files opened in read-mode may be read but not written, only those opened in random access mode may be seeked, etc. As the number of possible operations increases, policy complexity and the opportunity for error increase as well.

The `FileMode` policy specification in Figure 5 models a simplified filesystem API policy that supports read and write modes, *open* and *close* operations, and *read* and *write* operations. Lines 18 and 23 of the policy use the reserved # post-condition to cause the automaton to reject if a write or read operation is attempted in an incompatible mode. In addition, lines 31–34 prohibit all I/O operations when a file is in the closed state.

Line 32 of the policy contains a bug that has the unintended effect of rejecting even *open* operations when a file is in the *closed* state. The bug arises because the regular expression in that line is overly broad.

After submitting this policy to our analysis tool, it reported that the edge at line 31 conflicts with the edges at lines 3 and 9. Specifically, calls to `File.open` with a second argument of `"OpenRead"` or `"OpenWrite"` and a security state of $\mathtt{f} = 0$ solicit conflicting advice.

Upon discovering the bug, we disambiguated the policy by replacing the pointcut at lines 32–33 with the following refinement of the original pointcut expression.

```
(and (call "File.*)
     (not (call "File.open"))
     (target "x"))
```

The resulting policy passed the analysis and correctly enforced the desired policy.

```
 1 (state name="f" "File")
 2
 3 (edge name="openFileRead"
 4   (and (call "File.open")
 5        (argval 2 (streq "OpenRead"))
 6        (target "x"))
 7   (nodes "f" "x" 0,1))
 8
 9 (edge name="openFileWrite"
10   (and (call "File.open")
11        (argval 2 (streq "OpenWrite"))
12        (target "x"))
13   (nodes "f" "x" 0,2))
14
15 (edge name="illegalWrite"
16   (and (call "File.write")
17        (target "x"))
18   (nodes "f" "x" 1,#))
19
20 (edge name="illegalRead"
21   (and (call "File.read")
22        (target "x"))
23   (nodes "f" "x" 2,#))
24
25 (forall "i" from 1 to 2
26   (edge name="fileClose"
27     (and (call "File.close")
28          (target "x"))
29     (nodes "f" "x" i,0)))
30
31 (edge name="illegalFileOp"
32   (and (call "File.*")
33        (target "x"))
34   (nodes "f" "x" 0,#))
```

Figure 5: `FileMode` policy

4.2 Transaction Logging

A classic application of Aspect-Oriented Programming in the literature is transaction logging (e.g., [27]). An AOP-style transaction logger is one specific enforcement of a more general audit policy. The audit policy dictates that impending security-relevant transaction operations must be first logged via a trusted logging mechanism. An IRM or other security implementation can enforce the policy by injecting the necessary logging operations immediately before each transaction operation.

Figure 6 provides a fragment of one such audit policy for a hypothetical credit card processing library. The library includes a `CreditCardProcessor` class that contains numerous transaction implementations, all accessed via methods with names ending in `*Transaction`. The desired audit policy mandates exactly one call to the trusted `logTransaction` method before each such transaction.

The specification in Figure 6 contains bugs at lines 8 and 16 that mistakenly treat calls to the `logTransaction` method itself as transactions. Our analysis uncovered these bugs in the form of two sources of non-determinism: one associated with the edges at lines 3 and 15, and the other associated with those at lines 7 and 11. In the first case calls to `logTransaction` solicit conflicting advice in security state 0. In the second case the same ambiguity arises in state 1.

To correct the error, we introduced a named pointcut that lists each security-relevant transaction method explicitly in a large disjunctive pointcut. (An alternative would be to explicitly except the logging method using pointcut negation

```
(state name="logged")

(edge name="log"
  (call "CreditCardProcessor.logTransaction")
  (nodes "logged" 0,1))

(edge name="transaction"
  (call "CreditCardProcessor.*Transaction")
  (nodes "logged" 1,0))

(edge name="badLog"
  (call "CreditCardProcessor.logTransaction")
  (nodes "logged" 1,#))

(edge name="badTransaction"
  (call "CreditCardProcessor.*Transaction")
  (nodes "logged" 0,#))
```

Figure 6: Logger **policy**

```
1 (state name="exists" "File")
2
3 (forall "i" from 0 to 1
4   (edge name="rename"
5     (and (call "File.renameTo")
6          (target "x")
7          (argtyp 1 "y" "File"))
8     (nodes "x" "exists" 1,0)
9     (nodes "y" "exists" i,1)))
```

Figure 7: FileExists **policy**

and conjunction operators.) This eliminated the non-determinism and passed the analysis.

4.3 Object Aliasing

A particularly elusive form of unintentional non-determinism arises from object aliasing. As an illustration, consider the policy fragment in Figure 7, which tracks whether Java File objects refer to existent or non-existent files. When an existing file is renamed, the policy marks the source File object as referring to a non-existent file and the destination File object as referring to an existent one.

However, this policy has a subtle non-determinism bug that arises when the same object is passed as both the source and destination arguments of the File.renameTo method. In this case the policy stipulates that the File object transitions to both the *existent* and *non-existent* security states. A malicious program could exploit this loophole to transition File objects to incorrect security states and potentially circumvent the intended policy.

Our non-determinism analysis detected this bug in the form of a join point in which object identifiers x and y alias to a common File object. We corrected the error by changing the i in line 9 to the constant 0. This resolves the non-determinism by restricting only renameTo operations that change the name of an existing file to an unused filename; thus the source and destination cannot be the same object. The resulting policy was validated as deterministic.

4.4 Information Flow

A canonical information flow policy example in the IRM literature prevents untrusted programs from leaking confidential files over the network. One standard encoding of this policy prohibits all network send operations after a confidential file has been read [32]; however the resulting policy can be sometimes be too draconian to be useful in practice. A useful relaxation introduced in [2] permits subsequent network send operations only after the user has expressly permitted them via a trusted authentication and authorization mechanism.

```
 1 (state name="accessed")
 2 (state name="permission")
 3
 4 (edge name="authorize"
 5   (call "GUI.grantConnectPermission")
 6   (nodes "permission" 0,1))
 7
 8 (forall "i" from 0 to 1
 9   (edge name="read"
10     (and (call "File.open")
11          (argval 2 (streq "OpenRead")))
12     (nodes "accessed" i,1))
13   (edge name="send"
14     (call "Connection.open")
15     (nodes "permission" i,0)))
16
17 (edge name="badsend"
18   (call "Connection.open")
19   (nodes "accessed" 1,1)
20   (nodes "permission" 0,#))
```

Figure 8: GetPermission **policy**

We ported this latter policy specification to SPoX in Figure 8. Method GUI.grantConnectPermission has a trusted implementation that authorizes new connections. (When authorization is denied, the method does not return.) The policy maintains two security state variables: permission and accessed. The former is set to 1 (line 6) when the user authorizes a new connection and reset to 0 (line 15) after a new connection is established. Thus, future connections require subsequent authorizations. The accessed variable is set to 1 (line 12) when a confidential file is read. Unauthorized, new connections subsequent to such reads are prohibited by line 20, which signals a policy violation.

The policy specification contains a bug in the edge at line 13, which resets permission to 0 even when the new connection constitutes a policy violation. This results in a conflict between that edge and line 20 which signals the violation. In this case the unintended non-determinism therefore arises at the security state level rather than at the pointcut level.

To fix the problem, we split the edge at line 13 into two cases, one for new connections after confidential reads and another for new connections prior to any confidential reads:

```
(forall "i" from 0 to 1       (edge name="send2"
 (edge name="send1"             (call "Connection.open")
  (call "Connection.open")     (nodes "accessed" 1,1)
  (nodes "accessed" 0,0)       (nodes "permission" 1,0)))
  (nodes "permission" i,0)))
```

This repartitioning of the security state space intentionally omits the case where the send operation occurs after a confidential file-read and without authorization, since that case is adequately covered by the edge at line 17. The resulting policy is deterministic and therefore passed the analysis.

4.5 Free-riding Prevention

A recent case study [22] used SPoX to enforce a free-riding prevention policy for a peer-to-peer file-sharing application.

```
1 (state name="counter")
2
3 (forall "i" from -10000 to 2
4   (edge name="download"
5     (call "Connection.download")
6     (nodes "counter" i,i+1))
7   (edge name="upload"
8     (call "Connection.upload")
9     (nodes "counter" i,i-1)))
10
11 (edge name="illegalDownload"
12   (call "Connection.download")
13   (nodes "counter" 2,#))
```

Figure 9: NoFreeride policy

The policy permitted users to download at most 2 more files than had been uploaded by the application. This forces users of the network to share approximately as many files as they obtain. The policy is reproduced in Figure 9.

Surprisingly, this policy specification contains a non-determinism bug, which was uncovered by our analysis utility. Specifically, when counter `i` reaches 2 and another download operation occurs, the edges at lines 4 and 11 of the specification give conflicting advice. The design flaw can be traced to an off-by-one error in the bounds of the `forall` loop, which permit line 6 to increase the counter beyond 2 and line 9 to decrease the counter beyond -10000.

To correct the error, we reduced the upper bound in line 3 from 2 to 1 and changed the decrement operation in line 9 from `i,i-1` to `i+1,i`. This corrected the bug and yielded a deterministic policy.

4.6 Policy Composition

Automated non-determinism detection is also useful for composing related policies because it reveals join points for which the different policies conflict. This affords policy-writers an opportunity to decide how such conflicts should be resolved on a case-by-case basis.

For example, [13] discusses challenges related to merging policies that mandate logging and encryption of the same data in untrusted program operations. We adapted that example to SPoX in the form of a data encryption policy for the credit card transaction system discussed in Section 4.2. The encryption policy specification is shown in Figure 10.

Combining the policy in Figure 10 with the one in Figure 5 resulted in a composite policy that our analysis tool identified as non-deterministic. The non-determinism was witnessed by join points that both policies considered to be security-relevant, such as calls to `creditTransaction`.

In each case, we were able to resolve the unwanted non-determinism by prescribing an ordering (via an appropriate automaton encoding) on the policy-mandated security checks. For this example we adopted the strategy of requiring the encryption operations to be exhibited before the logging operations. Once all conflicts were resolved, the final composite policy passed the analysis and correctly enforced both policies.

```
(state name="encrypted")

(edge name="encrypt"
  (call "CreditCardProcessor.encryptTransaction")
  (nodes "encrypted" 0,1))

(edge name="transaction"
  (or (call "CreditCardProcessor.creditTransaction")
      (call "CreditCardProcessor.debitTransaction"))
  (nodes "encrypted" 1,0))

(edge name="badEncrypt"
  (call "CreditCardProcessor.encryptTransaction")
  (nodes "encrypted" 1,#))

(edge name="badTransaction"
  (or (call "CreditCardProcessor.creditTransaction")
      (call "CreditCardProcessor.debitTransaction"))
  (nodes "encrypted" 0,#))
```

Figure 10: Encrypt policy

4.7 Summary of Results

The results of our experiments are summarized in Table 2. All experiments were performed on an HP Pavilion notebook computer with 4 GB of RAM and an AMD Turion 64 X2 TL-62 mobile processor running at 2.10 GHz, using a 32-bit JVM and the 64-bit version of Windows 7. In the table headings, *size* refers to the number of characters in each policy specification (including spaces) and *pointcut vars* refers to the number of unique boolean variables introduced during the reduction to SAT described in Section 3.2. In one case (`FileExistsFixed`) no pointcut variables were generated because all potential non-determinism was eliminated after the first phase of the algorithm.

The MiniSat Prolog interface converts boolean sentences to conjunctive normal form (CNF) before processing. To indicate the size and complexity of the resulting SAT problems, we list the number of boolean variables and the number of clauses in the resulting CNF sentences in columns 4 and 5 of the table, respectively. Finally, the last column reports the average execution time in milliseconds that our analysis tool required for each policy including parsing, Prolog-generation, and both phases of the algorithm.

In general we found that runtimes and boolean sentence sizes were well within the range of practical use for the policies we tested. The only potential scaling issue we encountered concerns policies that contain large numbers of non-independent regular expressions containing wildcards. In those cases our treatment of regular expressions generates a large number of constraint terms, slowing the analysis. We believe this drawback could be overcome by making better use of the memoizing technique mentioned in Section 3.2 (which is not yet fully utilized by our prototype). An interesting and potentially more elegant alternative is to replace the SAT solver with an SMT solver equipped with a theory of regular languages. This is an avenue we intend to investigate in future work.

5. RELATED WORK

From its inception, aspect-oriented programming has been widely recognized as a promising technique for elegantly implementing cross-cutting security concerns at the source code level (cf., [36]). However, a major challenge faced by security practitioners involves the need to formally reason about aspects, which is difficult when the language includes an effectful, Turing-complete advice language. Past work has therefore proposed effect-free AOP languages to facilitate such formal reasoning [10]. AspectML [11] and SPoX

Table 2: Experimental Results

Policy	Size (chars)	Pointcut vars	CNF vars	CNF clauses	Runtimes
`FileMode`	1488	9	1764	2061	2243ms
`FileModeFixed`	1570	8	706	850	248ms
`Logger`	722	2	10	12	235ms
`LoggerFixed`	956	3	34	40	156ms
`GetPermission`	938	5	44	57	90ms
`GetPermissionFixed`	1189	5	49	61	85ms
`FileExists`	437	3	10	14	35ms
`FileExistsFixed`	423	0	0	0	25ms
`NoFreeride`	1024	3	22	28	797ms
`NoFreerideFixed`	1075	3	18	22	825ms
`Encrypt`	986	3	34	40	163ms
`Log&Encrypt`	2281	4	972	1180	538ms
`Log&EncryptFixed`	2321	4	578	703	391ms

[19] are purely functional and purely declarative (respectively) AOP languages that have emerged recently in response to this challenge.

A form of formal reasoning that has received particular attention in the AOP literature regards the problem of resolving potential conflicts between advice applied to shared join points [9]. One approach to this problem disambiguates such advice by considering and analyzing every possible ordering of advice insertions [1, 14]. Incremental testing [26] has been proposed as a means of modeling stateful effects caused by application of advice, which is potentially useful for detecting conflicts. Such analyses have also proved useful for elimination of unnecessary advice insertions [29]. However, each of these approaches regards disambiguation of advice applied to a specific, known target program. This limits their usefulness for analyzing generic, aspectual security policy specifications that are intended to be applicable to arbitrary untrusted target programs.

Another useful line of research involves extending AOP languages to include convenient mechanisms that allow the aspect programmer to eliminate unwanted non-determinism manually [29]. These are most valuable after potential non-determinism has been detected.

Pointcut independence without respect to a given target program (*strong independence* [12]) is considered by Palm et. al. [31] in the context of program componentization. Their model regards a restricted pointcut language that consists only of calls, control flow operators (`cflow`), and boolean operators. For this reduced language the pointcut satisfiability problem is shown to be NP-complete. In contrast, our work considers a richer language that includes stateful advice, AspectJ pointcut primitives [24], dynamic predicates over runtime values, and regular expressions. It is not clear that the aforementioned NP-completeness result applies to this richer language. For example, the SAT-reduction in Section 3 must potentially consider all subsets of the set of regular expressions in the policy during constraint-generation, and therefore generates boolean sentences that are worst-case exponential in size. Work on anti-patterns [25] has uncovered various other difficult challenges related to decision problems that include regular expressions and negation.

In-lined Reference Monitoring [32] is a general paradigm for shifting reference monitors traditionally implemented at the system level (e.g., within the operating system or virtual machine) into the untrusted code they monitor. This provides optimization opportunities not readily available to traditional system-level monitors, such as reduced context-switching overhead and various techniques for partially evaluating injected security guard code (e.g., [4, 6, 8]). The in-lining process is similar to aspect-weaving, which has led to an increasing convergence of IRM and AOP research.

JavaMOP [7] implements IRM's as AspectJ aspects. SPoX enforces policies expressible as security automata [3] and implements them by in-lining the automata into Java bytecode [22]. Related work has implemented IRM's for several other architectures, including the Microsoft .NET framework [20], ActionScript bytecode [33], and x86 assembly code [17]. The approach is also widely used for a variety of debugging purposes, such as for detecting race conditions [5].

Work on automatically certifying IRM implementations is ongoing, and includes type-checking [20], programming-by-contract [2], and model-checking [34]. Most IRM systems limit their attention to safety policies, since these are known to be precisely enforceable [21, 32]. However, recent work on edit automata [28] has extended IRM technology to include certain restricted classes of liveness policies as well.

6. CONCLUSION

Correct policy specification is recognized throughout the security literature as a notoriously difficult problem. Policies encoded as full aspects can include arbitrary imperative code, making them especially difficult to analyze and verify.

In this work we have demonstrated that restricting aspectual policy specifications to purely declarative (but stateful) advice, and strengthening the pointcut language to include declarative predicates over runtime values, improves the feasibility of aspectual policy analysis yet remains expressive enough to encode large classes of important security properties. Such policies can be effectively enforced as In-lined Reference Monitors that target Java bytecode binaries.

We have further observed that in such a policy language, many common specification errors manifest as policy non-determinism. The detection of such errors was achieved via a non-determinism analysis that reduces the problem to a combination of several well-known sub-problems including boolean satisfiability, linear programming, and regular language non-emptiness.

The effectiveness of the technique was demonstrated via the detection of policy specification bugs in a number of case studies. In each case the analysis reliably uncovered specification bugs in a way that facilitated understanding of the underlying policy design flaws and how to correct them. Policies were drawn from several existing aspect-oriented languages, indicating that the analysis is useful for debugging specifications for a variety of systems.

7. REFERENCES

[1] M. Aksit, A. Rensink, and T. Staijen. A graph-transformation-based simulation approach for analysing aspect interference on shared join points. In *Proc. of the ACM International Conference on Aspect-Oriented Software Development (AOSD)*, pages 39–50, Charlottesville, Virginia, March 2009.

[2] I. Aktug and K. Naliuka. ConSpec: A formal language for policy specification. In *Proc. of the 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.

[3] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1986.

[4] P. Avgustinov, J. Tibble, and O. de Moor. Making trace monitors feasible. In *Proc. of the International Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Montréal, Canada, 2007.

[5] E. Bodden and K. Havelund. Racer: Effective race detection using AspectJ. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 155–166, Seattle, Washington, 2008.

[6] E. Bodden, P. Lam, and L. Hendren. Finding programming errors earlier by evaluating runtime monitors ahead-of-time. In *Proc. of the ACM International Symposium on Foundations of Software Engineering (FSE)*, pages 36–47, Atlanta, Georgia, 2008.

[7] F. Chen and G. Roşu. Java-MOP: A Monitoring Oriented Programming environment for Java. In *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 546–550, Edinburgh, Scotland, April 2005.

[8] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 54–66, Boston, Massachusetts, 2000.

[9] C. A. Constantinides, A. Bader, and T. Elrad. Separation of concerns in concurrent software systems. In *Proc. of the International Workshop on Aspects and Dimensional Computing at ECOOP*, Cannes, France, 2000.

[10] D. S. Dantas and D. Walker. Harmless advice. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 383–396, Charleston, South Carolina, January 2006.

[11] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30(3), 2008.

[12] R. Douence, P. Fradet, and M. Südholt. A framework for the detection and resolution of aspect interactions. In *Proc. of the ACM Conference on Generative Programming and Component Engineering (GPCE)*, pages 173–188, October 2002.

[13] R. Douence, P. Fradet, and M. Südholt. Composition, reuse and interaction analysis of stateful aspects. In *Proc. of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 141–150, Lancaster, United Kingdom, March 2004.

[14] P. Durr, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *Proc. of the First Aspect, Dependencies, and Interactions Workshop (ADI)*, pages 10–18, Nantes, France, July 2006.

[15] N. Eén and N. Sörensson. MiniSat. `http://minisat.se/`, 2007.

[16] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join points. In *Proc. of the Foundations of Aspect-Oriented Languages Workshop (FOAL)*, pages 1–10, Bonn, Germany, 2006.

[17] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *Proc. of the New Security Paradigms Workshop (NSPW)*, pages 87–95, Caledon Hills, Canada, September 1999.

[18] Formal Systems Laboratory (FSL) of the Department of Computer Science at the University of Illinois at Urbana-Champaign (UIUC). JavaMOP 2.0 Finite State Machine (JavaFSM). `http://fsl.cs.uiuc.edu/index.php/Special:JavaFSMPlugin2.0`.

[19] K. W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors. In *Proc. of the ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 11–20, Tucson, Arizona, June 2008.

[20] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. of the ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 7–15, June 2006.

[21] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Transactions On Programming Languages And Systems (TOPLAS)*, 28(1):175–205, January 2006.

[22] M. Jones and K. W. Hamlen. Enforcing IRM security policies: Two case studies. In *Proc. of IEEE Intelligence and Security Informatics (ISI)*, pages 214–216, Dallas, Texas, June 2009.

[23] S. Kallel, A. Charfi, M. Mezini, M. Jmaiel, and K. Klose. From formal access control policies to runtime enforcement aspects. In *Proc. of the International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 16–31, Leuven, Belgium, 2009.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2072, pages 327–355, Budapest, Hungary, June 2001.

[25] C. Kirchner, R. Kopetz, and P.-E. Moreau. Anti-pattern matching. In *Proc. of the European Symposium on Programming (ESOP)*, pages 110–124, Braga, Portugal, March 2007.

[26] S. Krishnamurthi and K. Fisler. Foundations of incremental aspect model-checking. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 16(2), April 2007.

[27] R. Laddad. I want my AOP!, part 1. `http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html`, January 2002.

[28] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005.

[29] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proc. of Compiler Construction, volume 2622 of Springer Lecture Notes in Computer Science*, pages 46–60, Warsaw, Poland, April 2003.

[30] G. C. Necula. Proof-carrying code. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 106–119, Paris, France, January 1997.

[31] J. Palm, P. Wu, and K. Lieberherr. Understanding aspects through call graph enumeration and pointcut satisfiability. Technical Report NU-CCIS-04-01, College of Computer and Information Science, Northeastern University, Boston, Massachusetts, March 2004.

[32] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, February 2000.

[33] M. Sridhar and K. W. Hamlen. ActionScript in-lined reference monitoring in Prolog. In *Proc. of the International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 149–151, Madrid, Spain, January 2010.

[34] M. Sridhar and K. W. Hamlen. Model-checking in-lined reference monitors. In *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 312–327, Madrid, Spain, January 2010.

[35] The AspectJ Team. The AspectJ programming guide. `http://www.eclipse.org/aspectj/doc/released/progguide/index.html`, 2003.

[36] J. Viega, J. Bloch, and P. Chandra. Applying aspect-oriented programming to security. *Cutter IT Journal*, 14(2), February 2001.