Temporal Aspects as Security Automata

Peter Hui
Department of Computer Science
DePaul University
243 South Wabash
Chicago, IL 60604, USA

James Riely
Department of Computer Science
DePaul University
243 South Wabash
Chicago, IL 60604, USA

2005/10/14

Abstract

Aspect-oriented programming (AOP) has been touted as a promising paradigm for managing complex software-security concerns. Roughly, AOP allows the security-sensitive events in a system to be specified separately from core functionality. The events of interest are specified in a *pointcut*. When a pointcut triggers, control is redirected to *advice*, which intercepts the event, potentially redirecting it to an error handler.

Many interesting security properties are history-dependent; however, currently deployed pointcut languages cannot express history-sensitivity. (Mechanisms like `cflow` in AspectJ capture only the current call stack.) We present a language of pointcuts with past-time temporal operators and discuss their implementation using a variant of security automata. The main result is a proof that the implementation is correct.

AOP is usually presented as an extension of an underlying computational mechanism, but the extension is in no sense conservative. For program analysis, this is in some sense this is the worst of both worlds. We adopt a different approach: refining our earlier work, we define a minimal language of events and aspects in which "everything is an aspect". The minimalist approach serves to clarify the issues and may be of independent interest.

1 Introduction

Aspect-oriented programming (AOP) ([9]) is a relatively new programming paradigm designed to address crosscutting concerns across objects in the more traditional object-oriented paradigm. In this model, the programmer defines *aspects*, each consisting of a block of code (the *advice body*), and a *pointcut*, which states when the code is to be executed. Current implementations allow for the user to define pointcuts which trigger off of a specified atomic event, but facilities for triggering of a program's history typically limited to the current call stack (as in AspectJ's `cflow`).

AOP has some potential for specifying and enforcing security policies. However, many such policies are both *history-sensitive* and *dynamic* (likely to change at runtime).

In this paper, we define a syntax and operational semantics for *temporal aspects*, which allow for pointcuts to be defined temporally— that is, in terms of events which have happened in the past. An obvious solution is to record the entire program history, but such an implementation is clearly impractical for long-lived programs. In this vein, we present an equivalent, automaton-based syntax and operational semantics, which records only relevant events. The automaton state provides an abstraction of the history, and our main result demonstrates that this abstract view faithfully implements the original semantics.

The situation is complicated by two facts: First, a pointcut may cause an event to be intercepted *before* it occurs; this is typical of security policies that specify sequences which must be aborted, rather than those which are allowed. Second, new advice may arrive at runtime, dynamically modifying existing policies. In both cases a key difficulty is getting the semantics of the source language "right". Refining our previous work [5], we adopt a minimalist approach which lays bare the essence of the problem without a great deal of object-oriented cruft.

Our implementation language uses a variant of Schneider's security automata [14]. A *security automaton* enforces a security policy by monitoring the execution of a target system, and intercepting instructions which would otherwise violate the specified policy. For instance, a user may specify that subsequent to a *FileRead* operation, the user is forbidden from executing a *Send* operation. The corresponding automaton would monitor the target system, watching for instances of *FileRead*. If one was seen, the automaton would then monitor the system for an attempted *Send*, and if such an attempt were made, it would intercept the call and presumably execute some error handling code instead.

We proceed as follows: in Section 2, we review related work. In Section 3, we define Polyadic μABC, a minimal aspect-based calculus defining roles, advice, and non-temporal advised messages. In Section 4, we augment Polyadic μABC to include temporal pointcuts, specified using a subset of the regular expressions. In Section 5, we define an equivalent automaton-based semantics, and describe several security related examples. In Section 6, we demonstrate equivalence of the two semantics by providing a translation of a configuration in the history-based semantics to an equivalent configuration in the automaton-based semantics. Future work is discussed in Section 7.

2 Related Work

In [9], Kiczales, et. al. identified several problems for which common programming paradigms, namely object-oriented and procedural programming, could offer no known elegant solutions. They characterized these problems as *crosscutting* concerns, which cut across many facets of the system. They proposed a new paradigm, namely an *aspect oriented* approach to address these concerns. Current implementations such as AspectJ (`http://www.aspectj.org`) and Aspect# (`http://www.castleproject.org`) allow users to define pointcuts to trigger on atomic events, but with no regard for what the target system has done in the past. There have been several studies of the semantic foundations of AOP [20, 3, 12, 22, 17, 13, 7, 6, 11]. Recently, there has also been work on both the semantics and implementation of temporal pointcuts [15, 21, 10, 2];

however none of this work discusses dynamic aspects (ie, advice that can be loaded at runtime), which is central to our approach.

Security automata have been widely investigated as a means of implementing security policies, and in fact Schneider has shown that these automata can implement any *safety policy*, and that any safety policy can be implemented using such an automaton ([14]). In [19], Walker uses security automata to encode security policies to be enforced in automatically generated code. In [18], Erlingsson and Schneider use security automata to implement software fault isolation security policies, which prevent memory accesses outside of the allowable address space. In that work, they discuss techniques used to merge security automata directly into binary code at the x86 assembler and Java Virtual Machine Language (JVML) level. In [4], Barker and Stuckey investigate role based and temporal role based access control policies, implemented using constraint logic specifications. In [16], Thiemann incorporates security automata into an interpreter for a simply typed call-by-value lambda calculus, which he then translates to an equivalent two-level lambda calculus, upon which type specialization removes all run-time operations involving security state.

The limitations of stack-based security policies are explored in [8]; history-based solutions are presented in [1]. Our work can be used as an alternative implementation technique for the ideas in the later paper.

3 Polyadic μABC

NOTATION. For any metavariable X, we write $\bar{X}$ for an ordered sequence of X's.

We define a polyadic variant of μABC, introduced in [5]. The earlier paper followed the style of object-oriented languages; each message "$p \rightarrow q : \ell$" had a source p, a destination q, and a name ℓ. Such a message is *triadic* in that its meaning depends on a triple of names, or *roles*. Here we generalize triadic messages to polyadic *events*, $\langle p_1, \ldots, p_n \rangle$ (equivalently $\langle \bar{p} \rangle$), with triadic messages as a special case "$\langle p, q, \ell \rangle$".

For simplicity, we look at a single-threaded variant. At each moment in runtime, there is a single event $\langle \bar{p} \rangle$ under consideration. Execution is determined by *advice* that triggers on the event. At any given moment, the current event is decorated with a vector of advice $\bar{a}$, which is waiting to process the event. Thus $a_1, \ldots, a_n \langle \bar{p} \rangle$ indicates that advices named a_i are waiting to process event $\bar{p}$. We say that a_i *advises* $\bar{p}$, and that $\bar{a}\langle \bar{p} \rangle$ is an *advised* event. For consistency with the precedence of declarations, we read the advice list from right to left; thus a_n is the first advice to process the event.

The special advice call initiates advice lookup. When $\mathsf{call}\langle \bar{p} \rangle$ executes, all the advice triggering on $\langle \bar{p} \rangle$ is listed, resulting in a new execution state: $\bar{a}\langle \bar{p} \rangle$. To determine whether an advice is triggered, we use the pointcut α. Pointcuts may be defined to trigger on an exact role, or a set of roles. We facilitate the specification of such sets using a role preorder, with maximal element top.

An advice body $\mathsf{adv}\ a[\alpha] = u(\bar{x})\, N$ is parameterized both on the event $\bar{x}$ and the remaining advice u. Following the terminology of around advice in AspectJ, we referee u as the *proceed* variable.

3.1 Syntax and Evaluation

We give the syntax and evaluation semantics of the language parametrically with respect to pointcuts α and pointcut satisfaction $\bar{D} \vdash \langle\bar{p}\rangle$ sat α, described in the next subsection. Note that terms have the form $\bar{D}; \bar{a}\langle\bar{p}\rangle$; ie, a term is a list of declarations followed by a single advised event. We refer to $\bar{p}$ as the *current event*, $\bar{a}$ as the *current advice list*, and a_n as the *current advice* ($\bar{a} = a_1, \ldots, a_n$).

TERM SYNTAX

a-z	Names; call, commit, top and _ are reserved
$D, E ::=$	Declarations
role $p < q$	Declare Role; $dn(\text{role } p) = p$
adv $a[\alpha] = u(\bar{x})\, N$	Declare Advice; $dn(\text{adv } a) = a$; u and $\bar{x}$ bound in N
$L, M, N ::=$	Terms
$D; M$	Declaration; $dn(D)$ bound in M
$\bar{a}\langle\bar{p}\rangle$	Advised Message

NOTATION. We write $dn(D)$ for the declared name of D. Reserved names may not be declared. We identify syntax up to renaming of bound names. For any syntactic category with typical element $\mathcal{E}$, we write $fn(\mathcal{E})$ for the set of free names occurring in $\mathcal{E}$. We write $\mathcal{E}\{^a/_x\}$ for the capture avoiding substitution of a for x in $\mathcal{E}$. We write $\mathcal{E}\{^{\bar{a}}/_{\bar{x}}\}$ for $\mathcal{E}\{^{a_1}/_{x_1}, \ldots, ^{a_n}/_{x_n}\}$; note that $\mathcal{E}\{^{\bar{a}}/_{\bar{x}}\}$ is defined only if $\bar{x}$ and $\bar{a}$ have the same length.

CONVENTION. To improve readability, we use the following discipline for names:

- *a–e* are advice names (including the reserved names call, beg, end);
- *f–t* are role names (including the reserved name top);
- *u–w* are advice names that are bound in the body of an advice declaration;
- *x–z* are role names that are bound in the body of an advice declaration;
- _ is a reserved name used to bind a name that is not of interest — that is, does not occur free in any subterm.

We drop syntactic elements that are not of interest. Consider the declaration "adv $a[\alpha] = u(\bar{x})\, N$"; we may elide the name "adv$[\alpha] = u(\bar{x})\, N$", or the pointcut "adv $a = u(\bar{x})\, N$", or the body "adv $a[\alpha]$", or both the pointcut and the body "adv a".

Evaluation of is defined using configurations which consist of a vector of declarations and a term. By EVAL-DEC, declarations are recorded in the configuration whenever they are encountered in a term. By EVAL-CALL, if an event $\langle\bar{p}\rangle$ is being processed with first advice call, then the advice list $\bar{a}$ is calculated, consisting of the advice names a_i such that the pointcut declared with a_i is satisfied by $\langle\bar{p}\rangle$. By EVAL-ADV, if an event $\langle\bar{p}\rangle$ is being processed with first advice a, then the body of a is executed; the advice body is parameterized by both the event $\langle\bar{p}\rangle$ and remaining advice $\bar{b}$.

EVALUATION $(\bar{D} \rhd M \rightarrow \bar{E} \rhd N)$

(EVAL-DEC)
$$\frac{}{\bar{D} \rhd E\,;M \rightarrow \bar{D},E \rhd M}$$

(EVAL-CALL)
$$\frac{[\bar{a}] = \left[a \;\middle|\; \begin{array}{l} \bar{D} \ni \mathsf{adv}\; a[\alpha] \\ \bar{D} \vdash \langle \bar{p} \rangle \;\mathsf{sat}\; \alpha \end{array} \right]}{\bar{D} \rhd \bar{b}, \mathsf{call}\langle \bar{p} \rangle \rightarrow \bar{D} \rhd \bar{b}, \bar{a}\langle \bar{p} \rangle}$$

(EVAL-ADV)
$$\frac{\bar{D} \ni \mathsf{adv}\; a = u(\bar{x})\; N}{\bar{D} \rhd \bar{b}, a\langle \bar{p} \rangle \rightarrow \bar{D} \rhd N\{\bar{b}/u, \bar{p}/\bar{x}\}}$$

3.2 Atomic Event Pointcuts

We now consider a simple boolean logic over events. We allow event sets to be specified using role patterns which include subroles and "varargs", ie, optional roles.[1]

POINTCUT SYNTAX

P,Q ::=	Role Pattern
p	Exact Role
$+p$	Sub Role
α,β ::=	Atomic Event Pointcut
$\langle \bar{P} \rangle$	Call Event
$\langle \bar{P}, * \rangle$	Call Event, varargs
$\alpha \vee \beta$	Disjunction
$\neg\alpha$	Negation
σ,ρ ::= $\langle \bar{p} \rangle$	Atomic Event

Define 1 as $\langle * \rangle$; define 0 as $\neg 1$; and define $\alpha \wedge \beta$ as $\neg(\neg\alpha \vee \neg\beta)$. We write $\bar{D} \vdash r \leqslant p$ for the obvious preorder generated from the role declaration order.

ATOMIC POINTCUT SATISFACTION $(\bar{D} \vdash \sigma \;\mathsf{sat}\; \alpha)$ (Obvious rules for or/not)

(SAT-CALL-ANY)
$$\frac{}{\bar{D} \vdash \langle \bar{p} \rangle \;\mathsf{sat}\; \langle * \rangle}$$

(SAT-CALL-EMPTY)
$$\frac{}{\bar{D} \vdash \langle\rangle \;\mathsf{sat}\; \langle\rangle}$$

(SAT-CALL-EXACT)
$$\frac{\bar{D} \vdash \langle \bar{q} \rangle \;\mathsf{sat}\; \langle \bar{Q} \rangle}{\bar{D} \vdash \langle r, \bar{q} \rangle \;\mathsf{sat}\; \langle r, \bar{Q} \rangle}$$

(SAT-CALL-SUB)
$$\frac{\bar{D} \vdash \langle \bar{q} \rangle \;\mathsf{sat}\; \langle \bar{Q} \rangle \quad \bar{D} \vdash r \leqslant p}{\bar{D} \vdash \langle r, \bar{q} \rangle \;\mathsf{sat}\; \langle +p, \bar{Q} \rangle}$$

4 Temporal Pointcuts

We extend μABC with temporal pointcuts. To do this, we modify the language of advice to include a temporal formula ϕ in addition to the atomic formula α. Intuitively, the pointcut fires when ϕ matches the past and α matches the current event.

In an aspect language, the ontology of events is complicated by the fact that events can be diverted; that is, an event can trigger advice that intercepts the event *before* it occurs, potentially causing the event to abort. This is particularly common in applications

[1]In the full version we also allow vararg parameters in advice declarations, ie $\mathsf{adv}\; a[\alpha] = u(\bar{x}, *)\; N$.

to security, where pointcuts often specify dangerous event sequences that interrupt normal processing. To indicate that an event is to be recorded in the history, we include the special advice commit.

Thus when the past is considered in firing a pointcut, we require that advice specify both the past ϕ and the potential future α. The past is specified as a regular expression over atomic event pointcuts; the potential future is specified as an atomic event pointcut.

SYNTAX

$D, E ::= \cdots$	Declarations
$\quad \mathsf{adv}\; a[\phi\,\alpha] = u(\bar{x})\, N$	Declare Advice
$\phi, \psi, \chi ::=$	Temporal Pointcuts
$\quad \alpha$	Atomic Event Pointcut
$\quad \varepsilon$	Empty Sequence
$\quad \phi\psi$	Sequence
$\quad \phi^*$	Kleene Star
$\quad \phi + \psi$	Disjunction
$\sigma, \rho ::= \langle \bar{p} \rangle$	Atomic Events

The semantics of temporal formulas $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$ is defined in the standard way (recalled in Appendix A) over strings of events, building on the semantics of atomic events ($\bar{D} \vdash \sigma\ \mathsf{sat}\ \alpha$). Note that the regular expression $\emptyset$ is represented here as the atomic event pointcut 0. We define the language of the formula as follows: $\mathcal{L}_H(\bar{D}, \phi) = \{\bar{\sigma} \mid \bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi\}$.

We now give the evaluation semantics for the language with temporal advice. We augment the semantics to record an execution history. We write $|\bar{\sigma}|$ for the length of string $\bar{\sigma}$. We define $\alpha^n \triangleq \alpha\alpha^{n-1}$, where $\alpha^0 \triangleq \varepsilon$. We write "$\mathsf{adv}\; a[\alpha] = u(\bar{x})\, N$" as shorthand for "$\mathsf{adv}\; a[1^*\,\alpha] = u(\bar{x})\, N$".

EVALUATION $(\bar{\sigma}; \bar{D} \rhd M \longrightarrow \bar{\rho}; \bar{E} \rhd N)$

(EVAL-DEC-ROLE)

$$\frac{}{\begin{array}{l}\quad \bar{\sigma}; \bar{D} \rhd \mathsf{role}\, p < q\, ; M \\ \longrightarrow \bar{\sigma}; \bar{D}, \mathsf{role}\, p < q \rhd M\end{array}}$$

(EVAL-DEC-ADV)

$$\frac{}{\begin{array}{l}\quad \bar{\sigma}; \bar{D} \rhd \mathsf{adv}\; a[\phi\,\alpha] = u(\bar{x})\, N\, ; M \\ \longrightarrow \bar{\sigma}; \bar{D}, \mathsf{adv}\; a[1^{|\bar{\sigma}|}\,\phi\,\alpha] = u(\bar{x})\, N \rhd M\end{array}}$$

(EVAL-COMMIT)

$$\frac{}{\bar{\sigma}; \bar{D} \rhd \bar{b}, \mathtt{commit}\langle \bar{p} \rangle \longrightarrow \bar{\sigma}, \langle \bar{p} \rangle; \bar{D} \rhd \bar{b}}$$

(EVAL-CALL)

$$\frac{[\bar{a}] = \left[a \;\middle|\; \begin{array}{l} \bar{D} \ni \mathsf{adv}\; a[\phi\,\alpha] \\ \bar{D} \vdash \bar{\sigma}, \langle \bar{p} \rangle\ \mathsf{sat}\ \phi\,\alpha \end{array} \right]}{\bar{\sigma}; \bar{D} \rhd \bar{b}, \mathsf{call}\langle \bar{p} \rangle \longrightarrow \bar{\sigma}; \bar{D} \rhd \bar{b}, \bar{a}\langle \bar{p} \rangle}$$

(EVAL-ADV)

$$\frac{\bar{D} \ni \mathsf{adv}\; a = u(\bar{x})\, N}{\bar{\sigma}; \bar{D} \rhd \bar{b}, a\langle \bar{p} \rangle \longrightarrow \bar{\sigma}; \bar{D} \rhd N\{{}^{\bar{b}}\!/\!_{u}, {}^{\bar{p}}\!/\!_{\bar{x}}\}}$$

EVAL-COMMIT causes an event to be recorded in the history. The original EVAL-DEC is split into different cases for roles and advice. EVAL-DEC-ROLE, EVAL-CALL, and EVAL-ADV are largely unchanged from the non-temporal semantics. Note only that in

EVAL-CALL the history is used, along with the current event, to determine whether an advice fires.

Of particular note is the rule EVAL-DEC-ADV, which takes a newly declared advice, and prepends a string of 1s to the temporal pointcut prior to adding it to the list of declarations. The purpose of doing so is to ensure that the advice only triggers on the event α *from the point of declaration onwards*, as opposed to some event that has already occurred in the past.

5 Automaton

In this section, we define an equivalent automaton-based semantics specified using temporal pointcuts. The automaton is a modified security automaton, in which each state has a possibly empty set of advice associated with it. When a $\texttt{call}\langle\bar{p}\rangle$ is executed, the current state's set of advice is scanned for any advice whose pointcut is satisfied by the event $\langle\bar{p}\rangle$. If a matching advice is found, the `call` is intercepted, and the advice's body is executed instead.

The automaton initially starts out with a single state with no advice associated with it:

$$\mathcal{A}_\emptyset : \begin{tikzcd} \boxed{\omega} \arrow[loop right, "1"] \end{tikzcd} \qquad \omega \triangleq 1^*$$

The automaton is dynamically augmented with each newly declared piece of advice. We provide an example to illustrate.

EXAMPLE 1. Consider a security policy which prohibits *Send* operations after a *FileRead* has been executed ([14]). In an aspect-oriented world, one might implement such a policy by declaring an advice $\mathsf{adv}\ a[\phi\,\alpha] = u\,(\bar{x})\ M$ where M is the error handling code, $\phi \triangleq (\neg \mathit{FileRead})^*\ \mathit{FileRead}\ (\neg \mathit{Send})^*$, and $\alpha \triangleq \mathit{Send}$.

Upon declaring this piece of advice, and assuming that no prior advice had been declared, the program automaton becomes:

$$\phi_0 \triangleq (\neg\texttt{call}\langle \mathit{FileRead}\rangle)^*\ \texttt{call}\langle \mathit{FileRead}\rangle\ (\neg\texttt{call}\langle \mathit{Send}\rangle)^*$$
$$\phi_1 \triangleq (\neg\texttt{call}\langle \mathit{Send}\rangle)^*$$

$$\mathcal{A}_{RS} : \begin{tikzcd} \phi_0 \arrow[loop above, "\neg\texttt{call}\langle \mathit{FileRead}\rangle"] \arrow[rr, "\texttt{call}\langle \mathit{FileRead}\rangle"] & & \phi_1, \langle a\rangle \arrow[loop right, "\neg\texttt{call}\langle \mathit{Send}\rangle"] \end{tikzcd}$$

This automaton begins in state ϕ_0, and monitors the target system for a $\texttt{call}\langle \mathit{FileRead}\rangle$. If one is seen, it moves to state ϕ_1. Since this state has $\mathsf{adv}\ a[\phi\,\alpha]$ associated with it, it begins to monitor the system for a's atomic event, $\texttt{call}\langle \mathit{Send}\rangle$. If such an attempt is made, it intercepts the call and executes advice a's error handler instead.

EXAMPLE 2. Consider a Chinese Wall policy, in which we have two systems A and B, such that if a user logs into A, we forbid the user from subsequently logging into A, and vice versa. To implement one direction, we could declare an advice $\mathsf{adv}\ b[\psi\,\beta] = u\,(\bar{x})\ N$, where N is the error handling code, $\psi \triangleq 1^*\texttt{call}\langle \mathit{Login}, A\rangle(\neg\texttt{call}\langle \mathit{Login}, B\rangle)^*$, and $\beta \triangleq \texttt{call}\langle \mathit{Login}, B\rangle$, The automaton for adv b, the first half of our Chinese wall policy becomes:

$$\begin{tikzcd}
\mathcal{A}_{CW_0} : & \langle\psi_0\rangle \arrow[loop above, "1^*"] \arrow[r, "\texttt{call}\langle \mathit{Login},\ A\rangle"] & \langle\psi_1\rangle, \langle b\rangle \arrow[loop above, "\neg\texttt{call}\langle \mathit{Login},\ B\rangle"]
\end{tikzcd}$$

$$\psi_0 \triangleq 1^*\texttt{call}\langle \mathit{Login},\ A\rangle(\neg\texttt{call}\langle \mathit{Login},\ B\rangle)^*$$
$$\psi_1 \triangleq (\neg\texttt{call}\langle \mathit{Login},\ B\rangle)^*$$

This automaton begins in state ψ_0, and monitors the target system for a $\texttt{call}\langle \mathit{Login},\ A\rangle$. If one is seen, it moves to state ψ_1. Since this state has advice $\langle b\rangle$ associated with it, it begins to monitor the system for b's atomic event, $\texttt{call}\langle \mathit{Login},\ B\rangle$. If such an attempt is made, it intercepts the call and executes advice b's error handler instead.

The automaton for advice c, the second part of our Chinese wall policy, is given similarly:

$$\begin{tikzcd}
\mathcal{A}_{CW_1} : & \langle\chi_0\rangle \arrow[loop above, "1^*"] \arrow[r, "\texttt{call}\langle \mathit{Login},\ B\rangle"] & \langle\chi_1\rangle, \langle c\rangle \arrow[loop above, "\neg\texttt{call}\langle \mathit{Login},\ A\rangle"]
\end{tikzcd}$$

$$\chi_0 \triangleq 1^*\texttt{call}\langle \mathit{Login},\ B\rangle(\neg\texttt{call}\langle \mathit{Login},\ A\rangle)^*$$
$$\chi_1 \triangleq (\neg\texttt{call}\langle \mathit{Login},\ A\rangle)^*$$

By computing the product of the above two automata using the standard product construction, we arrive at an automaton which implements our desired Chinese wall policy:

$$\mathcal{A}_{CW_0} \times \mathcal{A}_{CW_1} = \mathcal{A}_{CW} :$$

$$\begin{tikzcd}
\langle\psi_1,\chi_0\rangle, \langle b\rangle & \langle\psi_0,\chi_0\rangle \arrow[l, "\texttt{call}\langle \mathit{Login},\ A\rangle"'] \arrow[r, "\texttt{call}\langle \mathit{Login},\ B\rangle"] & \langle\psi_0,\chi_1\rangle, \langle c\rangle
\end{tikzcd}$$

5.1 Syntax and Evaluation

We define syntax and operational semantics for our automata. The transition alphabet ranges over the atomic event pointcuts α. The states are sets of temporal pointcuts ϕ, and associated with each state is a possibly empty set of advice names. While our modified security automata are essentially NFAs, there is one key difference— in lieu of accepting states, we instead have *advice* states, in which we associate with the state a set of advice names. On a `call`, we search the current state's set of advice names for an advice whose atomic pointcut matches the `call`. If a matching advice is found, the `call` is intercepted and the advice's body executed instead.

Transitions between states are taken on `commits`. The advice states ($\phi\checkmark$) and transition function ($\phi \xrightarrow{\alpha} \psi$) are defined below. To make the presentation more readable, we elide and advice when a state has none associated with it. That is, we write the state "$\phi;\emptyset$" simply as ϕ.

ADVICE STATES $(\phi\checkmark)$

$$\frac{}{\varepsilon\checkmark} \qquad \frac{\phi\checkmark \quad \psi\checkmark}{\phi\psi\checkmark} \qquad \frac{\phi\checkmark}{\phi+\psi\checkmark} \qquad \frac{\psi\checkmark}{\phi+\psi\checkmark} \qquad \frac{}{\phi^*\checkmark}$$

TRANSITION RELATION $(\phi \xrightarrow{\alpha} \psi)$

$$\frac{}{\alpha \xrightarrow{\alpha} \varepsilon} \qquad \frac{\phi \xrightarrow{\alpha} \phi'}{\phi\psi \xrightarrow{\alpha} \phi'\psi} \qquad \frac{\phi\checkmark \quad \psi \xrightarrow{\alpha} \psi'}{\phi\psi \xrightarrow{\alpha} \psi'} \qquad \frac{\phi \xrightarrow{\alpha} \phi'}{\phi^* \xrightarrow{\alpha} \phi'\phi^*}$$

$$\frac{\phi \xrightarrow{\alpha} \phi'}{\phi+\psi \xrightarrow{\alpha} \phi'} \qquad \frac{\psi \xrightarrow{\alpha} \psi'}{\phi+\psi \xrightarrow{\alpha} \psi'}$$

We write $\overset{\phi}{\Longrightarrow}$ for the reflexive transitive closure of $\xrightarrow{\alpha}$. We can modulate the transition relation from atomic event pointcuts to atomic events: define $\bar{D} \vdash \phi \xrightarrow{\sigma} \phi'$ if $\phi \xrightarrow{\alpha} \phi$' and $\bar{D} \vdash \sigma$ sat α. Further we can lift the definition to automaton states: $\bar{D} \vdash \phi_1,\ldots,\phi_n \overset{\sigma}{\Longrightarrow} \psi_1,\ldots,\psi_n$ if $\bar{D} \vdash \phi_i \xrightarrow{\sigma} \psi_i$ for all i between 1 and n. Finally we lift the resulting relation ($\bar{D} \vdash \Phi \overset{\sigma}{\Longrightarrow} \Psi$) to event sequences: $D \vdash \Phi_0 \xRightarrow{\sigma_1,\ldots,\sigma_n} \Phi_n$ if $\bar{D} \vdash \Phi_{i-1} \overset{\sigma_i}{\Longrightarrow} \Phi_i$ for all i between 1 and n.

Next, we formally state how to derive an automaton from an advice adv $a[\phi\alpha]$:

DEFINITION 3. For any temporal formula ϕ and advice a, let the automaton $\iota(\phi,a)$ *induced by* ϕ and a be the security automaton with states and transitions as defined by the transition relation given above, with start state ϕ, and advice a associated with each advice state.

We now define a syntax for our modified security automaton. A state in our automaton consists of a set of temporal pointcuts. For instance, in Example 2, automaton $\mathcal{A}_{CW}$ has three states: $\langle\psi_1,\chi_0\rangle, \langle\psi_0,\chi_0\rangle$, and $\langle\psi_0,\chi_1\rangle$. We associate with each state a possibly empty set of advice names. An automaton is a set of states along with their associated advice sets. The transitions between states are not explicitly specified, since the transitions can be derived by examining the temporal pointcuts within the states.

AUTOMATON SYNTAX

$\mathcal{A} ::= \langle\Phi,\bar{a}\rangle \mid \langle\Phi,\bar{a}\rangle,\mathcal{A}$	Automaton — sequence of $\langle$state, advice set$\rangle$ pairs
$\Phi,\Psi ::= \phi \mid \phi,\Phi$	State — set of temporal pointcuts

We define the product of two automata using the standard product construction. In assigning sets of advice to the states in the product automaton, we take the set union of each component state's associated advice names:

DEFINITION 4. For any two automata A,B,

$$A \times B = \{\langle\Psi_A,\Psi_B;\bar{a},\bar{b}\rangle \mid \langle\Psi_A,\bar{a}\rangle \in A, \langle\Psi_B,\bar{b}\rangle \in B\}$$

Next, we show how to merge an advice adv $a[\phi\alpha]$ with an existing automaton $\mathcal{A}$. Essentially, we construct the automaton for the advice, and create the product automaton:

$$\nu(\mathcal{A},\phi,a) \triangleq \mathcal{A} \times \iota(\phi,a)$$

We now give an automaton-based evaluation semantics to our language. Whereas previously we recorded the entire program history, we now instead maintain an automaton and state, which records only events of interest.

EVALUATION $(\mathcal{A};\Phi;\bar{D} \rhd M \rightarrow \mathcal{A}';\Psi;\bar{D}' \rhd M')$ (EVAL-DEC-ROLE AND EVAL-ADV AS BEFORE)

(EVAL-COMMIT)

$$\frac{\bar{D} \vdash \Phi \overset{\bar{p}}{\Longrightarrow} \Psi}{\mathcal{A};\Phi;\bar{D} \rhd \bar{b}, \texttt{commit}\langle\bar{p}\rangle \rightarrow \mathcal{A};\Psi;\bar{D} \rhd \bar{b}}$$

(EVAL-DEC-ADV)

$$\frac{}{\mathcal{A};\Phi;\bar{D} \rhd (\mathsf{adv}\; a[\phi\alpha] = u(\bar{x})\, N; M) \rightarrow \nu(\mathcal{A},\phi,a);\langle\Phi,\phi\rangle;\bar{D},(\mathsf{adv}\; a[\alpha] = u(\bar{x})\, N) \rhd M}$$

(EVAL-CALL)

$$\frac{[\bar{a}] = \left[a \;\middle|\; \begin{array}{l} \langle\Phi,\bar{b}\rangle \in \mathcal{A} \quad a \in \bar{b} \\ \bar{D} \ni \mathsf{adv}\; a[\alpha] \\ \bar{D} \vdash \langle\bar{p}\rangle \;\mathsf{sat}\; \alpha \end{array} \right]}{\mathcal{A};\Phi;\bar{D} \rhd \bar{b}, \mathsf{call}\langle\bar{p}\rangle \rightarrow \mathcal{A};\Phi;\bar{D} \rhd \bar{b}, \bar{a}\langle\bar{p}\rangle}$$

Operationally, EVAL-DEC-ROLE and EVAL-ADV act the same as in the history-based semantics. EVAL-DEC-ADV takes a new advice, merges it into the automaton using the ν operator, updates the current state, and adds the advice to the list of declarations. EVAL-CALL looks through the list of advices attached to the current state for one whose atomic pointcut matches the role vector $\bar{p}$ being called. If a matching advice is found, then the $\texttt{call}\langle\bar{p}\rangle$ is replaced with the advice body. EVAL-COMMIT simply updates the state of the automaton.

6 Equivalence

In this section, we demonstrate equivalence of the history-based semantics provided in Section 4 with the automaton-based semantics provided in Section 5 by providing a translation from a configuration in the former to an equivalent one in the latter. We conclude by showing that evaluation preserves the translation.

Intuitively, we translate a history-based configuration $\langle\bar{\sigma},\bar{D}\rangle$ to an automaton-based configuration $\langle\mathcal{A},\Phi,\bar{E}\rangle$ as follows: given a history $\bar{\sigma}$ and a set of declarations $\bar{D}$, we first construct an intermediate automaton $\mathcal{A}'$ which essentially simulates the automata induced by each advice declared in $\bar{D}$. We then compute the state Φ by simulating the history $\bar{\sigma}$ on $\mathcal{A}'$. Finally, we convert the intermediate automaton $\mathcal{A}'$ to the final automaton $\mathcal{A}$ by pruning intermediate states.

Recall the manner in which EVAL-DEC-ADV is defined in the history-based semantics: whenever an advice is declared, the current "timestamp" is explicitly noted in the form of a string of '1's prepended to the temporal pointcut. Thus if an advice adv $a[\phi\alpha]$ is declared at time n, the resulting automaton will have a string of n "placeholder" states $\pi_1,...,\pi_n$, where $\pi_i \xrightarrow{1} \pi_{i+1}$ for i between 1 and $n-1$, and $\pi_n \xrightarrow{1} \phi$, as shown below:

$$\begin{tikzcd}
\pi_1 \arrow[r, "1"] & \pi_2 \arrow[r, "1"] & \cdots \arrow[r, "1"] & \pi_n \arrow[r, "1"] & \phi \arrow[ur, dotted] \arrow[dr, dotted] & \\
& & & & &
\end{tikzcd}$$

CONVENTION. In constructing an automaton for an advice adv $a[1^n\phi\alpha]$ declared at time n, we label the states used as placeholders for time 1 through n as $\pi_1,...,\pi_n$, and we refer to these as π-states.

Strictly speaking, we must account for the fact that for an advice adv $a[\phi\alpha]$, ϕ may in fact begin with a string of leading 1s. We can easily get around this by syntactically differentiating between those 1s implicitly inserted by EVAL-DEC-ADV as a timestamp, and those explicitly specified by the user. In the interest of simplifying the presentation, we choose not to do so here.

When we construct the product of two automata, if a state $\Phi = \langle \phi_i, \psi_i, ...\chi_i \rangle$ in the resulting automaton is such that none of $\phi_i, \psi_i, ...\chi_i$ are π-states, we say that Φ is π-free.

We will often need to project the π-free states of an automaton, so we formally define this operation:

$$\mathcal{P}_{\not\pi}(A) = \{\langle \Phi, \bar{a} \rangle \in \mathcal{A} | \Phi \text{ contains no } \pi \text{ states}\}$$

LEMMA 5. *For two automata $\mathcal{A}$ and $\mathcal{B}$, $\mathcal{P}_{\not\pi}(\mathcal{A} \times \mathcal{B}) = \mathcal{P}_{\not\pi}(\mathcal{A}) \times \mathcal{P}_{\not\pi}(\mathcal{B})$*

Proof. Immediate. □

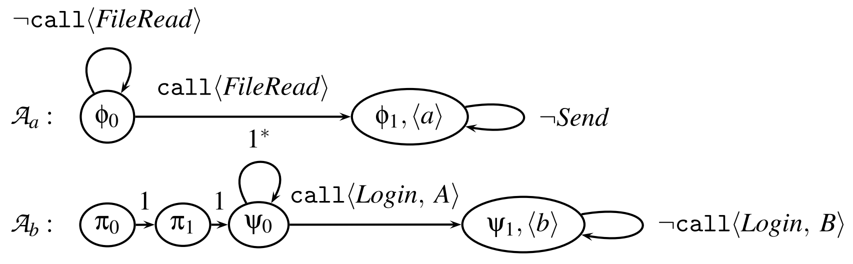
To construct $\mathcal{A}'$, we take the product of the automata induced by each advice in $\bar{D}$. Observe that as the product automaton, $\mathcal{A}'$ simulates each advice's automaton. As such, when the history $\bar{\sigma}$ is simulated on $\mathcal{A}'$, each advice's π-states serve as a placeholder, delaying monitoring of the pointcut until the point in the program's execution at which the advice was declared. When the simulation completes, we will have computed the current state in $\mathcal{A}'$, which is guaranteed to be π-free. We then convert $\mathcal{A}'$ to $\mathcal{A}$ by setting $\mathcal{A}$ to $\mathcal{P}_{\not\pi}(\mathcal{A}')$. We provide an example to illustrate:

EXAMPLE 6. Recall the following security policies and their corresponding advices from Section 1:

Advice	Policy	Pointcut
a	Read-send	$\phi\alpha$, where:
		$\phi \triangleq 1^*$`call`$\langle$*FileRead*$\rangle(\neg$`call`$\langle$*Send*$\rangle)^*$
		$\alpha \triangleq$ `call`$\langle$*Send*$\rangle$
b	Chinese Wall, part I	$\psi\beta$, where:
		$\psi \triangleq 1^*$`call`$\langle$*Login*, $A\rangle(\neg$`call`$\langle$*Login*, $B\rangle)^*$
		$\beta \triangleq$ `call`$\langle$*Login*, $B\rangle$

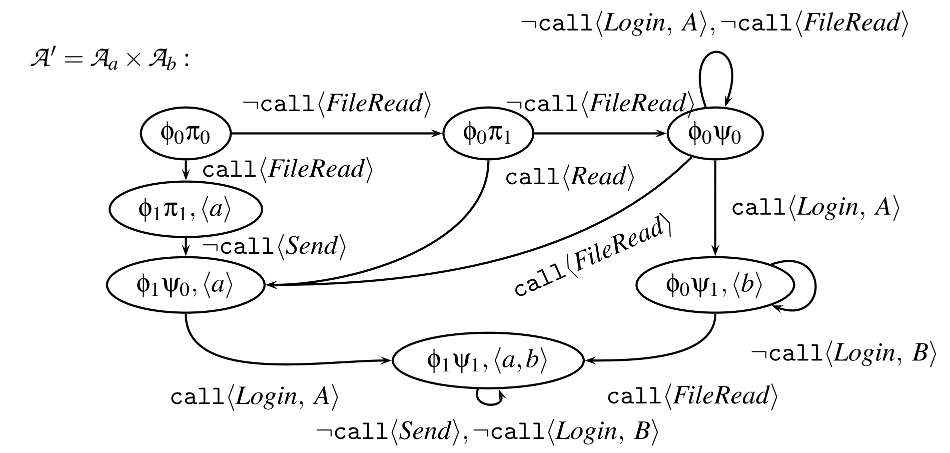
Consider the following scenario: advice a is declared, and we execute a $\texttt{call}\langle \mathit{read}\rangle$ followed by a $\texttt{call}\langle f\rangle$ for some function f. We then declare advice b. Our set of declarations now contains $\mathsf{adv}\ a[\phi\,\alpha]$ and $\mathsf{adv}\ b[11\ \psi\,\beta]$.

b's pointcut starts with '11', indicating that two $\texttt{call}$s were executed prior to its declaration. $\mathsf{adv}\ a[\phi\,\alpha]$ and $\mathsf{adv}\ b[\psi\,\beta]$ induce the following two automata, respectively:



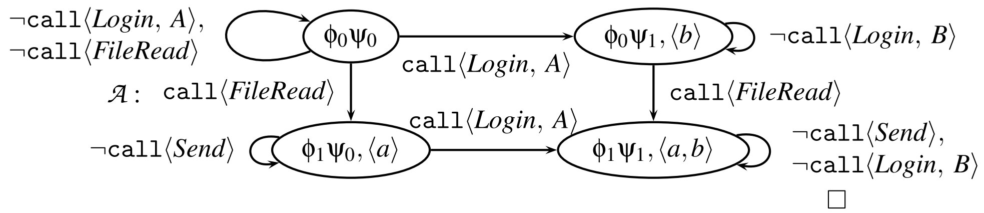
$\neg\texttt{call}\langle \mathit{FileRead}\rangle$
$\mathcal{A}_a$:
ϕ_0
$\texttt{call}\langle \mathit{FileRead}\rangle$
$\phi_1, \langle a\rangle$
$\neg \mathit{Send}$
1^*
$\mathcal{A}_b$:
π_0
1
π_1
1
ψ_0
$\texttt{call}\langle \mathit{Login},\ A\rangle$
$\psi_1, \langle b\rangle$
$\neg\texttt{call}\langle \mathit{Login},\ B\rangle$

The π states in $\mathcal{A}_b$ serve as dummy placeholders and represent the program history that was executed prior to a_1's declaration. To construct our intermediate automaton $\mathcal{A}'$, we compute the product automaton $\mathcal{A}_a \times \mathcal{A}_b$. The result is shown below:



$\mathcal{A}' = \mathcal{A}_a \times \mathcal{A}_b$:
$\neg\texttt{call}\langle \mathit{Login},\ A\rangle, \neg\texttt{call}\langle \mathit{FileRead}\rangle$
$\neg\texttt{call}\langle \mathit{FileRead}\rangle$
$\neg\texttt{call}\langle \mathit{FileRead}\rangle$
$\phi_0\pi_0$
$\phi_0\pi_1$
$\phi_0\psi_0$
$\texttt{call}\langle \mathit{FileRead}\rangle$
$\texttt{call}\langle \mathit{Read}\rangle$
$\texttt{call}\langle \mathit{Login},\ A\rangle$
$\phi_1\pi_1, \langle a\rangle$
$\neg\texttt{call}\langle \mathit{Send}\rangle$
$\texttt{call}\langle \mathit{FileRead}\rangle$
$\phi_1\psi_0, \langle a\rangle$
$\phi_0\psi_1, \langle b\rangle$
$\neg\texttt{call}\langle \mathit{Login},\ B\rangle$
$\phi_1\psi_1, \langle a,b\rangle$
$\texttt{call}\langle \mathit{Login},\ A\rangle$
$\texttt{call}\langle \mathit{FileRead}\rangle$
$\neg\texttt{call}\langle \mathit{Send}\rangle, \neg\texttt{call}\langle \mathit{Login},\ B\rangle$

To compute Φ, we simulate the program history $\bar{\sigma} = (\texttt{call}\langle \mathit{FileRead}\rangle, \texttt{call}\langle f\rangle)$ on $\mathcal{A}'$, and we find that $\Phi = \langle \phi_1\psi_0\rangle$. Finally, we compute $\mathcal{A}$ by removing from $\mathcal{A}'$ any states containing a π state. The result is shown below:



$\neg\texttt{call}\langle \mathit{Login},\ A\rangle$,
$\neg\texttt{call}\langle \mathit{FileRead}\rangle$
$\phi_0\psi_0$
$\phi_0\psi_1, \langle b\rangle$
$\neg\texttt{call}\langle \mathit{Login},\ B\rangle$
$\texttt{call}\langle \mathit{Login},\ A\rangle$
$\mathcal{A}$:
$\texttt{call}\langle \mathit{FileRead}\rangle$
$\texttt{call}\langle \mathit{FileRead}\rangle$
$\texttt{call}\langle \mathit{Login},\ A\rangle$
$\neg\texttt{call}\langle \mathit{Send}\rangle$
$\phi_1\psi_0, \langle a\rangle$
$\phi_1\psi_1, \langle a,b\rangle$
$\neg\texttt{call}\langle \mathit{Send}\rangle$,
$\neg\texttt{call}\langle \mathit{Login},\ B\rangle$

□

Finally, we formalize the translation from a history based configuration to an automaton-based one. That is, given a history, declaration pair $\langle \bar{\sigma}, \bar{D}\rangle$, we formally show how to construct the corresponding automaton, state, declaration triple $\langle \mathcal{A}, \Phi, \bar{E} \rangle$. Intuitively, we construct the intermediate product automaton $\mathcal{A}'$ as discussed above, we compute the state Φ by simulating the history $\bar{\sigma}$ on $\mathcal{A}'$, and we prune the intermediate states from the $\mathcal{A}'$ to obtain $\mathcal{A}$. Deriving $\bar{E}$ is trivial.

Our translation makes use of the following functions:

$$\mathcal{T}_{dec}(\bar{D}) = \{\mathsf{adv}\ a[\alpha] | \mathsf{adv}\ a[\phi\,\alpha] \in \bar{D}\}$$
$$\mathcal{T}_{state}(\bar{\sigma}, \bar{D}) = \Psi \text{ where } \bar{D} = \langle \mathsf{adv}\ \_[\phi_1\,\alpha_1], \ldots, \mathsf{adv}\ \_[\phi_n\,\alpha_n] \rangle \text{ and } \langle \phi_1, \ldots, \phi_n \rangle \overset{\bar{\sigma}}{\Longrightarrow} \Psi$$

We are now in a position to define the function $\mathcal{T}$ which translates a history-based configuration $\langle \bar{\sigma}; \bar{D} \rangle$ to an automaton-based configuration $\langle \mathcal{A}; \Phi; \bar{E} \rangle$:

DEFINITION 7. $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$, where

$$\mathcal{A} = \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\,\alpha] \in \bar{D}} \iota(\phi, a)\right] \qquad \Phi = \mathcal{T}_{state}(\bar{\sigma}, \bar{D}) \qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})$$

We define the language of a formula ϕ as follows:

$$\mathcal{L}_A(\bar{D}, \phi) = \{\bar{\sigma} | \bar{D} \vdash \phi \overset{\bar{\sigma}}{\Longrightarrow} \phi',\ \phi' \checkmark,\ \text{and } \phi' \in \mathcal{T}_{state}(\bar{\sigma}, \bar{D})\}$$

LEMMA 8. *For all $\bar{D}$ and ϕ, $\mathcal{L}_H(\bar{D}, \phi) = \mathcal{L}_A(\bar{D}, \phi)$.*
Proof. By induction on the structure of $\bar{\sigma}$. □

We conclude by showing that the translation is preserved by evaluation. That is, if

- $\bar{\sigma}; \bar{D} \rhd M \to \bar{\sigma}'; \bar{D}' \rhd M'$
- $\mathcal{T}(\bar{\sigma}, \bar{D}) = \mathcal{A}; \Phi; \bar{E}$,
- $\mathcal{A}; \Phi; \bar{E} \rhd M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M'$, and
- $\mathcal{T}(\bar{\sigma}', \bar{D}') = \mathcal{A}''; \Phi''; \bar{E}''$

then $\mathcal{A}' = \mathcal{A}'', \Phi' = \Phi''$, and $\bar{E}' = \bar{E}''$, as shown below:

$$\begin{tikzcd}
\bar{\sigma}; \bar{D} \arrow[r] \arrow[d, Rightarrow, "\mathcal{T}"'] & \bar{\sigma}'; \bar{D}' \arrow[d, Rightarrow, "\mathcal{T}"] \\
\mathcal{A}; \Phi; \bar{E} \arrow[r] & \mathcal{A}'; \Phi'; \bar{E}'
\end{tikzcd}$$

PROPOSITION 9. *If $\bar{\sigma}; \bar{D} \rhd M \to \bar{\sigma}'; \bar{D}' \rhd M'$ and $\mathcal{T}(\bar{\sigma}, \bar{D}) = \mathcal{A}, \Phi, \bar{E}$, then $\mathcal{A}; \Phi; \bar{E} \rhd M \to \mathcal{A}'; \Phi'; \bar{E}' \rhd M'$, where $\mathcal{T}(\bar{\sigma}', \bar{D}') = \mathcal{A}', \Phi', \bar{E}'$.*

Proof. In each case, we first translate the left hand side into the automaton-based semantics. We then apply the evaluation rule (e.g., EVAL-DEC-ADV) to the automaton to obtain the next configuration $\langle \mathcal{A}', \Phi', \bar{E}' \rangle$. We then translate the right hand side into the automaton based semantics and show that the result equals $\langle \mathcal{A}', \Phi', \bar{E}' \rangle$.

In the cases of EVAL-DEC-ROLE and EVAL-ADV, this is trivial.
In the case of EVAL-DEC-ADV, recall its evaluation rule:

$$\bar{\sigma}; \bar{D} \triangleright \mathsf{adv}\ b[\psi\beta], M \to \bar{\sigma}; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\beta] \triangleright M$$

The declarations $\bar{D}$ (equivalently $\bar{E}$) are trivially preserved by Eval-Dec-Adv, which leaves us to show that the automaton $\mathcal{A}$ and the state Φ are preserved.

We first show that the automaton is preserved by Eval-Dec-Adv: translating the left hand side yields $\mathcal{T}(\bar{\sigma}; \bar{D}) = \mathcal{A}; \Phi; \bar{E}$, where

$$\mathcal{A} \triangleq \mathcal{P}_{\not\pi}\Big[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\Big] \qquad \Phi \triangleq \mathcal{T}_{state}(\bar{D}) \qquad \bar{E} \triangleq \mathcal{T}_{dec}(\bar{D})$$

By EVAL-DEC-ADV, $\mathcal{A}; \Phi; \bar{E} \triangleright \mathsf{adv}\ b[\psi\beta], M \to \mathcal{A}'; \Phi'; \bar{E}' \triangleright M$, where

$$\mathcal{A}' \triangleq \mathcal{P}_{\not\pi}\Big[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\Big] \times \iota(\psi, b) \quad \Phi' \triangleq \mathcal{T}_{state}(\bar{D}), \psi \quad \bar{E}' \triangleq \mathcal{T}_{dec}(\bar{D}), b$$

Finally, we must show that $\mathcal{T}(\bar{\sigma}'; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\beta]) = \mathcal{A}'; \Phi'; \bar{E}'$. By definition, $\mathcal{T}(\bar{\sigma}'; \bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\beta]) = \mathcal{A}''; \Phi''; \bar{E}''$, where

$$\mathcal{A}'' = \mathcal{P}_{\not\pi}\Big[\big(\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\big) \times \iota(1^{|\bar{\sigma}|}\psi, b)\Big] = \mathcal{P}_{\not\pi}\Big[\big(\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\big) \times \iota(\psi, b)\Big]$$

Finally, Lemma 5 gives us that $\mathcal{A}' = \mathcal{A}''$:

$$\mathcal{P}_{\not\pi}\Big[\big(\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\big) \times \iota(\psi, b)\Big] = \mathcal{P}_{\not\pi}\Big[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\Big] \times \iota(\psi, b)$$

and hence that the automaton is preserved by EVAL-DEC-ADV.

To show that the state Φ is preserved by EVAL-DEC-ADV, we simulate $\bar{\sigma}$ on the intermediate automaton $\big[\prod_{\mathsf{adv}\ a[\phi\alpha] \in \bar{D}} \iota(\phi, a)\big] \times \iota(1^{|\bar{\sigma}|}\psi, b)$. It immediately follows that the resulting state $\Phi'' = \langle \Phi, \psi \rangle = \Phi'$. The declarations $\bar{E}$ are trivially preserved by EVAL-DEC-ADV.

We now consider the case of EVAL-CALL. We must show that in a history-based configuration $\langle \bar{\sigma}; \bar{D} \rangle$, for any declared advice $\mathsf{adv}\ a[\phi\alpha]$, if $\bar{D} \Vdash \bar{\sigma}\ \mathtt{sat}\ \phi$ and $\bar{D} \vdash \bar{p}\ \mathtt{sat}\ \alpha$ where $\bar{p}$ is the role vector being called, then $\mathsf{adv}\ a[\alpha]$ is associated with the state Φ in $\mathcal{T}(\bar{\sigma}, \bar{D})$. This follows directly from Lemma 8: if $\bar{D} \Vdash \bar{\sigma}\ \mathtt{sat}\ \phi$ in the history-based semantics, then in the automaton based semantics, $\phi \overset{\bar{\sigma}}{\Longrightarrow} \phi'$, where $\phi' \checkmark$, so $\langle \phi', a \rangle \in \Phi$.

Finally, the case of EVAL-COMMIT is trivial. Recall the evaluation rule in the history based semantics: $\bar{\sigma}; \bar{D} \triangleright M, \mathtt{commit}\langle \bar{p} \rangle \to \bar{\sigma}, \bar{p}; \bar{D} \triangleright M$, and in the automaton-based semantics:

$$\frac{\text{(EVAL-COMMIT)}\quad \bar{D} \vdash \Phi \overset{\bar{p}}{\Longrightarrow} \Psi}{\mathcal{A}; \Phi; \bar{E} \rhd \bar{b}, \texttt{commit}\langle \bar{p} \rangle \\ \rightarrow \mathcal{A}; \Psi; \bar{E} \rhd \bar{b}}$$

In this case, $\mathcal{T}(\bar{\sigma};\bar{D}) = \mathcal{A};\Phi;\bar{E}$, and $\mathcal{T}(\bar{\sigma},\bar{p};\bar{D}) = \mathcal{A};\Phi';\bar{E}$ where

$$\mathcal{A} = \mathcal{P}_{\not\pi}\left[\prod_{\mathsf{adv}\ a[\phi\,\alpha]\in\bar{D}} \iota(\phi,a)\right] \qquad\qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})$$

What remains is to show that $\Psi = \Phi'$. In doing so, we will have succeeded in showing that $\mathcal{A}, \Phi$, and $\bar{E}$ are all preserved by EVAL-COMMIT. By definition of $\mathcal{T}$, simulating $\bar{\sigma}$ on the intermediate automaton $\prod_{\mathsf{adv}\ a[\phi\,\alpha]\in\bar{D}} \iota(\phi,a)$ places $\mathcal{A}$ in state Φ. To derive Φ' from $\bar{\sigma},\bar{p};\bar{D}$, we simply carry the simulation one step further, taking transition $\bar{p}$. By EVAL-COMMIT in the automaton-based semantics, we know that $\Phi \overset{\bar{p}}{\Longrightarrow} \Phi'$, and hence that $\Psi = \Phi'$, which is what we needed to show. □

PROPOSITION 10. *If* $\mathcal{A};\Phi;\bar{E} \rhd M \rightarrow \mathcal{A}';\Phi';\bar{E}' \rhd M'$, *and* $\mathcal{T}(\bar{\sigma},\bar{D}) = \mathcal{A},\Phi,\bar{E}$, *then* $\bar{\sigma};\bar{D} \rhd M \rightarrow \bar{\sigma}';\bar{D}' \rhd M'$, *where* $\mathcal{T}(\bar{\sigma}',\bar{D}') = \mathcal{A}',\Phi',\bar{E}'$.

Proof. The proof closely parallels that of Proposition 9, and as such, we omit the details here. Details can be found in Appendix B.

This brings us to the main result: that the two semantics are equivalent:

THEOREM 11. $\bar{\sigma}; \bar{D} \rhd M \rightarrow^* \bar{\rho}; \bar{E} \rhd N$ *if and only if* $\mathcal{T}(\bar{\sigma}; \bar{D} \rhd M) \rightarrow^* \mathcal{T}(\bar{\rho}; \bar{E} \rhd N)$.

Proof. By Propositions 9 and 10, and induction on the length of $\rightarrow^*$. □

7 Conclusions

We have described a novel minimal language for aspect-oriented programming with temporal pointcuts. We described an implementation of the language using security automata and proved the correctness of the implementation. We have presented examples of applications to software security.

In extended the current work, we are most interested in the possibility of static analyses. In particular, we are interested in type-preserving translations of class-based languages into μABC. We have already developed untyped translations; finding type-preserving translations presupposes a suitable typing systems for μABC, which remains future work.

References

[1] Martín Abadi and Cedric Fournet. Access control based on execution history. In *Proceedings of the Network and Distributed System Security Symposium Conference*, 2003.

[2] Chris Allan, Pavel Avgustinov, Sascha Kuzins, Oege de Moor, Damien Sereni, Ganesh Sittampalam, Julian Tibble Aske Simon Christensen, Laurie Hendren, and Ondřej Lhoták. Adding trace matching with free variables to aspectj. In *OOPSLA 2005*, 2005.

[3] J. Andrews. Process-algebraic foundations of aspectoriented programming. In *In Reflection, LNCS 2192*, 2001.

[4] Steve Barker and Peter Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transations on Information and System Security*, 6(4):501–546, 2003.

[5] Glen Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In Philippa Gardner and Nobuko Yoshida, editors, *CONCUR 2004: Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224, London, August 2004. Springer.

[6] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parameterized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Submitted for publication, at ftp://ftp.ccs.neu.edu/pub/people/wand/papers/clw-03.pdf, oct 2003.

[7] R. Douence, O. Motelet, and M. Südholt. A formal definition of crosscuts. In *Proceedings of the 3rd International Conference on Reflection and Crosscutting Concerns*, LNCS. Springer Verlag, September 2001. long version is http://www.emn.fr/info/recherche/publications/RR01/01-3-INFO.ps.gz.

[8] Cdric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3):360–399, May 2003.

[9] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*. Springer, June 1997.

[10] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *In Foundations of Aspect-Oriented Languages workshop (FOAL'05)*, 2005.

[11] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[12] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs.

[13] W. De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, 1997.

[14] Fred Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[15] V. Stolz and E. Bodden. Temporal assertions using AspectJ. In *RV'05 - Fifth Workshop on Runtime Verification*, 2005. To Appear.

[16] Peter Thiemann. Enforcing safety properties using type specialization. In *Programming Languages and Systems: 10th European Symposium on Programming, ESOP 2001*, volume 2028. Springer, 2001.

[17] David Tucker and Shriram Krishnamurthi. Pointcuts and advice in higher-order languages. In *Conference Record of AOSD 03: The 2nd International Conference on Aspect Oriented Software Development*, 2003.

[18] Úlfar Erlingsson and Fred Schneider. Sasi enforcement of security policies: a retrospective. In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[19] David Walker. A type system for expressive security policies. In *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 254–267, New York, NY, USA, 2000. ACM Press.

[20] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Conference Record of ICFP 03: The ACM SIGPLAN International Conference on Functional Programming*, 2003.

[21] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *Twelfth International Symposium on the Foundations of Software Engineering*, 2004.

[22] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. appeared in Informal Workshop Record of FOOL 9, pages 67-88; also presented at FOAL (Workshop on Foundations of Aspect-Oriented Languages), a satellite event of AOSD 2002, 2002.

A Semantics of Temporal Pointcuts

TEMPORAL POINTCUT SATISFACTION $(\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi)$

(SAT-ATOM)

$$\frac{\bar{D} \vdash \sigma \text{ sat } \alpha}{\bar{D} \Vdash \sigma \text{ sat } \alpha}$$

(SAT-OR-LEFT)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi}{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi + \psi}$$

(SAT-SEQ)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi \qquad \bar{D} \Vdash \bar{\rho} \text{ sat } \psi}{\bar{D} \Vdash \bar{\sigma}, \bar{\rho} \text{ sat } \phi\psi}$$

(SAT-STAR)

$$\frac{\bar{D} \Vdash \bar{\sigma} \text{ sat } \phi \qquad \bar{D} \Vdash \bar{\rho} \text{ sat } \phi^*}{\bar{D} \Vdash \bar{\sigma}, \bar{\rho} \text{ sat } \phi^*}$$

$$\frac{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \psi}{\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi+\psi}\ \text{(SAT-OR-RIGHT)} \qquad \frac{}{\bar{D} \Vdash \varepsilon\ \mathsf{sat}\ \varepsilon}\ \text{(SAT-SEQ-EMPTY)} \qquad \frac{}{\bar{D} \Vdash \varepsilon\ \mathsf{sat}\ \phi^*}\ \text{(SAT-STAR-EMPTY)}$$

B Proof of Proposition 10

Again, in the cases of EVAL-DEC-ROLE and EVAL-ADV, this is trivial. In the case of EVAL-DEC-ADV, recall its evaluation rule:

$$\mathcal{A};\Phi;\bar{E} \rhd \mathsf{adv}\ b[\psi\beta], M \to \mathcal{A}';\Phi';\bar{E}' \rhd M$$

where

$$\mathcal{A}' = \mathcal{A} \times \iota(\psi, b) \qquad \Phi' = \Phi, \psi \qquad \bar{E}' = \bar{E}, b$$

In the history-based semantics, we have

$$\bar{\sigma};\bar{D} \rhd \mathsf{adv}\ b[\psi\beta], M \to \bar{\sigma};\bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\beta] \rhd M$$

where $\mathcal{T}(\bar{\sigma};\bar{D}) = \mathcal{A};\Phi;\bar{E}$. What remains is to show that $\mathcal{T}(\bar{\sigma};\bar{D}, \mathsf{adv}\ b[1^{|\bar{\sigma}|}\psi\beta]) = \mathcal{A}';\Phi';\bar{E}'$, which we already proved in Proposition 9.

In the case of EVAL-CALL, if a $\mathtt{call}\langle\bar{p}\rangle$ is replaced by the body of some advice $\mathsf{adv}\ a[\phi\alpha]$, this must mean that advice a is associated with the current state of the automaton, and that $\bar{D} \vdash \bar{p}\ \mathsf{sat}\ \alpha$. We must show that in the history-based semantics, (i) the advice $\mathsf{adv}\ a[\phi\alpha]$ is declared (trivial), (ii) that $\bar{D} \vdash \bar{p}\ \mathsf{sat}\ \alpha$ (given), and that (iii) $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$. Point (iii) follows directly from Lemma 8: since $\mathsf{adv}\ a[\alpha]$ is associated with the current state, it must mean that $\phi \overset{\bar{\sigma}}{\Longrightarrow} \phi'$, and $\phi' \checkmark$. By Lemma 8, it immediately follows that $\bar{D} \Vdash \bar{\sigma}\ \mathsf{sat}\ \phi$.

Finally, in the case of EVAL-COMMIT, recall its evaluation rule in the automaton-based semantics:

$$\frac{\bar{D} \vdash \Phi \overset{\bar{p}}{\Longrightarrow} \Psi}{\begin{array}{l} \quad \mathcal{A};\Phi;\bar{E} \rhd \bar{b}, \mathtt{commit}\langle\bar{p}\rangle \\ \to \mathcal{A};\Psi;\bar{E} \rhd \bar{b} \end{array}}\ \text{(EVAL-COMMIT)}$$

If $\mathcal{A};\Phi;\bar{E} \rhd \bar{b}, \mathtt{commit}\langle\bar{p}\rangle \to \mathcal{A};\Psi;\bar{E} \rhd \bar{b}$, then it must be the case that $\Phi \overset{\bar{p}}{\Longrightarrow} \Psi$. Now, let $\bar{\sigma};\bar{D}$ be the history-based configuration such that $\mathcal{T}(\bar{\sigma};\bar{D}) = \mathcal{A};\Phi;\bar{E}$. Then by definition of $\mathcal{T}$,

$$\mathcal{A} = \mathcal{P}_{\not{\pi}}\left[\prod_{\mathsf{adv}\ a[\phi\alpha]\in\bar{D}} \iota(\phi, a)\right] \qquad \bar{E} = \mathcal{T}_{dec}(\bar{D})$$

Recall the rule in the history-based semantics:

$$\bar{\sigma};\bar{D} \rhd M, \mathtt{commit}\langle\bar{p}\rangle \to \bar{\sigma}, \bar{p};\bar{D} \rhd M$$

We must show that $\mathcal{T}(\bar{\sigma},\bar{p};\bar{D}) = \mathcal{A};\Phi';\bar{E}$ where $\Phi' = \Psi$. $\mathcal{A}$ and $\bar{E}$ follow immediately from the definition of $\mathcal{T}$.

Furthermore, by definition of $\mathcal{T}$, simulating $\bar{\sigma}$ on the intermediate automaton $\prod_{\mathsf{adv}\ a[\phi\,\alpha]\in\bar{D}} \iota(\phi,a)$ puts the automaton in state Φ. To derive Φ' from $\bar{\sigma},\bar{p};\bar{D}$, we simply carry the simulation one step further, taking transition $\bar{p}$. By EVAL-COMMIT in the automaton-based semantics, we know that $\Phi \overset{\bar{p}}{\Longrightarrow} \Phi'$, and hence that $\Phi' = \Psi$, which is what we needed to show. □

C Derived Forms

To give a feel for the language, we define a few derived forms and discuss their execution. The derived forms make use of the syntactic transformation $hook(c,M)$, which places c at the end of the current advice list of M. (Recall that every term has the form $M = \bar{D}\,;\,\bar{a}\langle\bar{p}\rangle$).

$$hook(c,(\bar{D}\,;\,\bar{a}\langle\bar{p}\rangle)) \triangleq \bar{D}\,;\,c,\bar{a}\langle\bar{p}\rangle$$

DERIVED FORMS (LET)

$D,E\ ::=\ \cdots$	Declarations
$\quad\mathsf{let}\ \bar{x}\mathsf{=}N\,;$	$\quad$ Let; $dn(\mathsf{let}\ \bar{x}) = \bar{x}$
$\mathsf{let}\ \bar{x}\mathsf{=}N\,;\,M \triangleq (\mathsf{adv}\ c\mathsf{=}(\bar{x})\ M)\,;\,hook(c,N)$	Let; $c \notin fn(\bar{D}) \cup fn(M)$

For example we have the following.

$$\begin{aligned}\mathsf{let}\ \bar{x}\mathsf{=}N\,;\,\mathsf{let}\ \bar{y}\mathsf{=}L\,;\,M = {} & \mathsf{adv}\ c\mathsf{=}(\bar{x})\ \big((\mathsf{adv}\ d\mathsf{=}(\bar{y})\ M)\,;\,hook(d,L)\big)\,;\\ & hook(c,N)\end{aligned}$$

DERIVED FORMS (FUNCTIONS)

$L,M\ ::=\ \cdots$	Terms
$\quad\lambda\bar{x}.N$	$\quad$ Abstraction
$\lambda x.N \triangleq \mathsf{role}\,f;\ \big(\mathsf{adv}[\langle f,\mathsf{+top}\rangle]\mathsf{=}(\_,x)\ N\big)\,;\,\langle f\rangle$	Abstraction; $f \notin fn(N)$
$L\,M \triangleq \mathsf{let}\,f\mathsf{=}L;\ \mathsf{let}\ x\mathsf{=}M\,;\,\mathsf{call}\langle f,x\rangle$	Application; $f,x \notin fn(M)$

For example we have the following.

$$\begin{aligned}(\lambda x.\langle x\rangle)\ \langle p\rangle = {} & \mathsf{adv}\ c\mathsf{=}(f)\ (\mathsf{adv}\ d\mathsf{=}(x)\ \mathsf{call}\langle f,x\rangle)\,;\,d\langle p\rangle\,;\\ & \mathsf{role}\ g\,;\\ & \mathsf{adv}\ a[\langle g,\mathsf{+top}\rangle]\mathsf{=}(\_,x)\ \langle x\rangle\,;\\ & c\langle g\rangle\end{aligned}$$

Let $\bar{D} = \big(\mathsf{adv}\ c\mathsf{=}(f)\ (\mathsf{adv}\ d\mathsf{=}(x)\ \mathsf{call}\langle f,x\rangle)\,;\,d\langle p\rangle\big), \mathsf{role}\ g, \big(\mathsf{adv}\ a[\langle g,\mathsf{+top}\rangle]\mathsf{=}(\_,x)\ \langle x\rangle\big)$. Also let $\bar{E} = \bar{D},(\mathsf{adv}\ d\mathsf{=}(x)\ \mathsf{call}\langle g,x\rangle)$. Then we have the following.

$$(\lambda x.\langle x\rangle)\ \langle p\rangle \rightarrow\rightarrow\rightarrow \bar{D} \rhd c\langle g\rangle$$

$$
\begin{aligned}
&\rightarrow \bar{D} \triangleright (\mathsf{adv}\ d = (x)\ \mathsf{call}\langle g, x\rangle)\ ;\ d\langle p\rangle\\
&\rightarrow \bar{E} \triangleright d\langle p\rangle\\
&\rightarrow \bar{E} \triangleright \mathsf{call}\langle g, p\rangle\\
&\rightarrow \bar{E} \triangleright a\langle g, p\rangle\\
&\rightarrow \bar{E} \triangleright \langle p\rangle
\end{aligned}
$$