Provably Correct Runtime Monitoring* (Extended Abstract)

Irem Aktug[1], Mads Dam[2], and Dilian Gurov[1]

[1] Royal Institute of Technology (KTH), Sweden
[2] Access Linneaus Center, Royal Institute of Technology (KTH), Sweden

Abstract. Runtime monitoring is an established technique for enforcing a wide range of program safety and security properties. We present a formalization of monitoring and monitor inlining, for the Java Virtual Machine. Monitors are security automata given in a special-purpose monitor specification language, ConSpec. The automata operate on finite or infinite strings of calls to a fixed API, allowing local dependencies on parameter values and heap content. We use a two-level class file annotation scheme to characterize two key properties: (i) that the program is correct with respect to the monitor as a constraint on allowed program behavior, and (ii) that the program has an instance of the given monitor embedded into it, which yields state changes at prescribed points according to the monitor's transition function. As our main application of these results we describe a concrete inliner, and use the annotation scheme to characterize its correctness. For this inliner, correctness of the level II annotations can be decided efficiently by a weakest precondition annotation checker, thus allowing on-device checking of inlining correctness in a proof-carrying code setting.

1 Introduction

Program monitoring is a firmly established and efficient approach for enforcing a wide range of program security and safety properties [6, 10, 5, 9]. Several approaches to program monitoring have been proposed in the literature. In "explicit" monitoring, target program actions are intercepted and tested by some external monitoring agent [10]. A variant, examined by Schneider and Erlingsson [6], is monitor inlining, under which target programs are rewritten to include the desired monitor functionality, thus making programs essentially self-monitoring. This eliminates the need for a runtime enforcement infrastructure which may be costly on small devices. Also, it opens the possibility for third party developers to use inlining as a way of providing runtime guarantees to device users or their proxies. This, however, requires that users are able to trust that inlining has been performed correctly. In this work we propose a formalization of monitoring and monitor inlining as a first step towards addressing this concern.

* This work was partially funded by the S3MS project, IST-STREP-27004. The second author was partially supported by the Swedish Research Council grant 2003-6108.

We focus on monitors as security automata that operate on calls to some fixed API from a target program given as an abstract Java Virtual Machine (JVM) class file. Automaton transitions are allowed to depend locally on argument values, heap at time of call and (normal or exceptional) return, and return value. Our main contributions are characterizations, in terms of JVM class files annotated by formulas in a suitable Floyd-like program logic, of the following two conditions on a program:

1. That the program is policy-adherent.
2. The existence of a concrete representation of the monitor state inside the target program itself, as an inlined monitor which is compositional, in the sense that manipulations of the monitor state do not cross method call boundaries.

The annotations serve as an important intermediate step towards a decidable annotation validity problem, once the inliner is suitably instantiated. Compositionality allows validity to be checked per method. This is uncontroversial, and satisfied by all inliners we know of.

By these results, the verification of a concrete inliner reduces to proof of validity of the corresponding annotations. We use this to prove correctness for an inlining scheme which is introduced in the paper. We also sketch how, for a program inlined by such an inliner, the annotations can be completed to produce a fully annotated program for which validity can be efficiently decided. Such a fully annotated program can then be used by a bytecode weakest precondition checker in a proof-carrying code setting to certify monitor compliance to a third party such as a mobile device.

Related Work A closely related result is the recent work on type-based monitor certification by Hamlen et al [8]. That work focuses on per-object monitoring rather than the "per-session" model considered here. Also, their results are restricted to one particular inliner, whereas we give a characterization of a whole class of compositional inliners.

Our results can be seen as providing theoretical underpinnings for the earlier work by Schneider and Erlingsson [6]. The PoET/PSLang framework developed by Erlingsson represents monitors as Java snippets connected by an automaton superstructure. The code snippets are inserted into target programs at suitable points to implement the inlined monitor functionality. This approach, however, makes many monitor-related problems such as policy adherence and correctness undecidable. To overcome this, we base our results on a restricted monitor specification language, ConSpec [3], developed in the context of the EU project S3MS.

Organization In section 2 we present the JVM model used in this paper. Sections 3 and 4 introduce the automaton model in concrete and symbolic forms, the ConSpec language, and relations between the three. In section 5 we give an account of monitoring by interleaved co-execution of a target program with a monitor, and establish the equivalence of policy adherence and co-execution. In section 6, the two annotation levels are presented, and the main characterization theorems are proved. In section 7 an inliner and its correctness are presented. We also sketch how to produce, for this inliner, fully annotated programs with a

decidable validity problem. Finally, in section 8 we conclude and discuss future work. Due to space limitations, many technical details, all proofs and further examples are delegated to a technical report [1].

Acknowledgements Thanks are due to Andreas Lundblad for discussions on many issues relating to this paper, and to Johan Linde for his work on the inliner tool.

2 Program Model

We briefly present the components of JVM used in this paper.

Types Fix sets of class names $c \in \mathbb{C}$, method names $m \in \mathbb{M}$, and field names $f \in \mathbb{F}$. A type $\tau \in \mathit{Type}$ is either a primitive type, not further specified, or an object type, determined by a class name c. An object type determines a set of fields and methods defined through its class declaration. Class declarations induce a class hierarchy, and $c_1 <: c_2$ if c_1 is a subclass of c_2. If c is the smallest superclass (under $<:$) of c' that contains an explicit definition of $c.m$ then c defines $c'.m$. Single inheritance ensures that definitions are unique, if they exist.

Values and Methods Values of object type are (typed) locations $\ell \in \mathit{Loc}$, mapped to objects by a heap $h \in \mathbb{H}$, a partial assignment of objects to locations. Objects determine typed fields and methods, using standard dot notation, and $\mathit{type}(\ell, h)$ is the type of ℓ in h, if defined. A method definition is an environment Γ (usually elided) taking a method reference $M = c.m$ to a definition (P, H) consisting of a method body (instruction sequence) P, and an exception handler array H. Method overloading is not considered. The notation $M[L] = I$ indicates that $\Gamma(M) = (P, H)$ and $P(L)$ is defined and equal to the instruction I. The exception handler array H is a partial map from integer indices to exception handlers. An exception handler (b, e, t, c) catches exceptions of type c and its subtypes raised by instructions in the range $[b, e)$ and transfers control to address t, if it is the topmost handler that covers the instruction for this exception type.

Machine Configurations, Transitions and Type Safety A *configuration* of the JVM is a pair $C = (R, h)$, where R is a stack of activation records of the form either (M, pc, s, lv) for some method reference M, program counter pc, operand stack s, and local variables lv, or, for exceptional states, of the form $(\ell)_e$, where ℓ is the location of an exceptional object. $\mathit{Unhandled}(C)$ holds if C has an exceptional frame on top of the frame stack, and the current method does not have a handler for the exception. We assume a standard *transition relation* $\longrightarrow_{\mathrm{JVM}}$ on JVM configurations (cf. [7]). An *execution* E of a program (class file) T is then a (possibly infinite) sequence of JVM configurations $C_1 C_2 \ldots$ where C_1 is an initial configuration consisting of a single, normal activation record with an empty stack, no local variables, M as a reference to the main method of P, $pc = 1$, Γ set up according to T, and for each $i \geq 1$, $C_i \longrightarrow_{\mathrm{JVM}} C_{i+1}$. We restrict attention to configurations that are *type safe*, in the sense that heap contents match the types of corresponding locations, and that arguments and return/exceptional values for primitive operations as well as method invocations match their prescribed types. The Java bytecode verifier serves, among other things, to ensure that type safety is preserved under machine transitions.

API Method Calls The only non-standard aspect of $\longrightarrow_{\mathrm{JVM}}$ is the treatment of API methods. We assume a fixed API for which we have access only to the signature, but not the implementation, of its methods. We therefore treat API method calls as atomic instructions with a non-deterministic semantics. Our approach hinges on our ability to recognize such method calls. This property is destroyed by the *reflect* API, which is left out of consideration. Among the method invocation instructions, we discuss here only `invokevirtual`; the remaining invoke instructions are treated similarly.

3 Security Policies and Automata

Let T be a program for which we identify a set of *security relevant actions* A. Each execution of T determines a corresponding set $\Pi(\mathrm{T}) \subseteq A^* \cup A^\omega$ of finite or infinite traces of actions in A. A *security policy* is a predicate on such traces, and T *satisfies* a policy $\mathcal{P}$ if $\mathcal{P}(\Pi(\mathrm{T}))$.

The notion of security automata was introduced by Schneider [11]. We view a *security automaton* over alphabet A as an automaton $\mathcal{A} = (Q, \delta, q_0)$ where Q is a countable set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times A \rightharpoonup Q$ is a (partial) transition function. All $q \in Q$ are viewed as accepting. A security automaton $\mathcal{A}$ induces a security policy $\mathcal{P}_\mathcal{A} \subseteq 2^{A^* \cup A^\omega}$ through its language $L_\mathcal{A}$ by $\mathcal{P}_\mathcal{A}(X) \Leftrightarrow X \subseteq L_\mathcal{A}$.

In this study, we focus on security automata which are induced by policies in the ConSpec language (see section 4) and therefore are named *ConSpec automata*. The security relevant actions are method calls, represented by the class name and the method name of the method, along with a sequence of values that represent the actual arguments. We partition the set of security relevant actions into *pre-actions* $A^\flat \subseteq \mathbb{C} \times \mathbb{M} \times \mathit{Val}^* \times \mathbb{H}$ and *post-actions* $A^\sharp \subseteq \mathit{RVal} \times \mathbb{C} \times \mathbb{M} \times \mathit{Val}^* \times \mathbb{H} \times \mathbb{H}$, corresponding to method invocations and returns. Both types of actions may refer to the heap prior to method invocation, while the latter may also refer to the heap upon termination and to a return value from $\mathit{RVal} = \mathit{Val} \cup \{\mathrm{exc}\}$ where exc is used to mark exceptional return from a method call[3]. The partitioning on security relevant actions induces a corresponding partitioning on the transition function δ of ConSpec automata into a function $\delta^\flat$ on pre-actions, and a function $\delta^\sharp$ on post-actions.

4 ConSpec: A Monitor Specification Language

A monitor specification in ConSpec determines a collection of security relevant actions (sra's), a security state, and for each security relevant action, a transition rule, using a guarded command-like syntax. In addition, in [3] a scope declaration is introduced which is ignored in this paper. As an example, consider the following specification:

[3] We disregard the exceptional value since we do not, as yet, put constraints on these in ConSpec policies.

```
SECURITY STATE bool accessed   = false; bool permission = false;

BEFORE File.Open(string path, string mode, string access)
PERFORM mode.equals("CreateNew")                          -> { skip; }
        mode.equals("Open") && access.equals("OpenRead")  -> { accessed = true; }

AFTER bool answer = GUI.AskConnect()
PERFORM answer  -> { permission = true; }
        !answer -> { permission = false; }

BEFORE Connection.Open(string type, string address)
PERFORM !accessed || permission -> { permission = false; }
```

The sra's are self-explanatory. The security state is a pair of boolean variables `accessed` and `permission`, which record whether an existing file has been accessed and if a permission has been obtained. The example policy contains three *clauses* that state the conditions for and effect of the security relevant actions. The sra of a clause is identified by the signature of the method mentioned in the clause. The *modifiers* BEFORE and AFTER (or EXCEPTIONAL) indicate whether it is the call of, or the normal (or exceptional) return from, the method that is security relevant. For each sra, there can exist at most one event clause per modifier in the policy. In order to determine if the policy allows an sra, the guards of the corresponding clause are evaluated *top to bottom* using the current value of the security state variables and the values of the relevant program variables. If none of the conditions hold for the current sra, it is violating and no more sra's are allowed by the policy.

Fix a set *Svar* of security state variables and a set *Var* of program variables. The security state variables of ConSpec are restricted to strings, integers and booleans. Expressions *Exp* and boolean expressions *BoolExp* over $Svar \cup Var$ can access object fields and use standard arithmetic and boolean operations. Strings can be compared for equality or prefix.

The formal semantics of ConSpec policies is defined in terms of *symbolic security automata*, which in turn induce ConSpec automata.

Definition 1 (Symbolic Security Automaton). *A* symbolic security automaton *is a tuple* $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$*, where:*

(i) $q_s = Svar$ *is the initial and only state;*
(ii) $Init_s : q_s \to Val$ *is an* initialization function*;*
(iii) $A_s = A_s^\flat \cup A_s^\sharp$ *is a countable set of* symbolic actions*, where:*
$A_s^\flat \subseteq \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ *and* $A_s^\sharp \subseteq \{(Type \times Var) \cup \{exc\}\} \times \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ *are the* symbolic pre- *and* post-actions*, respectively;*
(iv) $\delta_s = \delta_s^\flat \cup \delta_s^\sharp$ *is a* symbolic transition relation*, where:*
$\delta_s^\flat \subseteq A_s^\flat \times BoolExp \times (q_s \to Exp)$ *and* $\delta_s^\sharp \subseteq A_s^\sharp \times BoolExp \times (q_s \to Exp)$
are the symbolic pre- *and* post-transitions*, respectively.*

ConSpec policies and symbolic automata are two very similar representations. The security state variables of a ConSpec policy determines the state of the symbolic automaton. Each sra clause gives rise to one symbolic action, and each guarded command of the clause gives rise to a symbolic transition consisting of the sra itself, the guard of the guarded command in conjunction with negations

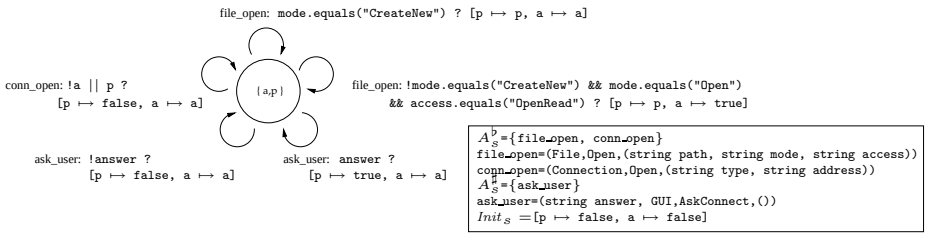

Fig. 1. Symbolic Automaton for the Example Policy

of the guards that lie above it in the clause, and the effect of the guarded command. The updates to security state variables, which are presented as a sequence of assignments in ConSpec, are captured in the automaton as functions that return one ConSpec expression per symbolic state variable, determining the value of that variable after the update. In fig. 1 we illustrate the construction on the earlier example, using "a" for `accessed` and "p" for `permission`.

Symbolic automata are converted to ConSpec automata without too much effort. The details are given in [1]. Here it suffices to note that states in the induced ConSpec automaton are members of the lifted function space $(q_s \rightarrow Val)_\perp$. The bottom element, in particular, is used only as the target of post-transitions that are disallowed (has an unsatisfied boolean guard) in the symbolic automaton; it has no outgoing transitions.

5 Monitoring with ConSpec Automata

In this section we formalize the enforcement language of a ConSpec automaton as a set of finite strings of security relevant actions. Each target transition can give rise to zero, one, or two security relevant actions, namely, in the latter case, a pre-action followed by a post-action. Accordingly, we define the security relevant pre-action, $act^\flat(C)$, of the configuration C, and the corresponding post-action, $act^\sharp(C_1, C_2)$, as in the table below. If none of the conditions of the table hold, the corresponding action is ϵ.

$act^\flat(C)$	Condition
(c, m, s, h_b)	$C = ((M, pc, s \cdot [d] \cdot s', lv) \cdot R, h^\flat) \quad M[pc] = \texttt{invokevirtual}\ c'.m$ c *defines* $type(d, h^\flat).m \quad type(h^\flat, d) <: c' \quad (c, m, s, h^\flat) \in A^\flat$
$act^\sharp(C_1, C_2)$	Condition
$(v, c, m, s, h^\flat, h^\sharp)$	$C_1 = ((M, pc, s \cdot d \cdot s', lv) \cdot R, h^\flat) \quad M[pc] = \texttt{invokevirtual}\ c'.m$ $C_2 = ((M, pc + 1, v \cdot s', lv) \cdot R, h^\sharp) \quad c$ *defines* $type(h^\flat, d).m$ $type(h^\flat, d) <: c' \quad (v, c, m, s, h^\flat, h^\sharp) \in A^\sharp$
$(\text{exc}, c, m, s, h^\flat, h^\sharp)$	$C_1 = ((M, pc, s \cdot d \cdot s', lv) \cdot R, h^\flat) \quad M[pc] = \texttt{invokevirtual}\ c'.m$ $C_2 = ((b)_e \cdot (M, pc, s \cdot d \cdot s', lv) \cdot R, h^\sharp) \quad c$ *defines* $type(h^\flat, d).m$ $type(h^\flat, d) <: c' \quad (\text{exc}, c, m, s, h^\flat, h^\sharp) \in A^\sharp$

We obtain the *security relevant trace*, $srt_A(w)$, of an execution w by lifting the operations $act^\flat$ and $act^\sharp$ co-inductively to executions in the following way:

$$srt_A(\epsilon) = \epsilon \qquad srt_A(C) = act^\flat(C)$$
$$srt_A(C_1C_2 \cdot w) = act^\flat(C_1) \cdot act^\sharp(C_1, C_2) \cdot srt_A(C_2 \cdot w)$$

Then a target program T *adheres* to a policy $\mathcal{P}$, if the security trace of each execution of T is in the enforcement language of the corresponding automaton $\mathcal{A}_\mathcal{P}$, i.e. $\forall E \in \Pi(\mathrm{T}).\ srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}$.

Program-Monitor co-execution A basic application of a ConSpec automaton is to execute it alongside a target program to monitor for policy compliance. We can view such an execution as an interleaving $w = (C_0, q_0)(C_1, q_1)\cdots$ such that C_0 and q_0 is the initial configuration and state of T and $\mathcal{A}$, respectively, and such that for each consecutive pair $(C_i, q_i)(C_{i+1}, q_{i+1})$, either the target (only) progresses: $C_i \longrightarrow_{\mathrm{JVM}} C_{i+1}$ and $q_{i+1} = q_i$ or the automata (only) progresses: $C_{i+1} = C_i$ and $\exists a \in A.\ \delta(q_i, a) = q_{i+1}$. In the former case we write $(C_i, q_i) \longrightarrow_{\mathrm{JVM}} (C_{i+1}, q_{i+1})$, and in the latter case we write $(C_i, q_i) \longrightarrow_{\mathrm{AUT}} (C_{i+1}, q_{i+1})$. We can w.l.o.g. assume that at most one of these cases apply, for instance by tagging each interleaving step.

The first projection function $w \downarrow 1$ on interleavings $w = (C_1, q_1)(C_2, q_2)\cdots$ extracts the underlying execution sequence $C_1'C_2'\cdots$ such that $C_1' = C_1$, and $C_2' = C_1$ if $(C_1, q_1) \longrightarrow_{\mathrm{AUT}} (C_2, q_2)$ and $C_2' = C_2$ otherwise, and so on. To extract the automaton states and the security relevant actions, we use the (co-inductive) function *extract*:

$$extract((C_1, q_1)(C_2, q_2)w) = q_1 q_2\, extract((C_2, q_2)w)$$

if $(C_1, q_1) \longrightarrow_{\mathrm{AUT}} (C_2, q_2)$,

$$extract((C_1, q_1)(C_2, q_2)w) = act^\flat(C_1)act^\sharp(C_1, C_2)extract((C_2, q_2)w),$$

if $(C_1, q_1) \longrightarrow_{\mathrm{JVM}} (C_2, q_2)$, $extract(C, q) = act^\flat(C)$, and $extract(\epsilon) = \epsilon$.
Note that $extract(w)$ may well be finite even if w is infinite.

Definition 2 (Co-Execution). *Let* $E^\flat = \{qq'a^\flat \mid q, q' \in Q, a^\flat \in A^\flat, \delta^\flat(q, a^\flat) = q'\}$, $E^\sharp = \{a^\sharp qq' \mid q, q' \in Q, a^\sharp \in A^\sharp, \delta^\sharp(q, a^\sharp) = q'\}$. *An interleaving w is a co-execution if* $extract(w) \in (E^\flat \cup E^\sharp)^* \cup (E^\flat \cup E^\sharp)^\omega$.

In other words, an interleaving is a co-execution, if the sequence of extracted automaton states corresponds to an automaton run for the security relevant trace of the underlying execution.

Theorem 1 (Correctness of Monitoring by Co-execution). *The program T adheres to policy $\mathcal{P}$ if, and only if, for each execution $C_1C_2\cdots$ of T there is a co-execution w for the automaton $\mathcal{A}_\mathcal{P}$ such that $w \downarrow 1 = C_1C_2\cdots$.*

6 Specification of Monitoring

We specify monitor inlining correctness using annotations in a Floyd-style logic for bytecode. The idea behind our annotation scheme is the following. In a first annotation, referred to as the *policy* (or, level I) *annotation*, we define a monitor for the given policy by means of "ghost" variables, updated before or after every security relevant action according to the symbolic automaton induced by the given security policy. In a second annotation, referred to as *synchronisation check annotation* (or level II), we add assertions that check at all relevant program points that the actual inlined monitor (represented by global program variables) agrees with the specified one (represented by ghost variables).

6.1 Language of Ghost Annotations

Assertions Methods are augmented with annotations that determine assertions on the extended state (current configuration and current ghost variable assignment), and actions on ghost variables. Let g range over ghost variables, $i \in \omega$, and let Op (Bop) range over a standard, not further specified, collection of unary and binary operations (comparison operations) on strings and integers. Assertions a, and expressions e used in assertions, have the following shape:

$$e ::= \bot \mid v \mid g \mid e.f \mid \mathrm{s}[i] \mid Op\ e \mid e\ Op\ e$$
$$a ::= e\ Bop\ e \mid e : c \mid \neg a \mid a \wedge a \mid a \vee a$$

Here, $\mathrm{s}[i]$ is the value at the i'th position of the current operation stack, if defined, and $\bot$ otherwise, and $e : c$ is a class membership test.

Ghost Variable Assignments Ghost variables are assigned using a single, guarded multi-assignment of the form

$$\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} | \cdots | a_m \rightarrow \overrightarrow{e_m} \tag{1}$$

such that the arities (and types) of $\overrightarrow{gs}$ and the $\overrightarrow{e_i}$ match. The idea is that the first assignment $\overrightarrow{gs} := \overrightarrow{e_i}$ is assigned such that the guard a_i is true in the current extended state. If no guard is true, the ghost state is assigned the constant $\bot$-vector. This happens, in particular, when $m \leq 0$ in (1) above, which we write as $\overrightarrow{gs} := ()$.

Method Annotations A target program is annotated by an extended environment, Γ^*, which maps method references M to tuples $(P, H, A, Requires, Ensures, Exsures)$ such that $Requires$, $Ensures$ and $Exsures$ are assertions, and such that A is an assignment to each program point $n \in Dom(P)$ of a sequence, ψ, of atomic annotations, either an assertion or a ghost variable assignment.

Annotation Semantics In the absence of ghost variable assignments the notion of annotation validity is the expected one, i.e. that the assertions annotating any given program point (or the point of exceptional return) are all guaranteed to be valid. To extend this account to ghost variables, we use a rewrite semantics, shown on table 1. In the table, extended configurations are triples of the form

$$(1)\quad \frac{Assert(a, C, \sigma)}{\Gamma^* \vdash (a\psi, C, \sigma) \to (\psi, C, \sigma)}$$

$$(2)\quad \frac{\| a_1 \| (C, \sigma) = \texttt{TRUE}, \quad m > 0}{\Gamma^* \vdash ((\overrightarrow{gs} := a_1 \to \overrightarrow{e_1} | \cdots | a_m \to \overrightarrow{e_m})\psi, C, \sigma) \to (\psi, C, \sigma[\| \overrightarrow{e_1} \| (C, \sigma)/\overrightarrow{gs}])}$$

$$(3)\quad \frac{\| a_1 \| (C, \sigma) \neq \texttt{TRUE}, \quad m > 0}{\Gamma^* \vdash ((\overrightarrow{gs} := a_1 \to \overrightarrow{e_1} | \cdots | a_m \to \overrightarrow{e_m})\psi, C, \sigma) \to ((\overrightarrow{gs} := a_2 \to \overrightarrow{e_2} | \cdots | a_m \to \overrightarrow{e_m})\psi, C, \sigma)}$$

$$(4)\quad \frac{\cdot}{\Gamma^* \vdash ((\overrightarrow{gs} := ())\psi, C, \sigma) \to (\psi, C, \sigma[\overrightarrow{\bot}/\overrightarrow{gs}])}$$

$$(5)\quad \frac{C \longrightarrow_{\text{JVM}} C' \quad Unexc(C')}{\Gamma^* \vdash (\epsilon, C, \sigma) \to (A(\Gamma^*(M(C')))(pc(C')), C', \sigma)}$$

$$(6)\quad \frac{C \longrightarrow_{\text{JVM}} C', \quad Unhandled(C')}{\Gamma^* \vdash (\epsilon, C, \sigma) \to (Exsures(\Gamma^*(M(C))), C', \sigma)} \qquad (7)\quad \frac{C \longrightarrow_{\text{JVM}} C' \quad Handled(C')}{\Gamma^* \vdash (\epsilon, C, \sigma) \to (\epsilon, C', \sigma)}$$

Table 1. Operational Semantics of Annotations

(ψ, C, σ) such that ψ is the sequence of annotations remaining to be evaluated for the current program point in C. We use abbreviations M, pc, A, *Requires*, *Ensures*, and *Exsures* for the first to sixth projections, respectively. *Unexc* holds of a configuration that does not have an exceptional frame on the top of the stack, and $Unexc(C) \Leftrightarrow \neg(Handled(C) \vee Unhandled(C))$. The side condition $Assert(a, C, \sigma)$ always returns true, but as a sideeffect causes the arguments to be "asserted", e.g. to appear on some output channel. For rule (6), note that unhandled exceptions causes the assertions in the *Exsures* clause to be asserted.

Definition 3 (Validity). *A program annotated according to the rules set up above is* valid *for the annotated environment Γ^*, if all predicates asserted as a result of a Γ^*-derivation $(\psi_0, C_0, \sigma_0) \longrightarrow_{\text{JVM}} \cdots \longrightarrow_{\text{JVM}} (\psi_n, C_n, \sigma_n) \longrightarrow_{\text{JVM}} \cdots$ are valid, where ψ_0 is $Requires(\Gamma^*(\langle\texttt{main}\rangle)) \cdot A_{\langle\texttt{main}\rangle}[1]$, C_0 is an initial configuration, and $\sigma_0 = \bot$.*

6.2 Policy Annotations (Level I)

The *policy annotations* define a monitor for the given policy by means of a ghost state. The ghost state is initialized in the precondition of the $\langle\texttt{main}\rangle$ method and updated at relevant points by annotating all the methods defined by the classes of the target program. We call each such method an *application method*. We assume that $\langle\texttt{main}\rangle$ is not called by any application method (including itself) and that all exceptions that may be raised by a security relevant instruction (i.e. an instruction that may lead to a security relevant action) are covered by a handler. We also assume that the exception handling is structured such that the only way an instruction in an exception handler gets executed is if an exception has been raised and caught by the handler that the instruction belongs to. Finally, we assume w.l.o.g. that there are no jumps to instructions below method invocations.

Updating the Specified Security State The updates to the specified security state are done according to the transitions of the symbolic automaton. If the automaton does not have a transition for a security relevant method call, the call is violating and the corresponding annotation sets the value of the specified state to undefined. Such a program should terminate without executing the next security relevant action in order to adhere to the policy. This is specified by asserting, as a precondition to each security relevant method invocation and at updates to the ghost state, that the ghost state is not undefined. If a security relevant instruction may cause a pre-action (an unexceptional post-action) of the automaton, then a ghost assignment annotation is inserted as a precondition (as a postcondition) to this instruction. Finally, if the instruction can cause an exceptional post-action, the update is inserted as a precondition to the first instruction of each exception handler that covers the instruction.

Preliminary Definitions In the definitions below, fix a program T and a policy $\mathcal{P}$. Let $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$ be the symbolic automaton induced by $\mathcal{P}$.We define the set $A_s^e \subseteq A_s^\sharp$ of exceptional symbolic post-actions as those which have the value exc as their first component. Given a symbolic action set A'_s, the function $RS((c,m), A'_s)$ returns those subclasses c' of c for which the method $(c'.m)$ is defined by a class c'' such that A'_s has an action with the reference $(c''.m)$. The variables of $\overrightarrow{gs}$ are named identical to the security state variables of the automaton.The ghost variable g_{pc} records labels of security relevant instructions. and ghost variables g stack values. For an expression mapping $E : q_s \to Exp$, let $\overrightarrow{e_E}$ denote the corresponding expression tuple, and for a boolean ConSpec expression $b \in BoolExp$, let a_b denote the corresponding assertion.

Level I Annotation We define the annotations for every method M, through three arrays of annotations: a pre-annotation array $A_M^\flat[i]$, a post-annotation array $A_M^\sharp[i][j]$, and an exceptional annotation array $A_M^e[i][k]$, where i ranges over the instructions of method M. The second index $j \in \{0,1\}, k \in \{0,1,2\}$ indicates whether the annotation will be placed as a precondition of the instruction $(j, k = 0)$, as a precondition to the next instruction $(j, k = 1)$, or as a precondition to all the exception handlers of the instruction $(k = 2)$. The predicate *Handler* holds for a label L and a method M if $(L_1, L_2, L, c) \in H_M$ for some labels L_1, L_2, and class name c. In addition, we define $Exc(L, M)$ as the sequence of all annotations $A_M^e[L'][2]$ where L' is a security relevant instruction and there exists an exception handler $(L_1, L_2, L, c) \in H_M$ such that $L_1 \leq L' < L_2$, and as ϵ if such an L' does not exist.

Given these annotations, the *level I annotation* of program T is given for each application method M as a precondition $Requires_M^{\mathrm{I}}$ and an array A_M^{I} of annotation sequences defined as follows (where $L > 0$):

$$Requires_M^{\mathrm{I}} = \begin{cases} (\overrightarrow{gs} := \overrightarrow{e_{Init_s}}) \cdot (g_{\mathrm{pc}} := 0) & \text{if } M = \langle \texttt{main} \rangle \\ (g_{\mathrm{pc}} := 0) & \text{otherwise.} \end{cases}$$

$$A_M^{\mathrm{I}}[1] = A_M^\flat[1] \cdot A_M^\sharp[1][0] \cdot A_M^e[1][0]$$

$$A_M^{\mathrm{I}}[L] = \begin{cases} Exc(L,M) \cdot A_M^\flat[L] \cdot A_M^\sharp[L][0] \cdot A_M^e[L][0] & \text{if } Handler(L,M) \\ A_M^e[L-1][1] \cdot A_M^\sharp[L-1][1] \cdot A_M^\flat[L] \cdot A_M^\sharp[L][0] \cdot A_M^e[L][0] & \text{otherwise} \end{cases}$$

The annotation $\mathit{Requires}_M$ resets the value of g_{pc} and, if $M = \langle \texttt{main} \rangle$, also initializes the ghost state using function Init_s of the automaton.

After Annotations For every method M, the elements of the post-annotation array $A^\sharp_M[L]$ are defined for each label L as follows:

(i) If the instruction at label L is not an `invokevirtual` instruction or is of the form $M[L] = \texttt{invokevirtual}\ c.m$ where $RS((c,m), A^\sharp_s \setminus A^e_s) = \emptyset$, we define the pre- and postconditions to be empty:$A^\sharp_M[L][0] = A^\sharp_M[L][1] = \epsilon$

(ii) Otherwise, if the instruction at label L is of the form $M[L] = \texttt{invokevirtual}\ c.m$ with $c.m : (\gamma \to \tau)$ and $|\gamma| = n$ and $RS((c,m), A^\sharp_s \setminus A^e_s) = \{c'_1, \ldots, c'_p\}$, then the precondition of the instruction saves the arguments and the object in ghost variables:

$$A^\sharp_M[L][0] = ((g_0, \ldots, g_{n-1}, g_{\mathrm{this}}) \texttt{:=} (\mathrm{s}[0], \ldots, \mathrm{s}[n])) \cdot \mathit{Defined}^\sharp$$

The assertion $\mathit{Defined}^\sharp$ checks if the ghost variables are defined:

$$\mathit{Defined}^\sharp = ((g_{\mathrm{this}} : c'_1 \vee \ldots \vee g_{\mathrm{this}} : c'_p) \Rightarrow (\overrightarrow{gs} \neq \overrightarrow{\bot}))$$

while the postcondition of the instruction uses these saved values to compute the new security state:

$$A^\sharp_M[L][1] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha)$$

where the α_k are the guarded expressions $(\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge g_{\mathrm{this}} : c'_i \wedge a_b\rho_i \to \overrightarrow{e_E}\rho_i$ where class c'' defines (c'_i, m) and there exists $a^\sharp_s = (\tau\ x, c'', m, (\tau_0\ x_0, \ldots, \tau_{n-1}\ x_{n-1}))$, $a^\sharp_s \in A^\sharp_s \setminus A^e_s$ such that $(a^\sharp_s, b, E) \in \delta^\sharp_s$. The substitution ρ_i is defined as $[\mathrm{s}[0]/x,\ g_0/x_0,\ \ldots,\ g_{n-1}/x_{n-1},\ g_{this}/\texttt{this}]$. Finally, $\alpha = \neg(g_{\mathrm{this}} : c'_1 \vee \ldots \vee g_{\mathrm{this}} : c'_p) \to \overrightarrow{gs}$.

The annotation arrays $A^\flat_M$ and A^e_M are defined similarly (see [1] for details).

Each execution of a program that is valid w.r.t. level I annotations for policy $\mathcal{P}$ is a co-execution of the program and the automaton for $\mathcal{P}$, where the automaton states are given by the ghost state; hence the program adheres to $\mathcal{P}$.

Theorem 2 (Correctness of Level I Annotations). *Program T annotated with level I annotations for policy $\mathcal{P}$ is valid, if and only if T adheres to $\mathcal{P}$.*

6.3 Synchronisation Check Annotations (Level II)

An inlined program can be expected to contain an explicit representation of the security state, an *embedded state*, which is updated in synchrony with the execution of security relevant actions. The level II annotations aim to capture this idea in a generic form that is independent of the design choices a specific inliner may make. To this end, we make two assumptions on the inliner. We require that the embedded state is in agreement with the ghost state immediately prior to execution of a security relevant action. This condition would be violated by, for

example, an optimized inliner which determines in advance that a fixed sequence of security relevant actions is permissible and reflects this to the embedded state through only a single update. The second assumption we make in this section is that updates to the embedded state are made *locally*, that is by the method that executes the security relevant method call. The specified and the embedded states are synchronized then at all call points.

For simplicity we assume that the embedded state is determined as a fixed vector $\overrightarrow{ms}$ of global static variables of the target program, of types corresponding pointwise to the type of ghost state vector $\overrightarrow{gs}$. The *synchronisation assertion* is the equality $\overrightarrow{gs} = \overrightarrow{ms}$, and the *level II annotations* are formed by appending the synchronization assertion to the level I annotations of each application method M at the following points: (i) each annotation $A(\Gamma^*(M))(i)$ such that $P(\Gamma^*(M))(i)$ is an invoke or a return instruction, and (ii) the annotation $Exsures(\Gamma^*(M))$.

$A^{II}[L]$	L	$M[L]$
		⋮
	L3	`dup`
	L4	`astore r1`
$\left\{\begin{array}{l} g_{\text{this}} := \text{s}[0] \cdot \\ g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \cdot \\ (g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}) \end{array}\right\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\left\{\begin{array}{l} (g_{\text{a}}, g_{\text{p}}) := \\ ((g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \text{s}[0]) \rightarrow (g_{\text{a}}, \mathit{true}) \quad \mid \\ ((g_{\text{a}}, g_{\text{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \neg\text{s}[0]) \rightarrow (g_{\text{a}}, \mathit{false}) \mid \\ (\neg(g_{\text{this}} : \texttt{GUI})) \rightarrow (g_{\text{a}}, g_{\text{p}}) \end{array}\right\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
$\{ (g_{\text{a}}, g_{\text{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}) \}$	L13	`ireturn`

Fig. 2. An application method with level II annotations for the example policy

Level II Annotation Example An application method annotated with level II annotations for the example policy of section 4 is shown in fig. 2. The ghost state is represented by the ghost variables g_{a} and g_{p}, i.e. $\overrightarrow{gs} = (g_{\text{a}}, g_{\text{p}})$. The embedded state consists of the static fields `accessed` and `permission` of the `SecState` class. It is assumed that the class `GUI` does not have any subclasses. The annotated method is valid since the embedded state is updated as is described by the policy, after a call to the method `GUI.AskConnect`. The annotations are enclosed by braces and placed on the left of the instruction label they are associated with.

Level II Characterization We now explain in what sense the level II annotations characterize the two conditions assumed in this section (the synchronous update assumption, and the method-local update assumption).

Consider a program T with a level II annotated environment Γ^*. Consider an execution $E = C_0 C_1 \cdots$ from an initial configuration C_0 of T. The index i is a *sampling point* if one of the following three conditions holds:

(i) the top frame of C_i has the shape $(M, pc, s, f) \cdot R; h$, and $M[pc]$ is either an `invokevirtual` instruction, or a return instruction;
(ii) the configuration C_{i-1} has the shape $(M, pc, s, f) \cdot R; h$ where $M[pc]$ is an `invokevirtual` instruction, and C_i, the shape $(N, 1, \epsilon, f')(M, pc, s, f) \cdot R; h$;
(iii) alternatively, $\mathit{Unhandled}(C_i)$.

We can then construct a sequence $w(E, \overrightarrow{ms}) = (C_0, q_0)(C_1, q_1) \cdots$ such that: q_0 is the initial automaton state, for all sampling points $i > 0$, $q_i = C_i(\overrightarrow{ms})$, where $C_i(\overrightarrow{ms})$ denotes the value of $\overrightarrow{ms}$ in configuration C_i, and for any two consecutive sampling points i and i', for all $j : i \leq j < i'$, $q_j = q_i$.

The role of the sequence $w(E, \overrightarrow{ms})$ is similar to that of interleavings in section 5. However, the sequence $q_0 q_1 \cdots$ may not necessarily correspond to an automaton run: the intermediate automaton state is not sampled when a post-action is followed by a pre-action without an intermediate method boundary crossing, as there is no well-defined point where this might be done. The construction also needs to account for the method-local nature of embedded state updates. For this reason, we define the operation $\mathit{extract}_{II}$, taking sequences w to strings over the alphabet $Q \cup A \cup \{\mathtt{I}\}$ where $\mathtt{I}$ is a distinguished symbol, by the following conditions:

- $\mathit{extract}_{II}((C_1, q_1)(C_2, q_2)w) = q_1 \mathit{act}^\flat(C_1) \mathit{act}^\sharp(C_1, C_2) q_2 \mathit{extract}_{II}((C_2, q_2)w)$, if C_1 is an API method call.
- $\mathit{extract}_{II}((C_1, q_1)(C_2, q_2)w) = q_1 \mathtt{I} q_2 \ \mathit{extract}_{II}((C_2, q_2)w)$, if C_1 is an application method call and $\mathit{Unexc}(C_2)$, i.e. C_2 is a method entry point.
- $\mathit{extract}_{II}((C, q)w) = q\mathtt{I}q \ \mathit{extract}_{II}(w)$, if C is a return point from an application method, either normal or exceptional.
- $\mathit{extract}_{II}((C_1, q_1)(C_2, q_2)w) = \mathit{extract}_{II}((C_2, q_2)w)$, otherwise.
- $\mathit{extract}_{II}((C, q)) = q \ \mathit{act}^\flat(C)$ if C is a method call and ϵ otherwise.

Definition 4 (Method-local Co-execution). *Let*

$$\begin{aligned}
\Sigma_0 &= \{\mathtt{I}, q, a^\flat, a^\sharp \mid q \in Q, a^\flat \in A^\flat, a^\sharp \in A^\sharp\},\\
\Sigma_1 &= \{\mathtt{I}\} \cup Q \cup E^\flat \cup E^\sharp \cup \{a^\sharp q q' a^\flat \mid \exists q''.\delta^\flat(q, a^\flat) = q'', \delta^\sharp(q'', a^\sharp) = q'\},\\
\Sigma_2 &= \{qq'q'', qq'q, \mathtt{I}qq'a^\sharp, \mathtt{I}qq'\mathtt{I}, \mathtt{I}qq'q', qa^\sharp q',\\
&\qquad qa^\flat a^\sharp q', a^\flat qq'q', a^\flat qq'\mathtt{I}, a^\flat qq'a^\sharp, q\mathtt{I}q', qa^\flat q' \mid q \neq q' \neq q''\}
\end{aligned}$$

A sequence w is a method-local co-execution, if

$$\mathit{extract}_{II}(w) \in (\Sigma_1^* \cup \Sigma_1^\omega) \setminus (\Sigma_0^* \cdot \Sigma_2 \cdot (\Sigma_0^* \cup \Sigma_0^\omega))$$

We can then extend theorem 2 to the situation where a target program T has a monitor for the given policy inlined into it.

Theorem 3 (Level II Characterization). *The level II annotation of T with embedded state $\overrightarrow{ms}$ is valid if, and only if, for each execution E of T, the sequence $w(E, \overrightarrow{ms})$ is a method-local co-execution.*

7 Correctness of Inlining

As an application of the annotation scheme described in the previous section, we characterize the correctness of a class of inliners in the flavor of PoET/PSLang [6]. We first describe the operation of a simple inliner that embeds, in target programs, a method-local monitor for a ConSpec policy.

Description of Inlining The inliner adds a class definition to the program. The static variables of this class serve as the embedded state. Since this class is not in the original namespace, the embedded state is safe from interference by the target. For each clause in the policy, a piece of bytecode is created, which evaluates, in turn, the guards of guarded commands and either updates the security state according to the update block associated with the first condition that holds or quits the program if none of them hold.

The rewriting process consists of identifying method invocation instructions that lead to security relevant actions (security relevant instructions), and for each such instruction, inserting code produced by policy compilation in an appropriate manner. The inliner inserts, immediately before the security relevant instruction, code that records the object the method is called for, and the arguments (and possibly parts of the heap) in local variables. Then, code for the relevant BEFORE clauses of the policy (if any) is inserted. Next, the object and the method arguments are restored on the stack. If there are AFTER clauses in the policy for the instruction, first the return value (if any) is recorded in a local variable, the code compiled from the AFTER clauses is inlined, followed by code to restore the return value on the stack. Finally, if there are EXCEPTIONAL clauses for the instruction, an exception handler is created that covers only the method invocation instruction and catches all types of exceptions. It is placed highest amongst the handlers for this label in the handler list, so that whenever the instruction throws an exception, this handler will be executed. The code of this exception handler consists of code created for the related EXCEPTIONAL clauses and ends by rethrowing the caught exception. All (original) exception handlers of the program that cover the security relevant instruction are redirected to cover this last throw instruction instead.

Due to virtual method call resolution, execution of an invocation instruction can give rise to different security relevant actions. The inliner inserts code to resolve, at runtime, the signature of the method that is called, using the type of the object that the method is invoked on, and information on which methods have been overridden. A check to compare this signature against the signature of the event mentioned in the clause is prepended to code compiled for the clause.

Correctness of Inlining Inliners as described above are expected to satisfy the following property. Let $\mathcal{P}$ be a policy, T a program and $M[L]$ be a post-security relevant instruction $M[L]$ of the inlined program T$'$. Let $M[L] =$ `invokevirtual` $(c.m)$ for some c and m, $\alpha_1, \ldots, \alpha_m$ be the guarded expressions $g_{\text{this}} : c'_i \wedge\ a_b\rho_i \rightarrow \overrightarrow{e_E}\rho_i$, $1 \leq i \leq m$, and α be $\neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \rightarrow \overrightarrow{gs}$, induced, by the policy, for $M[L]$ as described in section 6.2. Furthermore, let r_{this} be the local variable used by the inliner to record the reference of the object $M[L]$ operates on. Then the weakest pre-condition of the block of code

inlined immediately after the instruction $M[L]$ in T′ w.r.t. the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$ is the logical assertion

$$\begin{array}{c} \bigwedge_{1\leq i\leq m} r_{\text{this}} : c'_i \wedge \ a_b\rho'_i \ \rightarrow \ \overrightarrow{gs} = \overrightarrow{e_E}\rho'_i \\ \wedge \neg(r_{\text{this}} : c'_1 \vee \ldots \vee r_{\text{this}} : c'_p) \ \rightarrow \ \overrightarrow{gs} = \overrightarrow{ms} \end{array}$$

The blocks inlined above and at the exception handlers of security relevant instructions can be specified similarly.

We claim that it is possible to devise an inliner in accordance with the description above. Let I be such an inliner, and let I(T,$\mathcal{P}$) denote the program T inlined by I for the policy $\mathcal{P}$. Our implementation of such an inliner is found at [2].

The following result shows that programs inlined for a policy contain a monitor as characterized by theorem 3, and that level II annotations can be efficiently completed to a "fully" annotated program for which annotation validity, and hence policy adherence, is decidable. In the result, *local validity* refers to logical validity of the verification conditions resulting from a fully annotated program (see [4] for details).

Theorem 4. *Let $\mathcal{P}$ be a ConSpec policy and T a program.*

(i) The inlined program I(T,$\mathcal{P}$) is valid with respect to the level II annotation for this policy.

(ii) For I(T,$\mathcal{P}$), the level II annotation can be efficiently extended to an annotation so that: (a) the extended annotation is locally valid (in terms of the pre- and postconditions of the individual instructions) if and only if the level II annotation is valid (in terms of definition 3), and (b) local validity is decidable.

An extended (or level III) annotation as referred to above can be obtained by: (a) annotating all non-inlined instructions with the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$, (b) extending the annotation to inlined instructions by means of a syntactic weakest precondition function $wp(M[L])$ (as defined in [4]), and (c) collapsing every annotation to an equivalent single assertion (see [1] for details).

As a corollary of theorem 2 and the above result, every program inlined with the described inliner adheres to the policy it was inlined for.

Corollary 1 (Correctness of Inlining). *Let $\mathcal{P}$ be a ConSpec policy and T be a program. The inlined program I(T,$\mathcal{P}$) adheres to the policy.*

Another corollary of theorem 4 is that the inlined program I(T,$\mathcal{P}$) yields only method-local co-executions. This is so since programs that validate level III annotations validate also level II annotations and thus theorem 3 applies to inlined programs.

As a consequence, a level III annotation as described above can be used for on-device checking of inlining correctness in a proof-carrying code setting.

8 Conclusion

This extended abstract presents a specification language for security policies in terms of security automata, and a two-level class file annotation scheme in a Floyd-style program logic for Java bytecode, characterizing two key properties: (i) that a program adheres to a given policy, and (ii) that the program has an embedded method-compositional monitor for this policy. The annotation scheme thus characterizes a whole class of monitor inliners. As an application, we describe a concrete inliner and prove its correctness. For this inliner, validity of the annotations can be decided efficiently using a weakest precondition annotation checker, thus allowing the annotation scheme to be used in a proof-carrying code setting for certifying monitor compliance. This idea is currently being developed within the European S3MS project.

Future effort will focus on generalizing the level II annotations by formulating suitable state abstraction functions to extend the present approach to programs that are not inlined but still self-monitoring. Another interesting challenge is to extend the annotation framework to programs with threading.

References

1. I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. Technical Report TRITA-CSC-TCS 2008:1, CSC KTH, 2007. http://www.csc.kth.se/∼irem/S3MS/TechRep07.pdf.
2. I. Aktug and J. Linde. An inliner tool for mobile platforms. http://www.csc.kth.se/∼irem/S3MS/Inliner/.
3. I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In F. Piessens and F. Massacci, editors, *Proc. of The First Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197-1 of *Electronic Notes in Theoretical Computer Science*, pages 45–58, 2007.
4. F. Y. Bannwart and P. Müller. A logic for bytecode. In *Proc. of BYTECODE'05*, volume 141-1 of *ENTCS*, pages 255–273, 2005.
5. L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. of the ACM SIGPLAN Conf. on Prog. Lang. Design and Implementation*, pages 305–314, 2005.
6. Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symp. on Security and Privacy*, page 246. IEEE Computer Society, 2000.
7. S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Trans. Program. Lang. Syst.*, 21(6):1196–1250, 1999.
8. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'06)*, pages 7–16, June 2006.
9. K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.
10. K. Havelund and G. Rosu. Synthesizing monitors for safety properties. In *Proc. of Tools and Algorithms for Construction and Analysis of Systems (TACAS'02)*, volume 2280, pages 342–356, 2002.
11. F. B. Schneider. Enforceable security policies. *ACM Trans. Infinite Systems Security*, 3(1):30–50, 2000.