Provably Correct Runtime Monitoring*

Technical Report

Irem Aktug Mads Dam Dilian Gurov

Royal Institute of Technology (KTH), Sweden
{irem,mfd,dilian}@csc.kth.se

February 5, 2008

Abstract

Runtime monitoring is an established technique to enforce 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 behaviour, and (ii) that the program has a copy of the given monitor embedded into it, through a state abstraction function that yields state changes at prescribed points according to the monitor's transition function. As our main application of these results we present a concrete inliner, and use the annotation scheme to show its correctness. For this inliner, correctness of the level II annotations can be decided efficiently using a standard wp 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 to enforce a wide range of program security and safety properties [14, 6, 9, 8, 15, 13, 16, 19, 18, 12]. 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 [14, 15, 13, 16]. A variant, examined by Schneider and Erlingsson [9], is monitor inlining, under which target programs are rewritten to include the desired monitor functionality, thus making programs essentially self-monitoring [5, 8, 4]. 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.

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

*This work was funded by the IST-STREP-27004 S3MS project.

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 characterizations, the verification of a concrete monitor inliner reduces to proof of validity of the corresponding annotations. For practical monitors, this is not a difficult task. We illustrate this by describing a monitor inliner for which we prove correctness. We also sketch how, for this inliner, the annotations can be completed to produce a fully annotated program for which validity can be efficiently decided, and can thus be used by a standard bytecode wp 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 in this direction is the recent work on type-based monitor certification by Hamlen et al [11]. That work mainly focuses on per-object monitoring as compared to the "per-session" model considered in this paper. 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 [8]. 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 [2].

Organization The document is structured as follows. Section 2 presents 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. Section 5 gives an account of monitoring by interleaved (co-) execution of a target program with a monitor, and establishes 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 the inliner and its correctness proof are sketched. 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.

2 Program Model

We assume the reader to be familiar with Java bytecode syntax, the Java Virtual Machine (JVM), and formalisations of the JVM such as [10]). Here we only present components of the JVM, that are essential for the definitions in the rest of the text.

Classes and Types Fix sets of class names $c \in \mathbb{C}$, method names $m \in \mathbb{M}$, and field names $f \in \mathbb{F}$. A primitive type in the set *PrimType* is either `int`, `string`, or `Unit`, the latter used for methods that do not return a value. A type $\tau \in \mathit{Type}$ is either a primitive type or an object type, determined by a class name c and we let $\gamma \in (\mathit{Type})^*$ range over tuples of types. Each object type determines a set of fields and methods defined for that type through its class declaration. The class declarations induce a class hierarchy, and we write $c_1 <: c_2$ if c_1 is a subclass of c_2. If c defines m (declares f) explicitly, then c defines (declares) $c.m$ ($c.f$). We say that c defines $c'.m$ (declares $c'.f$) if c is the smallest superclass of c' that contains an explicit definition (declaration) of $c.m$ ($c.f$). Single inheritance ensures that definitions/declarations are unique, if they exist.

Values Each type $\tau \in \mathit{Type}$ determines a set $\|\tau\|$ of values. *Val* denotes the set of all values. Values of types `int` and `string` are integers and strings, values of *PrimType* are the primitive values *PrimVal*, and *void* is the single member of $\|\texttt{Unit}\|$. Values of object type are (typed) locations $\ell \in \mathit{Loc}$, mapped to objects by a heap $h \in \mathbb{H} = \mathit{Loc} \rightharpoonup \mathbb{O}$. The partial function $\mathit{type} : (\ell, h) \mapsto \mathbb{C}$ returns the type of location ℓ in heap h, if $\ell \in \mathit{Dom}(h)$, and is otherwise undefined (i.e. $\perp$). The structure of objects in $\mathbb{O}$ is not further specified here. It suffices to assume that if $h : \ell \mapsto o \in \mathbb{O}$ then $h(\ell)$ determines a field $h(\ell).f$ whenever f is declared in the class to which the object type of ℓ is associated.

Methods Method definitions are modelled as an environment Γ taking method references to their definitions. Where possible, Γ is elided. To simplify notation, method overloading is not considered, so a method is uniquely identified by a method reference of the form $M = (c, m)$. For a method $(c.m)$, $(c.m) : \gamma \to \tau$ if γ is the list of argument types and τ is the return type of the method. A method definition is a pair (P, H) consisting of a method body P and an exception handler array H. A method body P is a partial function from ω to the set of instructions such that $\mathit{ADDR}_P = \mathit{Dom}(P)$ has the form $\{1, \ldots, n\}$ for some $n \in \omega$. We use the notation $M[L] = I$ to indicate 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 A *configuration* of the JVM is a pair $C = (R, h)$ of a stack R of activation records and a heap h. For normal execution, the activation record at the top of the execution stack has the shape (M, pc, s, f), where

- The *program counter pc* is an index into the currently executing instruction array, i.e. it is a member of $\mathit{Dom}(P)$ where P is the body of the currently executing method. The configuration C is *calling*, if $P(\mathit{pc})$ is an invoke instruction, and it is *returning normally*, if $P(\mathit{pc})$ is a return instruction.
- The *operand stack* s is the stack of values (= primitive values or locations) currently being operated on.

- The *local variables lv* is a mapping of variables to values, preserving types.

For exceptional configurations C the top frame has the form $(b)_e$ where b is the location of an exceptional object. Then, C is *returning exceptionally*, and C is *returning* if C is either returning normally or exceptionally.

Machine Transitions We assume a transition relation $\longrightarrow_{\mathrm{JVM}}$ on JVM configurations. Such an operational semantics can be found for instance in [10]. An *execution* E of a program (class file) P is then a (possibly infinite) sequence of JVM configurations $C_1C_2C_3\ldots$ 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 P, 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 (cf. [17]).

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. This is similar to the approach taken, e.g., in [20]. In this sense, we do not practice *complete mediation* [].When an API method is called either the *pc* is incremented and arguments popped from the operation stack and replaced by an arbitrary return value of appropriate type, or else an arbitrary exceptional activation record is returned. Similarly, the return configurations for API method invocations contain an arbitrary heap, since we do not know how API method bodies change heap contents.

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 [21]. Here, 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

$RVal = Val \cup \{\text{exc}\}$ where exc is used to mark exceptional return from a method call[1]. The partitioning on security relevant actions induces a corresponding partitioning on the transition function δ of ConSpec automata.

4 ConSpec: A Contract Specification Language

In this section, we introduce the policy specification language ConSpec [2]. ConSpec is strongly inspired by PSLang, which was developed by Erlingsson and Schneider [7] for runtime monitoring. However, ConSpec is more restricted: a guarded-command language is used for the updates where the guards are side-effect free and commands do not contain loops. ConSpec supports expressing security requirements on different levels, like multiple executions of the same application, and on the executions of all applications of a system, besides requirements on all objects of a particular class and on single executions of an application, which can be expressed by PSLang. Notice that, we focus on policies on a single execution of an application in this work.

ConSpec Policy Example Assume method `Open` of class `File` is used for creating files (when argument `mode` has value "CreateNew") or for opening files (`mode` is "Open"), either for reading (argument `access` is "OpenRead") or for writing. Assume further that method `Open` of class `Connection` is used for opening connections, that method `AskConnect` is used for asking the user for permission to open a connection and that this latter method returns true in case of approval. Now, consider the security policy, which allows applications to access existing files for reading only, and requires, once such a file has been accessed, applications to obtain approval from the user each time a connection is to be opened. The policy also does not allow the application to execute further if a file opening operation raises an exception. This policy can be specified in ConSpec as follows:

```
SCOPE Session
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; }

EXCEPTIONAL File.Open(string path, string mode, string access)
PERFORM
      FALSE -> { skip; }

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

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

We first specify that the policy applies to each single execution of an application. The *security state* is represented by the boolean variables `accessed` and `permission`, which are intended to record whether an existing file has been accessed and whether there is an obtained permission. The example policy contains three *event clauses* that state the

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

conditions for and effect of the security relevant actions: call to the method `File.Open`, exceptional return from the method `File.Open`, call to the method `Connection.Open` and normal return from the method `GUI.AskConnect`. The event of an event clause is identified by the signature of the method mentioned in the clause. The types of the method arguments are specified along with representative names, which have the event clause as their scope. The *modifiers* BEFORE and AFTER mark whether the call of or the normal return from the method specified in the event clause is security relevant. If the exceptional return from a method is considered security relevant, then this is specified by the modifier EXCEPTIONAL. For each event, there can exist at most one event clause per modifier in the policy. In order to determine if the policy allows an event, the guards of the corresponding event clause is 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 event, it is a violating event and no more security relevant events are allowed by the policy.

ConSpec Expressions The security state variables of ConSpec are restricted to strings, integers and booleans. Expressions can access object fields and use standard arithmetic and boolean expressions. Strings can be compared for equality or prefix. The sets of expressions and boolean expressions of ConSpec are *Exp* and *BoolExp*, respectively.

The formal semantics of ConSpec policies is defined in terms of *symbolic security automata*, which in turn induce ConSpec automata. Fix a set *Svar* of security state variables and a set *Var* of program variables.

Definition 4.1 (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:*
$A_s^\flat \subseteq \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ *are* symbolic pre-actions*, and*
$A_s^\sharp \subseteq \{(\{PrimType \cup \mathbb{C}\} \times Var) \cup \texttt{Unit} \cup \{exc\}\} \times \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ *are* symbolic post-actions*;*

(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 set of security state variables of a ConSpec policy is 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 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`.

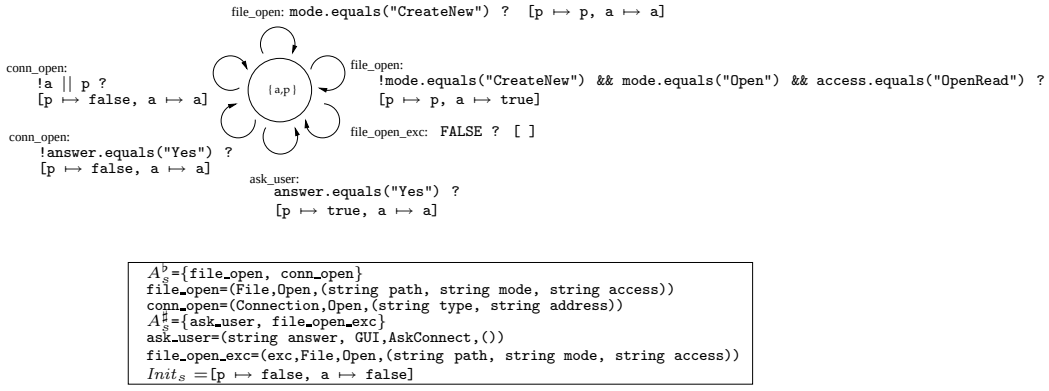

Figure 1: Symbolic Automaton for the Example Policy

Symbolic automata determine ConSpec automata in the following way: Let $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$ be a symbolic automaton. The ConSpec automaton induced by $\mathcal{A}$ is the automaton $\mathcal{A} = ((q_s \to Val)_\perp, \delta, Init_s)$ over alphabet A, determined as follows:

- The post-actions of A are all tuples $(v, c, m, v_1 \cdots v_n, h^\flat, h^\sharp)$ such that there is a symbolic post-action $a^\sharp_s = (r, c, m, ((\tau_1\ x_1),\ \ldots,\ (\tau_n\ x_n)))$ with $v_i : \tau_i$ for all $i : 1 \leq i \leq n$, and either $r = \tau x$ and $v : \tau$ or else $x = r \in \{void, \text{exc}\}$. The pre-actions are defined similarly.

- The post-transition function $\delta^\sharp$ is defined indirectly, by referring to the standard denotational semantic functions for expressions $e \in Exp$ and boolean expressions $b \in BoolExp$ such that $[\![e]\!] : (SVar \to Val) \to (Var \to Val) \to \mathbb{H} \to \mathbb{H} \to Val$ and $[\![b]\!] : (SVar \to Val) \to (Var \to Val) \to \mathbb{H} \to \mathbb{H} \to Val$, defined as expected. Then, if $\delta^\sharp_s(a^\sharp_s, b, E)$ in $\mathcal{A}_s$, we define $\delta^\sharp(q, a^\sharp) = q'$ in $\mathcal{A}$ if and only if there exists an interpretation I and heaps $h^\flat$ and $h^\sharp$ such that $[\![a^\sharp_s]\!]\ I\ h^\flat\ h^\sharp = a^\sharp$, $[\![b]\!]\ q\ I\ h^\flat\ h^\sharp = true$, and $[\![E(v)]\!]\ q\ I\ h^\flat\ h^\sharp = q'(v)$ for all $v \in SVar$. The pre-transition function $\delta^\flat$ is defined similarly. In addition, given post-action $a^\sharp_s$, let B be the set of boolean expressions b such that $\delta^\sharp_s(a^\sharp_s, b, E)$ for some E. Then, for every state $q \in Q$, interpretation I, and heaps $h^\flat$ and $h^\sharp$, we define $\delta^\sharp(q, a^\sharp) = \perp$ if $[\![a^\sharp_s]\!]\ I\ h^\flat\ h^\sharp = a^\sharp$ and $[\![b]\!]\ q\ I\ h^\flat\ h^\sharp = false$ for all $b \in B$.

It is not difficult to characterize the language of a ConSpec automaton obtained from a symbolic ConSpec automaton $\mathcal{A}_s$ directly in terms of $\mathcal{A}_s$ itself.

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 preaction followed by a postaction. Given the action set A, and the configurations C_1 and C_2, we define the security relevant preaction, $act^\flat_A(C_1)$, of the configuration C_1, and the corresponding postaction $act^\sharp_A(C_1, C_2)$, as in the table below. If none of the conditions of the table hold, the corresponding action is ϵ.

$act^{\flat}_{\mathcal{P}}(C)$	Condition
(c, m, s, h_b)	$C = ((M, pc, s \cdot [d] \cdot s', lv) \cdot R, h^{\flat})$ $M[pc] = \texttt{invokevirtual}\ c'.m, \quad c\ defines\ type(d, h^{\flat}).m, \quad type(h^{\flat}, d) <: c'$ $(c, m, s, h^{\flat}) \in A^{\flat}$

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

Check from JVM paper

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

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

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.

$$\forall E \in \Pi(\mathrm{T}).\ srt_A(E) \in L_{\mathcal{A}_{\mathcal{P}}}$$

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:

$$C_i \longrightarrow_{\mathrm{JVM}} C_{i+1} \text{ and } q_{i+1} = q_i$$

or the automata (only) progresses:

$$C_{i+1} = C_i \text{ and } \exists a \in A.\ \delta(q_i, a) = q_{i+1}.$$

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 as follows:

$$\begin{aligned} ((C_1, q_1)(C_2, q_2) \cdot w') \downarrow 1 &= \begin{cases} C_1(((C_2, q_2) \cdot w') \downarrow 1) & C_1 \longrightarrow_{\mathrm{JVM}} C_2 \\ ((C_2, q_2) \cdot w') \downarrow 1) & otherwise \end{cases} \\ (C, q) \downarrow 1 &= C \end{aligned}$$

To similarly extract automata derivations we use the (co-inductive) function *extract* such that

$$extract((C_1, q_1)(C_2, q_2)w) = q_1q_2 extract((C_2, q_2)w)$$

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

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

if $(C_1, q_1) \longrightarrow_{\mathrm{JVM}} (C_2, q_2)$, $extract(C, q) = act^{\flat}_A(C)$, and $extract(\epsilon) = \epsilon$. We call such an extracted sequence of automaton states and security relevant action a *potential derivation*. Note that $extract(w)$ may well be finite even if w is infinite.

Definition 5.1 (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 potential derivation it extracts corresponds to a real derivation.

Theorem 5.2 (Correctness, Monitoring by Co-execution)**.** *The program* T *adheres to policy* $\mathcal{P}$ *if, and only if, for each execution* $C_0C_1\cdots$ *of* T *there is a co-execution* w *for the automaton* $\mathcal{A}_\mathcal{P}$ *such that* $w \downarrow 1 = C_0C_1\cdots$.

Proof. Proof can be found in Appendix B □

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 *policy annotation* (or level I), 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) is "in sync" 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:

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

Here, 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

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

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 $\overrightarrow{gs} := ()$.

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

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

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

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

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

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

Table 1: Operational Semantics of Annotations

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, an account of the way ghost variables are updated and checked is needed. We use a rewrite semantics for this, shown on table 6.1. In the table, extended configurations are triples of the form (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, second, third, fourth, fifth and sixth projections, respectively. *Unhandled* holds of a configuration if it has an exceptional frame on top of the frame stack, and Γ does not contain a handler for that exception in the current method. *Handled* holds of a configuration if it has an exceptional frame on top of the frame stack, and Γ does contain a handler for that exception in the current method. Finally, *Unexc* holds of a configuration that does not have an exceptional frame on the top of the stack,i.e. $Unexc(C) \Leftrightarrow \neg(Handled(C) \vee Unhandled(C))$

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 (5), note that unhandled exceptions must cause any assertions in the *Exsures* clause to be asserted.

Definition 6.1 (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 $(\psi_0, C_0, \sigma_0) \to \cdots \to (\psi_n, C_n, \sigma_n) \to \cdots$ 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 $\sigma_0 = [\,]$.*

6.2 Policy Annotations (Level I)

The *policy annotations* define a monitor for the given policy by means of ghost variables. The ghost variables, which constitute the *specified security state*, are initialized in the precondition of the ⟨`main`⟩ method and updated at relevant points by annotating all the methods defined by the classes of the target program. We call each such method a *target method*. When adding the level I annotations, we assume that ⟨`main`⟩ is not called by any target 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 an exception handler. We also assume that the exception handling is structured such that unexceptional execution can not "fall through" to an exception handler, i.e. the only way an instruction in an exception handler gets executed is if an exception has been raised previously in the execution and caught by the handler that the instruction belongs to.

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 before each update to the specified state, that the specified state is not undefined.

If the execution of a method invocation instruction of a target method may lead to a preaction of the automaton, then an annotation is inserted as a precondition to this instruction, which updates the specified security state. If a method invocation instruction may lead to a postaction, we record the object the method is called on, values of the method arguments (and possibly a part of the heap) by assigning them to ghost variables as the precondition to the instruction. The updates to the specified state are done in the postcondition of the instruction, if the method invocation can lead to a normal (unexceptional) postaction. If the instruction can cause an exceptional postaction, however, the update to the specified security state is inserted as a precondition to the first instruction of each exception handler that cover the instruction. The recorded label is used then at the handler to resolve which instruction has caused the exception, so that the correct update (or no update if the exception was raised by an irrelevant instruction) is performed.

Preliminary Definitions In the definitions below, assume given a program P and a policy $\mathcal{P}$. Let $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$ be the symbolic automaton induced by $\mathcal{P}$, and let $q_s = \{s_1, \ldots, s_n\}$. We define the set $A_s^e \subseteq A_s^\sharp$ of exceptional symbolic post-actions as those post-actions 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). In the annotations, the ghost variables that represent the security state are named identically with the security state variables of the automaton, and we use the tuple $\overrightarrow{gs} = (s_1, \ldots, s_n)$ in guarded multi-assignments. We use the ghost variable g_{pc} to record labels of security relevant instructions. Ghost variables g also used for recording 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 Further below, we define an initializing ghost annotation IA, and for every method M, three arrays of annotations: a pre-annotation array $A^\flat_M[i]$, a post-annotation array $A^\sharp_M[i][j]$, and an exceptional annotation array $A^e_M[i][k]$, 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 ($k=2$). In addition, we define $Exc(L,M)$ as the sequence of all annotations of shape $A^e_M[L'][2]$ where L' is a security relevant instruction such that there exists an exception handler $(L_1,L_2,L,c) \in H_M$ for labels L_1 and L_2 and class name c such that $L_1 \leq L' < L_2$, and as ϵ if such an L' does not exist.

Given these annotations, the *level I ghost annotation* of program P is given for each target method M as a precondition $Requires^{\mathrm{I}}_M$ and an array $A^{\mathrm{I}}_M[i]$ of annotation sequences defined as follows (where $L > 0$):

$$Requires^{\mathrm{I}}_M = \begin{cases} (\overrightarrow{gs} := (Init_s(s_1),\ldots,Init_s(s_n))) \cdot (g_{\mathrm{pc}} := 0) & \text{if } M = \langle \texttt{main} \rangle \\ (g_{\mathrm{pc}} := 0) & \text{otherwise.} \end{cases}$$

$$\begin{aligned} A^{\mathrm{I}}_M[1] &= A^\flat_M[1] \cdot A^\sharp_M[1][0] \cdot A^e_M[1][0] \\ A^{\mathrm{I}}_M[L] &= \begin{cases} Exc(L,M) \cdot A^\flat_M[L] \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0] & \text{if } \exists L_1,L_2,c.(L_1,L_2,L,c) \in H_M \\ A^e_M[L-1][1] \cdot A^\sharp_M[L-1][1] \cdot A^\flat_M[L] \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0] & \text{otherwise} \end{cases} \end{aligned}$$

The initialization annotation $Requires_M$ resets the value of g_{pc} and additionally assigns to the ghost variables the values given by the initialization function $Init_s$ of the automaton if $M = \langle \texttt{main} \rangle$. We assume that the first instruction of a program can not be a handler instruction.

We now define the annotation arrays mentioned in the above definition.

Before Annotations For every method M, the elements of the pre-annotation array $A^\flat_M[i]$ 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 $RS((c,m),A^\flat_s) = \emptyset$, we define the precondition to be empty: $A^\flat_M[L] = \epsilon$.

(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 $RS((c,m),A^\flat_s) = \{c'_1,\ldots,c'_p\}$, then the precondition of the instruction computes the new security state using the arguments and the object of the called method and updates the ghost variables:

$$A^\flat_M[L] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha) \cdot Defined^\flat$$

The assertion $Defined^\flat$ checks if the ghost variables are defined:

$$Defined^\flat = ((s[n] : c'_1 \vee \ldots \vee s[n] : c'_p) \implies (\overrightarrow{gs} \neq \overrightarrow{\bot}))$$

The α_k are the guarded expressions

$$(\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge s[n] : c'_i \wedge\ a_b\rho_i \ \rightarrow\ \overrightarrow{e_E}\rho_i$$

where class c'' defines (c'_i,m) and there exists $a^\flat_s = (c'',m,(\tau_0 x_0,\ldots\tau_{n-1}x_{n-1})) \in A^\flat_s$ such that $(a^\flat_s,b,E) \in \delta^\flat_s$. The substitution ρ_i is defined as $[s[0]/x_0,\ldots,s[n-1]/x_{n-1},s[n]/\texttt{this}]$. Finally, $\alpha = \neg(s[n] : c'_1 \vee \ldots \vee s[n] : c'_p) \ \rightarrow\ \overrightarrow{gs}$.

After Annotations For every method M, the elements of the post-annotation array $A^\sharp_M[i][j]$ 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:

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

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

$$\mathit{Defined}^\sharp = ((g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \implies (\overrightarrow{gs} \neq \overrightarrow{\perp}))$$

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

$$A^\sharp_M[L][1] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha) \cdot (g_{\text{pc}} := 0)$$

where the α_k are the guarded expressions

$$(\overrightarrow{gs} \neq \overrightarrow{\perp}) \wedge g_{\text{this}} : c'_i \wedge a_b\rho_i \ \to\ \overrightarrow{e_E}\rho_i$$

where class c'' defines (c'_i, m) and there exists $a^\sharp_s = (r, c'', m, (\tau_0 x_0, \ldots \tau_{n-1} x_{n-1})) \in A^\sharp_s \setminus A^e_s$ such that $(a^\sharp_s, b, E) \in \delta^\sharp_s$. The substitution ρ_i is defined as $[s[0]/x, g_0/x_0, \ldots g_{n-1}/x_{n-1}, g_{this}/\mathtt{this}]$ if $r = (\tau\, x)$ and as $[g_0/x_0, \ldots g_{n-1}/x_{n-1}, g_{this}/\mathtt{this}]$ if $r = \mathit{void}$. Finally, $\alpha = \neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \ \to\ \overrightarrow{gs}$.

Exceptional Annotations For every method M, the elements of the exceptional annotation array $A^e_M[i][j]$ 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 $RS((c,m), A^e_s) = \emptyset$, we define the pre- and post-conditions to be empty: $A^e_M[L][0] = A^e_M[L][1] = A^e_M[L][1] = \epsilon$.

(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 $RS((c,m), A^e_s) = \{c'_1, \ldots, c'_p\}$, then the precondition of the instruction saves the arguments, the object and the label of the instruction in ghost variables:

$$A^e_M[L][0] = ((g_0, \ldots, g_{n-1}, g_{\text{this}}, g_{\text{pc}}) := (\mathtt{s}[0], \ldots, \mathtt{s}[n]), L) \cdot \mathit{Defined}^e$$

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

$$\mathit{Defined}^e = ((g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \implies (\overrightarrow{gs} \neq \overrightarrow{\perp}))$$

The postcondition of the instruction resets the value of g_{pc} to 0. Notice that this annotation gets executed only if the method invocation did not return with an exception.

$$A^e_M[L][1] = g_{\text{pc}} := 0$$

The precondition of each handler that covers this instruction uses g_{pc} to check whether the exception catched was thrown by a security relevant instruction. If the exception was raised by a method called by the instruction with the relevant label, the annotation uses the saved values to compute the new security state:

$$A^e_M[L][2] = (\overrightarrow{gs} := \alpha_1 \mid \cdots \mid \alpha_m \mid \alpha) \cdot (g_{\text{pc}} := 0)$$

where the α_k are the guarded expressions

$$(g_{\text{pc}} = L) \wedge (\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge g_{\text{this}} : c'_i \wedge \ a_b\rho_i \ \rightarrow \ \overrightarrow{e_E}\rho_i$$

where class c'' defines (c'_i, m) and there exists $a^e_s = (\text{exc}, c'', m, (\tau_0 x_0, \ldots \tau_{n-1} x_{n-1})) \in A^e_s$ such that $(a^e_s, b, E) \in \delta^e_s$. The substitution ρ_i is defined as $[g_0/x_0, \ldots g_{n-1}/x_{n-1}, g_{this}/\texttt{this}]$. Finally, $\alpha = \neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \ \rightarrow \ \overrightarrow{gs}$.

Each execution of a program that is valid with respect to level I annotations for policy $\mathcal{P}$ corresponds to a co-execution of the program and the automaton for $\mathcal{P}$ where the automaton states coincide with the specified security state, hence the program adheres to $\mathcal{P}$.

Theorem 6.2 (Correctness of Level I Annotations)**.** *The level I annotation of program P for policy $\mathcal{P}$ is valid if, and only if, P adheres to $\mathcal{P}$.*

Proof. See Appendix B. □

Example An application annotated with level I annotations for the example policy in section 4 is shown below. The annotations are surrounded by braces {} and placed above the instruction they are associated to. The application creates a new file, asks the user if it can open a connection, after which it opens a connection. Although the user's answer is disregarded when opening the connection, the annotated application is valid since it does not access existing files.

```
public static void main(java.lang.String[]);

{ (a, p) := (false, false)
  g_pc := 0 }
L1   new     GUI
L2   dup
L3   invokespecial   GUI.<init>()V
L4   astore r1
L5   new     File
L6   dup
L7   invokespecial   File.<init>()V
L8   astore r2
L9   aload r2
L10  ldc "Data.txt"
L11  ldc "CreateNew"
L12  ldc "OpenWrite"
{ g_this : File ⇒ (a, p) ≠ ⊥→
  (a, p) := s[3] : File ∧ (s[1] equals "CreateNew") → (a, p) |
            s[3] : File ∧ ¬(s[1] equals "CreateNew") ∧ (s[1] equals "Open") ∧ (s[2] equals "OpenRead") → (a, false) |
            ¬(s[3] : File) → (a, p)
  g_this : File ⇒ (a, p) ≠ ⊥→
  (g1, g2, g3, g_this, g_pc) := (s[0], s[1], s[2], s[3], 12)
  g_this : File ⇒ (a, p) ≠ ⊥→ }
L13  invokevirtual   File.Open(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
{ g_pc := 0 }
L14  aload r1
{ g_this := s[0]
  g_this : GUI ⇒ (a, p) ≠ ⊥→ }
L15  invokevirtual   GUI.AskConnect()Z
{ g_this : GUI ⇒ (a, p) ≠ ⊥→
  (a, p) := g_this : GUI ∧ s[0] → (a, true) |
            g_this : GUI ∧ ¬s[0] → (a, false) |
            ¬(g_this : GUI) → (a, p) }
L16  new Connection
L17  dup
L18  invokespecial   Connection.<init>()V
L19  astore r3
L20  aload r3
L21  ldc "HTTPS"
L22  ldc "www.someurl.com"
{ g_this : Connection ⇒ (a, p) ≠ ⊥→
  (a, p) := s[0] : Connection ∧ (¬a ∨ p) → (a, p) |
            ¬(s[0] : Connection) → (a, p)
  g_this : Connection ⇒ (a, p) ≠ ⊥→ }
L23  invokevirtual   Connection.Open(Ljava/lang/String;Ljava/lang/String;)Z
L24  goto L26
{ (g_this : File ∧ g_pc = 12) ⇒ (a, p) ≠ ⊥→
  (a, p) := g_pc = 12 ∧ g_this : File ∧ false → (a, p) |
            ¬(g_pc = 12) ∨ ¬(g_this : File) → (a, p)
  g_pc := 0 }
L25  astore r4
L26  invokestatic   java/lang/System.exit:(I)V
L27  return
Exception Table:
L1  L24  L25  Class java/lang/Exception
```

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 enough form that it is independent of many design choices a specific inliner may make. In particular, it seems natural to require of an inlined monitor that it maintains agreement between the ghost state and the embedded state immediately prior to execution of a security relevant

action. That is, program and monitor state are both tested and, where necessary, updated whenever a security relevant action is about to be performed. This is by no means a necessary condition: For instance, a monitor implementation may in advance determine that some fixed sequence of security relevant actions is permissible without necessarily reflecting this through an explicit sequence of updates to the embedded state. Thus, in the middle of such a sequence, the embedded state and the ghost state may disagree. In this paper, however, we assume that this type of optimized inlining is not performed.

The second assumption we make in this section is that updates to the embedded state are made locally, by the same method that executes the security relevant method call. This allows correctness to be expressed by asserting equality of the ghost state and the embedded state for every method at point of entry, at normal and exceptional exit, and at each method invocation. This compositionality property has the important advantage that virtual call resolution can be avoided for the level II annotations: The specified and the embedded states are synchronized at all call points, not just at the points where a security relevant action is invoked.

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 method M of the target program at the following points:

1. Each annotation $A(\Gamma^*(M))(i)$ such that $P(\Gamma^*(M))(i)$ is an invoke or a return instruction.

2. The annotation $\mathit{Exsures}(\Gamma^*(M))$.

We explain a sense in which the level II annotations can be argued to characterize exactly the two conditions assumed in this section (the synchronous update assumption, and the method-local update assumption).

Consider a program P with a level II annotated environment Γ^*. Consider an execution $E = C_0C_1\cdots$ from an initial configuration C_0 of P. We sample the embedded state $\vec{ms}$ at all configurations that are either invoke instructions, return instructions, the first instruction of a method, or an unhandled exception. More precisely, the index i is a *sampling point* if one of the following two conditions hold:

1. The top frame of C_i has the shape (M, pc, s, f), and $M[pc]$ is either an `invokevirtual` instruction, a return instruction, or else $pc = 1$.

2. Alternatively, the top frame of C_i is exceptional, of the form $(b)_e$.

We can then construct a sequence $w = (C_0, q_0)(C_1, q_1)\cdots$ (or, $w(E, \overrightarrow{ms})$ if the underlying execution and embedded state needs emphasis) such that:

- q_0 is the initial automaton state,
- for all sampling points $i > 0$, $q_i = C_i(\overrightarrow{ms})$, the value of $\overrightarrow{ms}$ in configuration C_i, and
- for any two consecutive sampling points i and i', for all $j : i \leq j < i'$, $q_j = q_i$.

In other words, the embedded state is sampled at the sampling points and maintained constant inbetween.

The role of the sequence w is roughly similar to the role of interleavings in section **??**. However, a slightly different treatment is needed here since the sequence $q_0q_1\cdots$ may not

necessarily correspond to an automaton derivation. This is so for the case of post-actions followed by pre-actions where the intermediate automaton state is not sampled, as there is no well-defined point where this might be done. Also, the construction needs to account for the method-local nature of embedded state updates.

For this reason define the operation $extract_{II}$ taking sequences w to strings over the alphabet $Q \cup A \cup \{\texttt{brk}\}$ where `brk` is a distinguished symbol by the following conditions:

- $extract_{II}((C_1, q_1)(C_2, q_2)w) = q_1\ act^\flat(C_1)\ act^\sharp(C_1, C_2)q_2\ extract_{II}((c_2, q_2)w)$, if C_1 is an API method call.
- $extract_{II}((C, q)w) = \texttt{brk}q\ extract_{II}(w)$, if C is entering to or returning from an application method call.
- $extract_{II}((C, q)) = q\ act^\flat(C)$ /marginparonly if API call
- $extract_{II}(\varepsilon) = \varepsilon$

Definition 6.3 (Method-local Co-execution)**.** *Let* $\Sigma_1 = \{\texttt{brk}\} \cup Q \cup E^\flat \cup E^\sharp \cup \{a^\sharp qq' a^\flat \mid q, q' \in Q, a^\flat \in A^\flat, a^\sharp \in A^\sharp, \exists q''.\delta^\flat(q, a^\flat) = q'', \delta^\sharp(q'', a^\sharp) = q'\}$*, and* $\Sigma_2(q) = \{q\texttt{brk}q\} \cup \{qq\} \cup \{qaq \mid a \in A\} \cup \{(qa^\flat a^\sharp q) \mid a^\flat \in A^\flat, a^\sharp \in A^\sharp\}$*. A sequence* w *is a method-local co-execution, if* $extract_{II}(w) \in (\Sigma_1^* \cup \Sigma_1^\omega) \cap ((\bigcup_{q \in Q} \Sigma_2(q))^* \cup (\bigcup_{q \in Q} \Sigma_2(q))^\omega)$ *.*

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

Theorem 6.4 (Level II Characterization)**.** *The level II annotation of* P *with embedded state* $\overrightarrow{ms}$ *is valid if, and only if, for each execution* E *of* P*, the sequence* $w(E, \overrightarrow{ms})$ *is a method-local co-execution.*

The idea of the proof is to sample pre- and post-actions from E, immediately preceded and followed by a sample of the embedded state $\overrightarrow{ms}$. The sequence extracted in this way is almost a potential derivation, but in the case of a post-action followed, some time later, by a pre-action, an intermediate automaton state may be missing. It is not clear, however, how to sample this state. Also, it is necessary to ensure that embedded state updates do not cross method boundaries. To this end, extracted sequences need to be completed by (a) missing intermediate automaton states, and (b) indicators of method boundary crossings at: method invocations that are not security relevant actions, return instructions, configurations that contain an exceptional frame at the top of the frame stack, and at the first instruction of each method.

First, we note that the embedded state $\overrightarrow{ms}$ is equal to the ghost state $\overrightarrow{gs}$ at sampling points if and only if the synchronisation assertions added at level II hold. We show in the proof of theorem 2 that the ghost state and machine configurations consitute a co-execution if and only if the program annotated with level I annotations is valid. If the program annotated with level II annotations is valid then the sampling of the embedded stated as described above amounts to taking the co-execution of the ghost state and the program and "skipping" some ghost updates, which the embedded state does not follow (as the sampling of the embedded state is not done as frequently). Then $extract_{II}$ applied to this sequence falls in the set stated in def 6.3.

7 Correctness of a Simple Inliner

As an application of the annotation scheme described in the previous section, we prove the correctness of a simple inliner in the flavor of PoET/PSLang [8, 7]. The inliner inputs a ConSpec policy and a program, and inserts code for (i) storing the security state and (ii) for updating it according to the policy clauses at calls to security relevant methods. We describe the inlining process briefly here, in enough detail for the reader to get an intuition for the proof of correctness. The implementation of the the tool is available at [1].

7.1 The Inliner

Storing the Security State The inliner adds a class definition to the program and stores the security state in its static variables. We refer to this as the *inlined security state*. Since this new class is not in the previous name space, the security state is safe from interference by the target program. In this text, we assume that `SecState` is a fresh name for the untrusted program, and use this name for the class storing the security state.

Compiling Policy Body to Bytecode The first part of the transformation consists of compiling the policy to bytecode and is independent of the method(s) to be inlined. For each clause in the policy, we produce a bytecode fragment, which is to be inlined in the program text in the rewriting stage.

Each clause of a policy consists of an event modifier, an event specification and a list of guarded commands. The code created evaluates, in turn, the guards 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 variables that occur in an event clause are of three kinds: security state variables, method arguments and fields of method arguments. The three kinds of variables are stored at different parts of the JVM, hence the compilation of an event clause produces different code to access them:

1. Security state variables are stored in the the `SecState` class. Since all fields of the class are static, they are created as soon as the class is loaded to the JVM. (Notice that the only variables that get updated are security state variables)

2. Arguments can be obtained by accessing the stack immediately before the method invocation. In order to use argument values while computing the new values of the security state variables, we copy them from the top of the stack and store them in local variables (that are not used by the original program). Since local variables of the calling method are not affected by the method invocations argument values stored in this manner can even be accessed right after the control returns to the calling method.

3. Fields of arguments are accessed using the arguments and heap.

Rewriting Methods According to Policy The inlining 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 the policy compilation in an appropriate manner. The inlined code is depicted for a single instruction in fig. 2. The inliner inserts, immediately before the security relevant instruction, code that records the object the method is called for, 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.

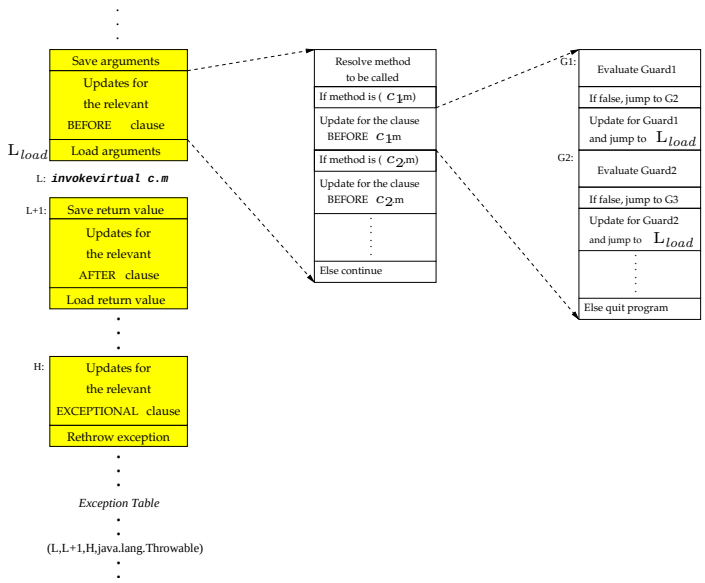

Figure 2: Inlining of an Instruction

The object and the method arguments are restored on stack immediately before the security relevant instruction. If there are AFTER clauses in the policy for the instruction, first the return value (if there is 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.

Method Resolution 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[2].

7.2 Correctness of the Inliner

We first describe how, for programs inlined with the inliner described above, level II annotations can be efficiently completed to an (equivalent) level III annotation, and then show that validity of level III annotations – and thus policy adherence – holds for such programs and is efficiently checkable.

Level III annotation uses the weakest precondition function $wp(M[L])$ defined in [3]. Table 2 contains the definition of the function for the instructions occurring in the examples. The function $shift(A)$ denotes the substitution, for all i, of s[i] by s[i + 1] in assertion A, while function $unshift(A)$ denotes the inverse function. The last two rows of the table refer

[2]This can be accomplished using a call to the `Object.getClass()` method followed by a call to the method `Class.getName()` of the `java.lang` API. Else Reflect API can be used.

$M[L]$	$wp(M[L])$
`dup`	$unshift((head(A_M[L+1]))[\mathrm{s}[1]/\mathrm{s}[0]])$
`ldc v`	$unshift((head(A_M[L+1]))[\mathtt{v}/\mathrm{s}[0]])$
`iload r / \verb`aload r+	$unshift((head(A_M[L+1]))[\mathtt{r}/\mathrm{s}[0]])$
`aconst_null`	$unshift((head(A_M[L+1]))[\mathtt{null}/\mathrm{s}[0]])$
`istore r / \verb`astore r+	$(shift(head(A_M[L+1])))[\mathrm{s}[0]/\mathtt{r}]$
`putstatic m`	$(shift(head(A_M[L+1])))[\mathrm{s}[0]/\mathtt{m}]$
`goto` L'	$head(A_M[L'])$
`ifeq` L'	$(\mathrm{s}[0]=0 \Rightarrow shift(head(A_M[L'])))\wedge$ $(\neg(\mathrm{s}[0]=0) \Rightarrow shift(head(A_M[L+1])))$
`invokevirtual` $c.m$	$(shift^n(head(A_M[L+1])))[f_{c.m}(\mathrm{s}[0],\ldots,\mathrm{s}[\mathrm{n}])/\mathrm{s}[\mathrm{n}]]$
`invokestatic` $c.m$	$(shift^{n-1}(head(A_M[L+1])))[f_{c.m}(\mathrm{s}[0],\ldots,\mathrm{s}[\mathrm{n}-1])/\mathrm{s}[\mathrm{n}-1]]$

Table 2: Weakest precondition function $wp(M[L])$

only to calls to API methods used by the inliner - these are side-effect free and therefore treated as atomic operations. In both rows, n denotes the arity of method $c.m$, and $f_{c.m}$ denotes the operation implemented by method $c.m$ (which is of arity $n+1$ in the case of `invokevirtual`, with the reference to the object as an implicit argument).

Level III annotation also uses a normalizing function *norm* on annotations, with the combined effect of conjuncting consecutive logical assertions and backward weakest precondition propagation:

$$\begin{aligned} norm(\alpha) &= \alpha \\ norm(\gamma \cdot \alpha_0 \cdot \alpha_1) &= norm(\gamma \cdot (\alpha_0 \wedge \alpha_1)) \\ norm(\gamma \cdot (\overrightarrow{g} := ce) \cdot \alpha) &= norm(\gamma \cdot \alpha[ce/\overrightarrow{g}]) \cdot (\overrightarrow{g} := ce) \cdot \alpha \end{aligned}$$

where α, α_0, and α_1 range over logical assertions, γ over annotation sequences, and where $\alpha[ce/\overrightarrow{g}]$ denotes the substitution in α of each ghost variable $g_i \in \overrightarrow{g}$ by the conditional expression ce_i, obtained from ce by replacing each expression vector $\overrightarrow{e_E}$ occurring in ce with its i-th component. The function *head* returns the first element of the annotation sequence.

Level III Annotation A level III (or "full") annotation is obtained as follows.

1. *Requires*$(\Gamma^*(M))$, *Ensures*$(\Gamma^*(M))$ and *Exsures*$(\Gamma^*(M))$ are all defined as the synchronisation assertion.

2. For all non-inlined instructions $M[L]$, not (level II) annotated with the synchronisation assertion, define

$$A^{III}_M[L] \;=\; norm(A^{II}_M[L] \cdot (\overrightarrow{gs} = \overrightarrow{ms}))$$

3. For all (non-inlined) potentially post-security relevant instructions $M[L]$, define

$$A^{III}_M[L] \;=\; norm(A^{II}_M[L] \cdot (g_0 = r_0) \cdot \ldots \cdot (g_{n-1} = r_{n-1}) \cdot (g_{this} = r_{this}))$$

where $r_0, \ldots, r_{n-1}, r_{this}$ are the local variables used by the inliner to store the values of the parameters and the reference to the object with which the method is invoked.

4. For all remaining non-inlined instructions $M[L]$, define

$$A_M^{III}[L] \;=\; norm(A_M^{II}[L])$$

5. For all blocks of inlined code, we apply the weakest precondition function $wp(M[L])$ defined in Table 2 to propagate backwards the head assertion of the first instruction following the block (which is the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$). Notice that these blocks are cycle-free and do not contain jumps to any other instruction outside of the block, thus the backward *wp*-propagation is well-defined (and in effect computes the weakest precondition of the whole block w.r.t. the synchronisation assertion).

   Thus, if $M[L]$ is an inlined instruction immediately following a potential (nonexceptional) post-security relevant instruction or the first instruction of a handler for a potential (exceptional) post-security relevant instruction, define

$$A_M^{III}[L] \;=\; norm((g_0 = r_0) \cdot \ldots \cdot (g_{n-1} = r_{n-1}) \cdot (g_{this} = r_{this}) \cdot A_M^{II}[L] \cdot wp(M[L]))$$

   and otherwise define

$$A_M^{III}[L] \;=\; wp(M[L])$$

The following result uses the full annotation to show that programs inlined with the inliner described above contain a monitor as characterised by Theorem 6.4. In the result, local validity refers to logical validity of the verification conditions resulting from a fully annotated program.

Theorem 7.1. *Let P be a program, $\mathcal{P}$ a ConSpec policy, and I(P,$\mathcal{P}$) denote program P inlined for policy $\mathcal{P}$. The level III annotation of I(P,$\mathcal{P}$) is locally valid, and validity is efficiently checkable.*

Proof. See Appendix B. □

If the level III annotation of I(P,$\mathcal{P}$) is locally valid, then it is also valid in terms of Definition 6.1. Hence, by the above result, the inlined program I(P,$\mathcal{P}$) is valid with respect to the level I annotation for policy $\mathcal{P}$, and therefore, by Theorem 6.2, adheres to the policy.

Corollary 7.2 (Correctness of Inlining)**.** *Let $\mathcal{P}$ be a ConSpec policy and P be a program. The inlined program I(P,$\mathcal{P}$) adheres to the policy.*

Notice that 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 report 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 the program adheres to a given policy, and (ii) that the program has an embedded method-compositional monitor for this policy. The annotation scheme thus characterises a whole class of monitor inliners. As an application, we present a concrete inliner and show its correctness. For this inliner, validity of the annotations can be decided efficiently using a standard wp 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 and J. Linde. An inliner tool for mobile platforms. http://www.csc.kth.se/∼irem/S3MS/Inliner/.

[2] I. Aktug and K. Naliuka. ConSpec – a formal language for policy specification. In F. Piessens and F. Massacci, editors, *to appear in Proc. of The First International Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, Electronic Notes in Theoretical Computer Science, 2007.

[3] F. Y. Bannwart and P. Müller. A logic for bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141-1 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[4] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, June 2005.

[5] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proc. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '00)*, pages 54–66, New York, NY, USA, 2000. ACM Press.

[6] D. Drusinsky. The temporal rover and the atg rover. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 323–330, London, UK, 2000. Springer-Verlag.

[7] Ú. Erlingsson. *The inlined reference monitor approach to security policy enforcement.* PhD thesis, Dep. of Computer Science, Cornell University, 2004.

[8] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, page 0246, New York, NY, USA, 2000. IEEE Computer Society.

[9] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: a retrospective. In *Proc. Workshop on New Security Paradigms (NSPW '99)*, pages 87–95, New York, NY, USA, 2000. ACM Press.

[10] 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.

[11] 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)*, June 2006.

[12] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Computability classes for enforcement mechanisms. *ACM Trans. Program. Lang. Syst.*, 28(1):175–205, 2006.

[13] 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.

[14] C. Jeffery, W. Zhou, K. Templer, and M. Brazell. A lightweight architecture for program execution monitoring. In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '98)*, pages 67–74, New York, NY, USA, 1998. ACM Press.

[15] M. Kim, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: a run-time assurance tool for Java. In *1st International Workshop on Run-time Verification*, volume 55, July 2001.

[16] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswantathan. Computational analysis of run-time monitoring: Fundamentals of Java-MaC. In *Proc. of the Second International Workshop on Runtime Verification (RV 2002)*, volume 70, December 2002.

[17] X. Leroy. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):235–269, 2003.

[18] J. Ligatti, L. Bauer, and D. Walker. Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16, February 2005. (Published online 26 Oct 2004.).

[19] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proceedings of the 10th European Symposium on Research in Computer Security (ESORICS)*, September 2005.

[20] T. Rezk. *Verification of Confidentiality Policies for Mobile Code*. PhD thesis, INRIA Sophia Antipolis and University of Nice Sophia Antipolis, November 2006.

[21] F. B. Schneider. Enforceable security policies. *ACM Trans. Infinite Systems Security*, 3(1):30–50, 2000.

A Examples

1. Java source code The program contains the Ask class has a single field called gui, which is of type GUI. The run method of this class calls GUI.AskConnect() on the object stored in the gui field. It returns true if this method returns "Yes", it returns false otherwise.

```
public class Report {
    public static void main(String[] args) {
        GUI g= new GUI();
        File f= new File();
        f.Open("Data.txt", "Open", "OpenRead");
        Ask a= new Ask(g);
        a.run();
        Connection c = new Connection();
        c.Open("HTTPS", "www.someurl.com");
    }

}

public class Ask {
   GUI gui;
   public void Ask(GUI g) {
      gui = g;
   }
   public bool run() {
      return gui.AskUser();
   }
 }
```

2. Policy The policy is the one from section 4.

3. Java bytecode

```
      public class Ask extends java/lang/Object

      GUI gui;

      public <init> (GUI)V
L1    aload r0
L2    invokespecial    java/lang/Object/<init>()V
L3    aload r0
L4    aload r1
L5    putfield gui
L6    return

      public run ()Z
L1    aload r0
L2    getfield gui
L3    invokevirtual GUI/AskConnect()Z
L4    ireturn
```

```
      public class Report extends java/lang/Object

      public static void main(java.lang.String[]);
 L1   new GUI
 L2   dup
 L3   invokespecial   GUI.<init>()V
 L4   astore r1
 L5   new File
 L6   dup
 L7   invokespecial   File.<init>()V
 L8   astore r2
 L9   ldc "Data.txt"
L10   ldc "Open"
L11   ldc "OpenRead"
L12   invokevirtual File.Open(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
L13   new Ask
L14   dup
L15   aload r1
L16   invokespecial   Ask.<init>(LGUI)V
L17   invokevirtual Ask.run()Z
L18   new Connection
L19   dup
L20   invokespecial    Connection.<init>()V
L21   astore r3
L22   aload r3
L23   ldc "HTTPS"
L24   ldc "www.someurl.com"
L25   invokevirtual Connection.Open(Ljava/lang/String;Ljava/lang/String;)Z
L26   return
```

4. Inlined Java bytecode

```
      public class Ask extends java/lang/Object

      GUI gui;

      public <init> (GUI)V
 L1   aload  r0
 L2   invokespecial    java/lang/Object/<init>()V
 L3   aload  r0
 L4   aload  r1
 L5   putfield gui
 L6   return

      public run ()Z
 L1   aload r0
 L2   getfield gui
      // begin of inlined code //
 L3   dup
 L4   astore r1 // saves object in reg r1
      // end of inlined code //
 L5   invokevirtual    GUI/AskConnect()Z
      // begin of inlined code //
 L6   istore r2 // saves the return value in reg r2
 L7   aload r1 // load object to stack
      // get class of object
 L8   invokevirtual java/lang/Object/getClass()Ljava/lang/Class
 L9   aconst_null // load null (pointer to array of ...)
L10   ldc "AskUser"
      // Returns actual method
L11   invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)
      Ljava/lang/reflect/Method
      // Returns declaring class of method
L12   invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
L13   ldc "GUI"
      // Load class of GUI
L14   invokestatic java/lang/Class/forName(Ljava/lang/String) Ljava/lang/Class
      // Compare class of GUI with class of called method
L15   invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
L16   ifeq L24 // if not equal then not security relevant;
L17   iload r2 // otherwise load the return value from r2, and
L18   ifeq L22 // update security state accordingly
L19   ldc 1
L20   putstatic SecState/permission
L21   goto L24
L22   ldc 0
L23   putstatic SecState/permission
L24   iload r2 // load the return value on stack to return
      // end of inlined code //
L25   ireturn
```

```
        public class Report extends java/lang/Object

        public static void main(java.lang.String[]);
 L0     new     GUI
 L1     dup
 L2     invokespecial   GUI.<init>()V
 L3     astore r1
 L4     new     File
 L5     dup
 L6     invokespecial   File.<init>()V
 L7     astore r2
 L8     ldc "Data.txt"
 L9     ldc "Open"
L10     ldc "OpenRead"
        // begin of inlined code //
L12     astore r4 //saves argument in reg4
L13     astore r5 // saves argument in reg5
L14     astore r6 //saves argument in reg6
L15     astore r7 // saves object in reg7
        // get class of object
L16     aload r7 // loads object to stack+
L17     invokevirtual java/lang/Object/getClass()Ljava/lang/Class
L18     ldc "Open"
L19     iconst_3
L20     anewarray   java/lang/Class
L21     dup
L22     dup
L23     dup
L24     iconst_0
L25     ldc    java/lang/String
L26     aastore
L27     iconst_1
L28     ldc     java/lang/String
L29     aastore
L30     iconst_2
L31     ldc     java/lang/String
L32     aastore
        // Returns actual method
L33     invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method
        // Returns declaring class of method
L34     invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
        //Load class of File
L35     ldc "File"
L36     invokevirtual java/lang/Class/forName(Ljava/lang/String)Ljava/lang/Class
        // Compare File with class of method called
L37     invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
L38     ifne L1' //if not equal then not security relevant
L39     aload r5
L40     ldc "CreateNew"
L36     invokevirtual java/lang/String/equals(Ljava/lang/Object;)Z
L36     ifneq G2'
        goto L1'
G2':    aload r5
        ldc "Open"
        ifne ABORT1
        invokevirtual java/lang/String/equals(Ljava/lang/Object;)Z
        aload r6
        ldc "OpenRead"
        ifne ABORT1
        ldc 1
        putstatic SecState/accessed
        goto L1'
```

```
ABORT1:  iconst_1
         invokestatic java/lang/System.exit:(I)V // (ABORT)
   L1'   aload r7 //load object to stack
         aload r6 //load argument to stack
         aload r5 //load argument to stack
         aload r4 //load argument to stack
         // end of inlined code //
     L   invokevirtual File.Open(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
         //beginning of inlined code //
         goto AFTERL
     H   aload r7 // loads object to stack
         invokevirtual java/lang/Object/getClass()Ljava/lang/Class
         ldc "Open"
         iconst_3
         anewarray java/lang/Class
         dup
         dup
         dup
         iconst_0
         ldc java/lang/String
         aastore
         iconst_1
         ldc java/lang/String
         aastore
         iconst_2
         ldc java/lang/String
         aastore
         // Returns actual method
         invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method
         // Returns declaring class of method
         invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
         //Load class of File
         ldc "File"
         invokevirtual java/lang/Class/forName(Ljava/lang/String)
         Ljava/lang/Class
         // Compare File with class of method called
         invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
         ifne HL1' //if not equal then not security relevant
         iconst_1
         invokestatic java/lang/System.exit:(I)V // (ABORT)
         //Rethrow exception
  HL1'   athrow
         // end of inlined code
AFTERL   new Ask
         dup
         aload r1
         invokespecial Ask.¡init¿(LGUI)V
         invokevirtual Ask.run()Z
         new Connection
         dup
         invokespecial Connection.¡init¿()V
         astore r3
         aload r3
         ldc "HTTPS"
         ldc "www.someurl.com"
         astore r4 //saves argument in reg4
         astore r5 // saves argument in reg5
         astore r6 //saves object in reg6
         // get class of object
         aload r6 // loads object to stack
         invokevirtual java/lang/Object/getClass()Ljava/lang/Class
         ldc "Open"
```

```
        iconst_2
        anewarray java/lang/Class
        dup
        dup
        iconst_0
        ldc java/lang/String
        aastore
        iconst_1
        ldc java/lang/String
        aastore
        // Returns actual method
        invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method
        // Returns declaring class of method
        invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
        //Load class of Connection
        ldc "Connection"
        invokevirtual java/lang/Class/forName(Ljava/lang/String) Ljava/lang/Class
        // Compare Connection with class of method called
        invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
        ifne L4' //if not equal then not security relevant
        getstatic SecState/accessed
        ifeq L5'
        getstatic SecState/permission
        ifne L6'
L5'     ldc 0
        putstatic SecState/permission
        goto L4'
L6'     iconst_1
        invokestatic java/lang/System.exit:(I)V // (ABORT)
L4'     aload r6 //load object to stack
        aload r5 //load argument to stack
        aload r4 //load argument to stack
        // end of inlined code //
        invokevirtual Connection.Open(Ljava/lang/String;Ljava/lang/String;)Z
        return
```

Exception Table:

L L+1 H any

5. Level II Annotated Inlined Java bytecode

```
      public class Ask extends java/lang/Object

      GUI gui;

      public <init> (GUI)V
L1    aload  r0
L2    invokespecial    java/lang/Object/<init>()V
L3    aload  r0
L4    aload  r1
L5    putfield gui
L6    return
```

Requires = ((a, p) = (SecState.accessed, SecState.permission))
Ensures = ϵ
Exsures = ((a, p) = (SecState.accessed, SecState.permission))

```
      public run ()Z
```
$\{g_{\text{pc}} := 0\}$
```
L1    aload r0
L2    getfield gui
      // begin of inlined code //
L3    dup
L4    astore r1 // saves object in reg r1
      // end of inlined code //
```
$$\left\{ \begin{array}{lcl} g_{\text{this}} & := & \text{s}[0] \\ g_{\text{this}} & : & \texttt{GUI} \wedge (\text{a, p}) \neq \overrightarrow{\perp} \\ (\text{a, p}) & = & (\texttt{SecState.accessed}, \texttt{SecState.permission}) \end{array} \right\}$$
```
L5    invokevirtual    GUI/AskConnect()Z
```
$$\left\{ \begin{array}{lcll} g_{\text{this}} & : & \texttt{GUI} \wedge (\text{a, p}) \neq \overrightarrow{\perp} & \\ (\text{a, p}) & := & g_{\text{this}} : \texttt{GUI} \wedge \text{s}[0] \rightarrow (\text{a}, \text{true}) & | \\ & & g_{\text{this}} : \texttt{GUI} \wedge \neg\text{s}[0] \rightarrow (\text{a}, \text{false}) & | \\ & & \neg(g_{\text{this}} : \texttt{GUI}) \rightarrow (\text{a}, \text{p}) & \end{array} \right\}$$
```
      // begin of inlined code //
L6    istore r2  // saves the return value in reg r2
L7    aload r1  // load object to stack
      // get class of object
L8    invokevirtual java/lang/Object/getClass()Ljava/lang/Class
L9    aconst_null  // load null (pointer to array of ...)
L10   ldc "AskUser"
      // Returns actual method
L11   invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)
      Ljava/lang/reflect/Method
      // Returns declaring class of method
L12   invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
L13   ldc "GUI"
      // Load class of GUI
L14   invokestatic java/lang/Class/forName(Ljava/lang/String) Ljava/lang/Class
      // Compare class of GUI with class of called method
L15   invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
L16   ifeq L24  // if not equal then not security relevant;
L17   iload r2  // otherwise load the return value from r2, and
L18   ifeq L22 // update security state accordingly
L19   ldc 1
L20   putstatic SecState/permission
L21   goto L24
L22   ldc 0
L23   putstatic SecState/permission
L24   iload r2 // load the return value on stack to return
      // end of inlined code //
```
$\{$ (a, p) = (SecState.accessed, SecState.permission) $\}$
```
L25   ireturn
```

```
        public class Report extends java/lang/Object

        public static void main(java.lang.String[]);
 L0     new     GUI
 L1     dup
 L2     invokespecial   GUI.<init>()V
 L3     astore r1
 L4     new     File
 L5     dup
 L6     invokespecial   File.<init>()V
 L7     astore r2
 L8     ldc "Data.txt"
 L9     ldc "Open"
L10     ldc "OpenRead"
        // begin of inlined code //
L12     astore r4 //saves argument in reg4
L13     astore r5 // saves argument in reg5
L14     astore r6 //saves argument in reg6
L15     astore r7 // saves object in reg7
        // get class of object
L16     aload r7 // loads object to stack+
L17     invokevirtual java/lang/Object/getClass()Ljava/lang/Class
L18     ldc "Open"
L19     iconst_3
L20     anewarray   java/lang/Class
L21     dup
L22     dup
L23     dup
L24     iconst_0
L25     ldc    java/lang/String
L26     aastore
L27     iconst_1
L28     ldc     java/lang/String
L29     aastore
L30     iconst_2
L31     ldc     java/lang/String
L32     aastore
        // Returns actual method
L33     invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method
        // Returns declaring class of method
L34     invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
        //Load class of File
L35     ldc "File"
L36     invokevirtual java/lang/Class/forName(Ljava/lang/String)Ljava/lang/Class
        // Compare File with class of method called
L37     invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
L38     ifne L1' //if not equal then not security relevant
L39     aload r5
L40     ldc "CreateNew"
L36     invokevirtual java/lang/String/equals(Ljava/lang/Object;)Z
L36     ifneq G2'
        goto L1'
G2':    aload r5
        ldc "Open"
        ifne ABORT1
        invokevirtual java/lang/String/equals(Ljava/lang/Object;)Z
        aload r6
        ldc "OpenRead"
        ifne ABORT1
        ldc 1
        putstatic SecState/accessed
        goto L1'
```

ABORT1: iconst_1
invokestatic java/lang/System.exit:(I)V // (ABORT)
L1' aload r7 //load object to stack
aload r6 //load argument to stack
aload r5 //load argument to stack
aload r4 //load argument to stack
// end of inlined code //
$\left\{ \begin{array}{l} Defined^{\flat} \cdot ((a,p) := \ldots) \cdot Defined^{\flat} \\ \cdot(g_0, g_1, g_2, g_{\text{this}}, g_{\text{pc}}) := (s[0], s[1], s[2], s[3], L) \cdot Defined^{e} \cdot (\overrightarrow{gs} = \overrightarrow{ms}) \end{array} \right\}$
L invokevirtual File.Open(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)Z
$\{g_{\text{pc}} := 0\}$
//beginning of inlined code //
goto AFTERL
$\left\{ \begin{array}{l} ((g_{\text{pc}} = L) \Rightarrow Defined^{e}) \cdot ((a,p) := \ldots) \cdot (\overrightarrow{gs} = \overrightarrow{ms}) \\ \cdot(g_0, g_1, g_2, g_{\text{this}}, g_{\text{pc}}) := (s[0], s[1], s[2], s[3], L) \cdot Defined^{e} \cdot (\overrightarrow{gs} = \overrightarrow{ms}) \end{array} \right\}$
H aload r7 // loads object to stack
invokevirtual java/lang/Object/getClass()Ljava/lang/Class
ldc "Open"
iconst_3
anewarray java/lang/Class
dup
dup
dup
iconst_0
ldc java/lang/String
aastore
iconst_1
ldc java/lang/String
aastore
iconst_2
ldc java/lang/String
aastore
// Returns actual method
invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method
// Returns declaring class of method
invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
//Load class of File
ldc "File"
invokevirtual java/lang/Class/forName(Ljava/lang/String)
Ljava/lang/Class
// Compare File with class of method called
invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
ifne HL1' //if not equal then not security relevant
iconst_1
invokestatic java/lang/System.exit:(I)V // (ABORT)
//Rethrow exception
HL1' athrow
// end of inlined code
AFTERL new Ask
dup
aload r1
invokespecial Ask.¡init¿(LGUI)V
invokevirtual Ask.run()Z
new Connection
dup
invokespecial Connection.¡init¿()V
astore r3
aload r3
ldc "HTTPS"
ldc "www.someurl.com"
astore r4 //saves argument in reg4
astore r5 // saves argument in reg5
astore r6 //saves object in reg6
// get class of object

```
        aload r6 // loads object to stack
        invokevirtual java/lang/Object/getClass()Ljava/lang/Class
        ldc "Open"
        iconst_2
        anewarray java/lang/Class
        dup
        dup
        iconst_0
        ldc java/lang/String
        aastore
        iconst_1
        ldc java/lang/String
        aastore
        // Returns actual method
        invokevirtual java/lang/Class/getMethod(Ljava/lang/String;[Ljava/lang/Class;)Ljava/lang/reflect/Method
        // Returns declaring class of method
        invokevirtual java/lang/reflect/Method/getDeclaringClass()Ljava/lang/Class
        //Load class of Connection
        ldc "Connection"
        invokevirtual java/lang/Class/forName(Ljava/lang/String) Ljava/lang/Class
        // Compare Connection with class of method called
        invokevirtual java/lang/Object/equals(Ljava/lang/Object;)Z
        ifne L4' //if not equal then not security relevant
        getstatic SecState/accessed
        ifeq L5'
        getstatic SecState/permission
        ifne L6'
L5'     ldc 0
        putstatic SecState/permission
        goto L4'
L6'     iconst_1
        invokestatic java/lang/System.exit:(I)V // (ABORT)
L4'     aload r6 //load object to stack
        aload r5 //load argument to stack
        aload r4 //load argument to stack
        // end of inlined code //
        invokevirtual Connection.Open(Ljava/lang/String;Ljava/lang/String;)Z
        return
```

Exception Table:

L	L+1	H	any

6. Level III Annotated Inlined Java bytecode

In this example, we will be using the four functions *strclass*, *objclass* and *meth*, and *declass*, for modeling the semantics of the Reflect library methods that occur in the inlined code. The reflect library is used to determine the declaring class of a method called in case the method may be security relevant. The first function *strclass* $:\!\|\ \texttt{string}\ \|\!\rightarrow\!\|\ Class\ \|$ returns for each string c which is a class name, the special object of type $Class$. Such an object exists for each defined class name. Similarly, *objclass* $: \ell \rightharpoonup \|\ Class\ \|$ returns for each object of the heap the type of the object as an object of type $Class$. The function *meth* $:\!\|\ \texttt{string}\ \|\ \times (Val)^* \times Class \rightarrow\!\|\ Method\ \|$ returns, for a method name, the list of its argument types and the object the method is to be called on, an object of the $Method$ class. In turn, the function *declass* $:\!\|\ Method\ \|\!\rightharpoonup\!\|\ Class\ \|$ returns the declaring class of an object of the $Method$ class.

The calls to the library functions are in a certain order so that the wp of this block contains the expression:

$$strclass(c) = declass(meth(m, a, objclass(r)))$$

where c is a class name, m is the method name, r is the object the method is called on, and a is the address of an array which stores the list of the argument types. Since we do not consider overloading, we assume that a is $\Gamma(c, m)$. Then we can use the following equivalence:

$$strclass(c) = declass(meth(m, a, objclass(r))) \Leftrightarrow \bigvee_{c' \in S_c} r : c'$$

where $S_c = \{c' \mid c \text{ defines } (c', m)\}$.

Annotations of $L1 - L4$ are computed using rule 1.

$$
\begin{array}{lcl}
A^{III}[L1] & = & \mathit{norm}(g_{\mathrm{pc}} := 0 \cdot (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \\
& = & ((\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \\
& \cdot & (g_{\mathrm{pc}} := 0) \\
& \cdot & ((\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}) \\
A^{III}[L2] = A^{III}[L3] & = & A^{III}[L4] \\
& = & \mathit{norm}(\epsilon \cdot (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))
\end{array}
$$

Annotation of $L5$ is computed using rule 2.

$$
\begin{array}{lcl}
A^{III}[L5] & = & \mathit{norm}((g_{\mathrm{this}} := s[0]) \cdot (g_{\mathrm{this}} : \texttt{GUI} \Rightarrow (\mathrm{a},\ \mathrm{p}) \neq \overrightarrow{\perp}) \\
& \cdot & ((\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \cdot (g_{\mathrm{this}} = r1)) \\
& = & ((s[0] : \texttt{GUI} \Rightarrow (\mathrm{a},\ \mathrm{p}) \neq \overrightarrow{\perp}) \wedge ((\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \wedge (s[0] = r1)) \\
& \cdot & (g_{\mathrm{this}} := s[0]) \\
& \cdot & ((g_{\mathrm{this}} : \texttt{GUI} \Rightarrow (\mathrm{a},\ \mathrm{p}) \neq \overrightarrow{\perp}) \wedge ((\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \wedge (g_{\mathrm{this}} = r1))
\end{array}
$$

Annotation of $L25$ (used for computing the annotations of $L6 - L24$) is computed using rule 3.

$$
\begin{array}{lcl}
A^{III}[L25] & = & \mathit{norm}((\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})) \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})
\end{array}
$$

Annotations of $L6 - L24$ are computed using rule 4 using wp computation.

$$
\begin{array}{lcl}
A^{III}[L23] & = & \mathit{wp}(M[23]) \\
& = & \mathit{shift}(\mathit{head}(A^{I}II_{M}[L24]))[s[0]/\texttt{SecState.permission}] \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})[s[0]/\texttt{SecState.permission}] \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, s[0]) \\
\\
A^{III}[L22] & = & \mathit{wp}(M[22]) \\
& = & \mathit{unshift}(\mathit{head}(A^{I}II_{M}[L23])[0/s[0]]) \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 0) \\
\\
A^{III}[L21] & = & \mathit{wp}(M[21]) \\
& = & \mathit{head}(A^{I}II_{M}[L22]) \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}) \\
\\
A^{III}[L20] & = & \mathit{wp}(M[20]) \\
& = & (\mathit{shift}(\mathit{head}(A^{I}II_{M}[L21])))[s[0]/m] \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, s[0]) \\
\\
A^{III}[L19] & = & \mathit{wp}(M[19]) \\
& = & \mathit{unshift}(\mathit{head}(A^{I}II_{M}[L20])[1/s[0]]) \\
& = & (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 1) \\
\\
A^{III}[L18] & = & \mathit{wp}(M[18]) \\
& = & (s[0] = 0 \Rightarrow \mathit{shift}(\mathit{head}(A^{I}II_{M}[L22])))\wedge \\
& & (\neg(s[0] = 0) \Rightarrow \mathit{shift}(\mathit{head}(A^{I}II_{M}[19]))) \\
& = & (s[0] = 0 \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(s[0] = 0) \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 1)) \\
\\
A^{III}[L17] & = & \mathit{wp}(M[17]) \\
& = & \mathit{unshift}(\mathit{head}(A^{I}II_{M}[18])[r2/s[0]]) \\
& = & (r2 = 0 \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 1)) \\
\\
A^{III}[L16] & = & \mathit{wp}(M[16]) \\
& = & (s[0] = 0 \Rightarrow \mathit{shift}(\mathit{head}(M[L24])))\wedge \\
& & (\neg(s[0] = 0) \Rightarrow \mathit{shift}(\mathit{head}(M[L17]))) \\
& = & (s[0] = 0 \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & (\neg(s[0] = 0) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\mathrm{a},\ \mathrm{p}) = (\texttt{SecState.accessed}, 1)))
\end{array}
$$

$$
\begin{array}{lcl}
A^{III}[L15] & = & wp(M[15]) \\
& = & (shift(head(M[L16])))[s[0] = s[1]?1 : 0/s[1]] \\
& = & ((s[0] = s[1]?1 : 0) = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & (\neg((s[0] = s[1]?1 : 0) = 0) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 1))) \\
& = & (\neg(s[0] = s[1]) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((s[0] = s[1]) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 1)))
\end{array}
$$

$$
\begin{array}{lcl}
A^{III}[L14] & = & wp(M[14]) \\
& = & (head(M[L15]))[strclass(s[0])/s[0]] \\
& = & ((\neg(strclass(s[0]) = s[1]) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(s[0]) = s[1]) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 1))))
\end{array}
$$

$$
\begin{array}{lcl}
A^{III}[L13] & = & wp(M[L13]) \\
& = & (unshift(head(M[L14]))[\text{"}\texttt{GUI}\text{"}/s[0]]) \\
& = & ((\neg(strclass(\text{"}\texttt{GUI}\text{"}) = s[0]) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\text{"}\texttt{GUI}\text{"}) = s[0]) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 1))))
\end{array}
$$

$$
\begin{array}{lcl}
A^{III}[L12] & = & wp(M[L12]) \\
& = & (head(M[L13])[declass(s[0])/s[0]]) \\
& = & ((\neg(strclass(\text{"}\texttt{GUI}\text{"}) = declass(s[0])) \Rightarrow \\
& & (\text{a, p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\text{"}\texttt{GUI}\text{"}) = declass(s[0])) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 1))))
\end{array}
$$

$$
\begin{array}{lcl}
A^{III}[L11] & = & wp(M[L11]) \\
& = & (shift^2(head(M[L12]))[meth(s[0], s[1], s[2])/s[2]] \\
& = & ((\neg(strclass(\text{"}\texttt{GUI}\text{"}) = declass(meth(s[0], s[1], s[2]))) \\
& & \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\text{"}\texttt{GUI}\text{"}) = declass(meth(s[0], s[1], s[2]))) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow (\text{a, p}) = (\texttt{SecState.accessed}, 1))))
\end{array}
$$

$$
\begin{array}{lcl}
A^{III}[L10] & = & wp(M[L10]) \\
& = & unshift(head(M[L11])[\texttt{"AskUser"}/s[0]]) \\
& = & ((\neg(strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, s[0], s[1]))) \Rightarrow \\
& & \text{(a, p)} = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, s[0], s[1]))) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 1)))) \\
A^{III}[L9] & = & wp(M[L9]) \\
& = & unshift(head(M[L10])[/s[0]]) \\
& = & ((\neg(strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, null, s[0])))) \\
& & \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, , s[0]))) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 1)))) \\
A^{III}[L8] & = & wp(M[L8]) \\
& = & (head(M[L9])[objclass(s[0])/s[0]] \\
& = & ((\neg(strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, null, objclass(s[0]))))) \Rightarrow \\
& & \text{(a, p)} = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, , objclass(s[0]))))) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 1)))) \\
A^{III}[L7] & = & wp(M[L7]) \\
& = & unshift((head(M[L8])[r1/s[0]])) \\
& = & ((\neg(strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, null, objclass(r1))))) \\
& & \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, null, objclass(r1)))) \Rightarrow \\
& & \quad (r2 = 0 \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(r2 = 0) \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 1)))) \\
A^{III}[L6] & = & norm((g_{\text{this}} = r1) \cdot A^{II}_M[L6] \cdot wp(M[L6])) \\
& = & norm((g_{\text{this}} = r1) \cdot A^{II}_M[L6] \cdot (shift(head(M[7])))[s[0]/r2]) \\
& = & norm((g_{\text{this}} = r1) \cdot A^{II}_M[L6] \cdot \alpha) \\
& = & norm((g_{\text{this}} = r1) \cdot (g_{\text{this}} : \texttt{GUI} \wedge \text{(a, p)} \neq \overrightarrow{\perp}) \cdot \alpha[ce/\text{(a, p)}]) \cdot ((\text{a, p}) := ce) \cdot \alpha \\
& = & ((g_{\text{this}} = r1) \wedge (g_{\text{this}} : \texttt{GUI} \Rightarrow \text{(a, p)} \neq \overrightarrow{\perp}) \wedge \alpha[ce/\text{(a, p)}]) \cdot ((\text{a, p}) := ce) \cdot \alpha
\end{array}
$$

where

$$
\begin{array}{lcl}
\alpha & = & (\neg(r1 : \texttt{GUI}) \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, \texttt{SecState.permission}))\wedge \\
& & ((r1 : \texttt{GUI}) \Rightarrow \\
& & \quad (s[0] = 0 \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 0))\wedge \\
& & \quad (\neg(s[0] = 0) \Rightarrow \text{(a, p)} = (\texttt{SecState.accessed}, 1))) \\
ce & = & g_{\text{this}} : \texttt{GUI} \wedge \text{s[0]} \rightarrow (\text{a, true}) \mid g_{\text{this}} : \texttt{GUI} \wedge \neg\text{s[0]} \rightarrow (\text{a, false}) \mid \neg(g_{\text{this}} : \texttt{GUI}) \rightarrow (\text{a, p})
\end{array}
$$

Notice that $r1 : \texttt{GUI} = (strclass(\texttt{"GUI"}) = declass(meth(\texttt{"AskUser"}, null, objclass(r1))))$ for this target program.

B Proofs

B.1 Proof of Theorem 5.2

Theorem 5.2 (Correctness, Monitoring by Co-execution) *The program T adheres to policy $\mathcal{P}$ if, and only if, for each execution $C_0C_1\cdots$ of T there is a co-execution w for the automaton $\mathcal{A}_\mathcal{P}$ such that $w \downarrow 1 = C_0C_1\cdots$.*

Proof. ($\Rightarrow$) Let $\mathcal{A}_\mathcal{P} = (Q, A, \delta, q_0)$. We show that if T adheres to $\mathcal{P}$, then for each execution E of T, there exists a co-execution w such that $w \downarrow 1 = E$ by (1) constructing a configuration-automaton state sequence w for each execution and (2) proving that w is a co-execution with $w \downarrow 1 = E$.

(1) If the program adheres to the policy, the security relevant trace of every execution E is in the language of the automaton: $srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}$. We construct a configuration-automaton state sequence w for an execution using the accepting run $q_0q_1\ldots$ of the automaton for the security relevant trace of the execution. We begin the construction with the initial configuration C_0 and the initial state q_0. We add the remaining configurations, paired with this state until a security relevant action (s.r.a.) is met. Whenever an s.r.a. is

met, the state component of the added pair is changed with the next automaton state in the run. This process is repeated until both the end of the execution and of the automaton run is reached. Each time the sequence is to be extended for the next configuration, an s.r.a. is induced if the latest configuration and this new configuration induce a postaction or if the new configuration induces a preaction by itself.

Suppose the constructed sequence for the execution $C_0C_1C_{n-1}$ where $n \geq 0$ is w_{n-1} and the current state is q_k (the state component of the last pair of w_{n-1}). The sequence w_n for the execution $C_0C_1 \ldots C_n$ is defined as follows:

- $act^\sharp(C_{n-1}, C_n)act^\flat(C_n) = \epsilon$:

$$w_n = w_{n-1} \cdot (C_n, q_k)$$

- $act^\sharp(C_{n-1}, C_n)act^\flat(C_n) = a^\flat$ for some $a^\flat \in A^\flat$:

$$w_n = w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1})$$

- $act^\sharp(C_{n-1}, C_n)act^\flat(C_n) = a^\sharp$ for some $a^\sharp \in A^\sharp$:

$$w_n = w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1})$$

- $act^\sharp(C_{n-1}, C_n)act^\flat(C_n) = a^\flat a^\sharp$ for some $a^\flat \in A^\flat, a^\sharp \in A^\sharp$:

$$w_n = w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) \cdot (C_n, q_{k+2})$$

The sequence for the execution of length 0 is the sequence of length 0: $w_{-1} = \epsilon$. When constructing w_1, the current state is taken as q_0 and $act^\sharp(C_{-1}, C_0)$ as ϵ. Note that the number of s.r.a.'s in the execution is the position of the current state in the given automata run. For instance when there are no configurations (and hence s.r.a's) in the execution, the current state is q_0. If the new configuration induces one s.r.a, and the state was q_k, the new current state is q_{k+1} and if it induces two s.r.s's, the current state is q_{k+2}. This means that if the program adheres to the policy, the sequence is well-defined.

(2) We prove the statement by induction on the length of the execution. We consider finite executions only and note that the statement holds for an infinite execution since it holds for all its finite prefixes. Think!

(*Base Case*) Consider the execution consisting of the initial configuration C_0. If $act^\flat(C_0) \in A^\flat$, then the security relevant trace of the execution consists of this action: $srt_A(E) = act^\flat(C_0)$. Let the automaton run for this trace be q_0q_1 where $\delta(q_0, act^\flat(C_0)) = q_1$. Then $w_0 = (C_0, q_0)(C_0, q_1)$ by construction. This sequence is an interleaving since q_0q_1 is an automaton run: $(C_0, q_0) \longrightarrow_{\text{AUT}} (C_0, q_1)$. By the definition of the first projection function $w_0 \downarrow 1 = C_0$. Finally, by the definition of the *extract* function $extract(w_0) = q_0q_1\ act^\flat(C_0)$ and clearly $q_0q_1\ act^\flat(C_0) \in E^\flat$. If $act^\flat(C_0) = \epsilon$ on the other hand, the security relevant trace is empty $srt_A(E) = \epsilon$. The accepting run then consists of q_0 and the constructed sequence w_0 of (C_0, q_0). By the definition of the first projection function $w_0 \downarrow 1 = C_0$, and $extract(w_0) = \epsilon$.

(*Induction Hypothesis*) Assume that the statement holds for all executions of length k, $k \leq n$.

(*Inductive Step*) Consider the sequence w_n constructed for the execution $C_0 \ldots C_{n-1}C_n$ using the automaton run $q_0 \ldots q_m$. Consider the execution $E_{n-1} = C_0 \ldots C_{n-1}$. By definition, the following holds:

$$srt_A(E_n) = srt_A(E_{n-1})\ act^\sharp(C_{n-1}, C_n)\ act^\flat(C_n)$$

We consider the case where $act^\sharp(C_{n-1}, C_n) \in A^\sharp, \quad act^\flat(C_n) \in A^\flat$. (The other cases are similar, but simpler.) Since $q_0 \ldots q_m$ is an accepting run for this execution:

$$\begin{array}{rcll} \delta^*(q_0, srt_A(E_{n-1})) & = & q_{m-2} & (i) \\ \delta(q_{m-2}, act^\sharp(C_{n-1}, C_n)) & = & q_{m-1} & (ii) \\ \delta(q_{m-1}, act^\flat(C_n)) & = & q_m & (iii) \end{array}$$

Then the sequence w_{n-1}, constructed (as described above) for E_{n-1} using the run $q_0 \ldots q_{m-2}$, is a co-execution by the induction hypothesis. Note that the last component of this co-execution is C_{n-1}, q_{m-2} by the construction. Again by construction, the sequence w_n is an extension of w_{n-1} (last case of the construction):

$$w_n = w_{n-1}(C_n, q_{m-2})(C_n, q_{m-1})(C_n, q_m)$$

We prove that:

- *w_n is an interleaving:* The sequence w_n is an interleaving by the induction hypothesis. Since E_n is an execution, there is a machine transition from C_{n-1} to C_n. There are transitions between the consecutive states $q_{m-2}q_{m-1}q_m$ of the automaton run. Thus the extension to w_{n-1} consists of one machine transition followed by the automaton transitions:

$$\left.\begin{array}{rcl} (C_{n-1}, q_{m-2}) & \longrightarrow_{\text{JVM}} & (C_n, q_{m-2}) \\ (C_n, q_{m-2}) & \longrightarrow_{\text{AUT}} & (C_n, q_{m-1}) \\ (C_{n-1}, q_{m-1}) & \longrightarrow_{\text{AUT}} & (C_n, q_m) \end{array}\right\} (*)$$

- *$w \downarrow 1 = E_n$:* This simply follows from the induction hypothesis and applying the first projection function to w_n.

- *w is a co-execution:* By assumption w_{n-1} is a co-execution. Then $extract(w_{n-1}) \in (E^\flat \cup E^\sharp)^{m-2}$ since there $m-2$ s.r.a's in E_{n-1}. By definition of the *extract* function and using (*):

$$extract(w_n) = extract(w_{n-1}) act^\sharp_A(C_{n-1}, C_n)\ q_{m-2}q_{m-1}q_{m-1}q_m\ act^\flat_A(C_n)$$

  By (ii), $act^\sharp_A(C_{n-1}, C_n)q_{m-2}q_{m-1} \in E^\sharp$ and by (iii), $q_{m-1}q_m act^\flat_A(C_n) \in E^\flat$. Hence $extract(w_n) \in (E^\flat \cup E^\sharp)^m$.

($\Leftarrow$) Let $\mathcal{A}_\mathcal{P} = (Q, A, \delta, q_0)$. We first show that an execution $C_0C_1 \cdots$ for which there exists a co-execution w for $\mathcal{A}_\mathcal{P}$ such that $w \downarrow 1 = C_0C_1 \ldots$ is in the language of the automaton:

$$\forall E. \exists w. (w \downarrow 1) = E \Rightarrow srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}$$

Since this is the case for all executions of T, we can then conclude that T obeys the policy.

We begin by noting that reflecting the result of the extract function of a co-execution to the program actions produces the program's security relevant trace, that is, $(extract_\mathcal{A}(w)) \downarrow A = srt_\mathcal{A}(w \downarrow \text{T})$. For a program to adhere to the policy, there should exist a trace of the automaton that accepts the security relevant trace of the program. Such a trace is provided by the automaton component of the co-execution and can be constructed by reflecting the result of the extract function, this time, to the automaton states. Here we consider the case when the security relevant trace of the program is finite (the case when it is infinite is similar) and let $(extract_\mathcal{A}(w)) \downarrow A = a_1 \cdot a_2 \cdot \ldots a_n$. Then $(extract_\mathcal{A}(w)) \downarrow Q = q_1q_2 \ldots q_m$ for some $m >= 0$ and by the definition of co-execution, $\delta(q_{2*i}, a_i) = q_{2*i+1}$ for all $1 \leq i \leq n$

and $m = 2 * n$. We are left to show that the transitions in $(extract_{\mathcal{A}}(w)) \downarrow Q$ follow each other to form an automaton trace, that is for all $k \leq m$ where k is even, $q_k = q_{k+1}$. This is so since when we apply the extract function to the interleaving, we consider each state twice: once as the possible destination state of a transition and another time as the possible initial state for a transition. So the trace constructed by filtering the states in even positions from the state sequence $(extract_{\mathcal{A}}(w)) \downarrow Q)$ gives us an accepting trace for the security relevant trace $srt_{\mathcal{A}}(w \downarrow \mathrm{T})$. Hence, the execution of the program is in the enforcement language of the automaton. □

In the text below, the program T annotated with level I annotations for policy $\mathcal{P}$ is $\mathrm{T}_{\mathcal{P}}$. Furthermore, $pc(C_k)$ denotes the value of the program counter and $M(C_k)$ the method at the top frame of configuration C_k. Finally, $\sigma(\overrightarrow{gs})$ denotes the value of the ghost state given by the mapping σ.

The following two propositions follow from the definition of level I annotations.

Proposition B.1. *Let C be an unexceptional configuration of program T. If the BEFORE annotation $A^{\flat}_{M}[pc(C)]$ of $pc(C)$ in $T_{\mathcal{P}}$ is ϵ, then $act^{\flat}_{A}(C) = \epsilon$.*

Proposition B.2. *Let C_1 and C_2 be two unexceptional consecutive configurations in an execution of program T. If the AFTER annotation $A^{\sharp}_{M}[pc(C_1)]$ of $pc(C_1)$ in $T_{\mathcal{P}}$ is ϵ, then $act^{\sharp}_{A}(C_1, C_2) = \epsilon$.*

Given an annotated program T_A, a sequence of extended configurations $(\psi_0, C_0, \sigma_0)(\psi_1, C_1, \sigma_1)\ldots$ is an *extended execution* of T_A, if:

- C_0 is an initial machine configuration, $\psi_0 = \mathit{Requires}_{\langle \texttt{main} \rangle} \cdot A_{\langle \texttt{main} \rangle}[1]$, $\sigma_0 = [\,]$ and
- $\forall i.\Gamma^* \vdash (\psi_i, C_i, \sigma_i)(\psi_{i+1}, C_{i+1}, \sigma_{i+1})$

That is, an extended execution is any execution as described in def 6.1.

The projection of an extended execution to its second component isolates the execution of the JVM program, and is described similar to the definition of the first projection function in section 5.

Given a finite execution $E = C_0 C_1 \ldots C_n$ of program T, the extended execution $X_E = (\psi_0, C'_0, \sigma_0)(\psi_1, C'_1, \sigma_1)\ldots(\psi_m, C'_m, \sigma_m)$ of the annotated program $\mathrm{T}_{\mathcal{P}}$ is the *complete* extended execution of E if $X_E \downarrow 2 = E$ and $\psi_m = \epsilon$. That is an extended execution is complete if it executes the precondition (if any) of the instruction at the program counter of its last configuration to completion. Given a finite execution $E = C_0 C_1 \ldots C_{n-1}$ of program T, notice that the following hold for the execution $E' = C_0 C_1 \ldots C_{n-1} C_n$ if C_{n-1} is not an application method call and C_n is not exceptional:

$$X_{E'} = X_E\,(A_{M(C_n)}[pc(C_n)], C_n, \sigma)\ldots(\epsilon, C_n, \sigma)\ \ (*)$$

where (ϵ, C_n, σ) is the last element of X_E. If C_{n-1} is an application method call and C_n is not exceptional $A_M[pc(C_n)]$ is replaced by $\mathit{Requires}_{M(C_n)} A_{M(C_n)}[1]$. Finally, if C_n is exceptional, there is no annotation to be executed (as *Exsures* clause is empty in level I annotations):

$$X_{E'} = X_E\,(\epsilon, C_n, \sigma)$$

For each extended execution X of the annotated program $\mathrm{T}_{\mathcal{P}}$, we extract a sequence of configuration-automaton pairs by sampling the value of the ghost state. If a configuration induces a preaction, the annotated program $\mathrm{T}_{\mathcal{P}}$ updates the ghost state immediately before transiting to the next configuration (that is "executing the method"). If two consecutive configurations induce a non-exceptional postaction, the ghost state is updated immediately after transiting to the second configuration (that is upon return). However, in the case of an exceptional postaction the update is not immediate. When two consecutive configurations C and C' induce an exceptional action, the new state can not be extracted from the extended execution that ends with C'. The reason is that there is no annotation associated with exceptional configurations. The necessary ghost update in this case is done as a precondition to the first instruction of the handler. This is executed after the transition to the configuration following C'. Since we assume that there is a handler for any exception

raised by an API method call, there can not be a maximal execution having the suffix CC'. Hence, below, we consider each finite execution as a prefix of a maximal execution.

Let $E = C_0 \dots C_j$ be a finite execution and let $X_E = (\psi_0, C'_0, \sigma_0) \dots (\psi_k, C'_k, \sigma_k)$ be its corresponding extended execution. Notice that the first two extended configurations correspond to the execution of $Requires_{\langle \texttt{main} \rangle}$. If the last two configurations C_{j-1}, C_j of E do not induce an exceptional action, the sequence of configuration-automaton pairs corresponding to this extended execution is defined as

$$w(X_E) = (C_0, q_0)\ subw((\psi_1, C'_1, \sigma_1) \dots (\psi_k, C'_k, \sigma_k))$$

where q_0 is the initial state of $\mathcal{A}_\mathcal{P}$ and $subw$ is defined below. If C_{j-1} and C_j induce an exceptional action, we extract the co-execution using the complete extended execution of $E' = C_0 \dots C_j C_{j+1}$. Let X' be the extended execution where $X' \downarrow 2 = E'$ and X' ends with the execution of the annotation $Exc(pc(C_{j+1}), M(C_{j+1}))$ as described in sec 6.2. Then the value of the ghost state at the last element of X' is taken.

A sequence of configuration-automaton pairs are extracted from a sequence of extended configurations using the function $subw$. This function forms a sequence by sampling the machine configuration and the ghost state whenever one of the two is updated. If the machine configuration changes in consecutive extended configurations, the sequence is extended with the machine configuration and the ghost state of this second extended configuration. If the current annotation is a conditional ghost assignment, then the configuration and ghost state of the extended configuration after the execution of this annotation is added to the sequence. If the current extended configuration is the last one, then the sequence is not extended further.

$$subw((\psi_1, C_1, \sigma_1) \cdot (\psi_2, C_2, \sigma_2) \cdot X') = \begin{cases} (C_2, \sigma_2(\overrightarrow{gs})) \cdot subw((\psi_2, C_2, \sigma_2) \cdot X') & \text{if } C_1 \longrightarrow_{\text{JVM}} C_2 \\ (C_2, \sigma_2(\overrightarrow{gs})) \cdot subw((\psi_2, C_2, \sigma_2) \cdot X') & \text{if } \psi_1 = (\overrightarrow{gs} := \alpha_1 | \dots | \alpha_k) \cdot \psi_2 \\ & \text{for some } k \neq 1 \\ subw((\psi_2, C_2, \sigma_2) \cdot X') & \text{otherwise} \end{cases}$$

$$subw(\psi, C', \sigma) = \epsilon$$

In the definition above, the update of the ghost state causes a sampling only if the update is not done by the last condition of the conditional update. The reason is that when the program is annotated with level I annotations, an update on the ghost state using the last condition of the conditional expression is a stutter.

Definition 5.1 captures all interleavings of the monitor and the program, for a monitor that updates the security state every time a s.r.a. occurs. If a configuration induces a preaction, the update should happen before the transition to the next configuration. If two consecutive configurations induce a postaction, the update should be done after the transition to the latter configuration. The definition aims to specify the interval where the update may be done for the interleaving to be a co-execution. A co-execution is a *closest updating co-execution* if the monitor makes a corresponding transition at the latest possible point when the update is for a preaction and at the earliest possible point when the update is for a postaction.

Definition B.3 (Closest Updating Co-execution)**.** *A co-execution is* closest updating co-execution *if the following holds for consecutive pairs* $(C_1, q_1)(C_2, q_2)(C_3, q_3)(C_4, q_4)$*:*

- $act^\flat_A(C_1) \in A^\flat \wedge (C_2, q_2) \longrightarrow_{\text{JVM}} (C_3, q_3) \implies (C_1, q_1) \longrightarrow_{\text{AUT}} (C_2, q_2)$
- $act^\sharp_A(C_1, C_2) \in A^\sharp \wedge Unexc(C_2) \implies (C_2, q_2) \longrightarrow_{\text{AUT}} (C_3, q_3)$

- $act^\sharp_A(C_1, C_2) \in A^\sharp \wedge \mathit{Handled}(C_2) \implies (C_2, q_2) \longrightarrow_{\text{JVM}} (C_3, q_3) \wedge (C_3, q_3) \longrightarrow_{\text{AUT}} (C_4, q_4)$

We now prove that, the configuration-automaton state pairs extracted from a level I annotated program is a co-execution, provided that the annotations are valid and vice versa. What is more, due to the shape of the annotations, we prove that these co-executions are closest updating.

Lemma B.4. *$T_\mathcal{P}$ is valid, if and only if, for every maximal execution E of T, the extracted sequence $w(X_E)$ of the complete extended execution X_E of $T_\mathcal{P}$ is closest updating and $w(X_E) \downarrow 1 = E$.*

Proof. There are two aspects to the proof. First, we are showing that ghost assignments follow security relevant method executions and are performed according to the way described in the policy. Second, that no security relevant action execution happens when the ghost state is undefined if and only if the annotated program is valid.

We proceed by induction on the number of configurations in E.

When the number of configurations in E is 0, *TODO*!!

For all executions $E_k = C_0 \dots C_{k-2} C_{k-1}$ of length k such that $k \leq n$ and $act^\sharp_A(C_{k-2}, C_{k-1})$ is not an exceptional post action, we assume that $w(X_k)$ is a co-execution where $w(X_k) \downarrow 1 = E_k$ if and only if all boolean formulas asserted in the complete extended execution X_k holds except possibly the assertions $\mathit{Defined}^\sharp$ and $\mathit{Defined}^e$ asserted in the course of the execution of the precondition of $pc(C_{k-1})$.

Notice that this induction assumption is sufficient, since no maximal execution can end with an exceptional configuration that is immediately preceded by an exceptionally security relevant API method call. Similarly, for no maximal execution $\mathit{Defined}^\sharp$ or $\mathit{Defined}^e$ is asserted in the course of the execution of the precondition of $pc(C_{k-1})$. If the maximal execution is one which returns from the $\langle$`main`$\rangle$, then $pc(C_{k-1})$ is `return` and hence no definedness precondition. If the maximal execution is one which ends exceptionally, then this exception is not one thrown by a security relevant API method.

$n' = n + 1$ Consider the execution $E_{n+1} = C_0 \dots C_n$ of T and its corresponding extended execution $X_{E_{n+1}}$.

We consider the different forms of the pair C_{n-1}, C_n:

- C_{n-1} and C_n are both not exceptional, and C_{n-1} is not an application method call: We have assumed that the statement holds for $E_n = C_0 \dots C_{n-1}$. Since $X_{E_{n+1}}$ is an extension of X_{E_n}, the assertions met in $X_{E_{n+1}}$ hold if and only if assertions met in X_{E_n} and X hold where $X_{E_{n+1}} = X_{E_n} \cdot X$. By the induction assumption, the assertions met in X_{E_n} of $\mathrm{T}_\mathcal{P}$ hold if and only if $w(X_{E_n})$ is a co-execution and $w(X_{E_n}) \downarrow 1 = E_n$.

  Let the last element of X_{E_n} be $(\epsilon, C_{n-1}, \sigma)$ for some σ, executing method of C_n be M and $\mathrm{pc}(C_n)$ be L. Notice that since C_{n-1} is not exceptional, L is not a handler instruction. By the definition of a complete extended execution, the first element of the suffix X is $(A_M[L], C_n, \sigma)$, and its last element is (ϵ, C_n, σ') for some σ' that is determined by the assignments in $A_M[L]$. That is X corresponds to the execution of the annotation sequence that is associated with L in M: $A_M[L]$. By the definition of *subw* and w:

  $$w(X_{E_{n+1}}) \quad = \quad w(X_{E_n})(C_n, \sigma(\overrightarrow{gs})) \cdot \mathit{subw}(X)$$

  By the definition of level I annotations,

  $$A_M[L] = \quad A^e_M[L-1][1] \cdot A^\sharp_M[L-1][1] \cdot A^\flat_M[L] \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0]$$

In the rest of the argument of this case, we take $A^e_M[L-1][1] = \epsilon$ for simplicity. This annotation otherwise would set g_{pc} to 0, which does not change the argument.

Notice that, again by the definition of level I annotations, $A_M[L]$ contains at most two assignments to the ghost state in this case. (For all L', $A^\sharp_M[L'][1]$ and $A^\flat_M[L']$ can contain at most one ghost assignment (to the ghost state), while $A^\sharp_M[L'][0]$, $A^e_M[L'][0]$ and $A^e_M[L'-1][1]$ can not contain any.) In order to go through all shapes the suffix $subw(X)$ can have, we consider the possible ghost assignments in X:

1. $A^\sharp_M[L-1][1] = \epsilon$, $A^\flat_M[L] = \epsilon$
   In this case there are no ghost assignments in $A_M[L]$ and so $subw(X) = \epsilon$ by definition. Then, $w(X_{E_{n+1}}) = w(X_{E_n})(C_n, \sigma(\overrightarrow{gs}))$.
   ($\Rightarrow$) From this and the induction hypothesis, the following can be concluded: (i) $w(X_{E_{n+1}})) \downarrow 1 = E_{n+1}$ by the definition of $\downarrow$, (ii) $w(X_{E_{n+1}})$ is an interleaving, since the last element of $w(X_{E_n})$ is $(C_{n-1}, \sigma(\overrightarrow{gs}))$ and $C_{n-1} \longrightarrow_{\text{JVM}} C_n$.
   By the definition of the *extract* function:

   $$extract(w(X_{E_{n+1}})) = extract(w(X_{E_n}))act^\sharp_A(C_{n-1}, C_n)act^\flat_A(C_n)$$

   By lemma B.1, $act^\flat_A(C_n) = \epsilon$ and by lemma B.2, $act^\sharp_A(C_{n-1}, C_n) = \epsilon$. By the induction hypothesis, $w(X_{E_{n+1}})$ is a co-execution.
   ($\Leftarrow$) Assume that $w(X_{E_{n+1}})$ is a closest updating co-execution The only way this is possible is that $w(X_{E_n})$ is itself a closes updating (c.u.) co-execution, and $act^\flat_A(C_n) = \epsilon$, $act^\sharp_A(C_{n-1}, C_n) = \epsilon$. (Otherwise there would be ghost updates executed in $w(X_{E_{n+1}})$)). The latter we have already shown to hold. By the induction hypothesis, if $w(X_{E_n})$ is a c.u. co-execution then all assertions (except possibly the definedness assertions $Defined^\sharp$ and $Defined^e$ executed for $pc(C_{n-1})$) hold. Since $A^\flat_M[L] = \epsilon$, there is no $Defined^\flat$ that is asserted in the precondition of $pc(C_{n-1})$, hence the only assertions that should be shown to hold are $Defined^\sharp$ and $Defined^e$ of $pc(C_{n-1})$. If $pc(C_{n-1})$ is not a method invocation instruction, there is no definedness assertions in its precondition, and we are done. If $pc(C_{n-1})$ is a method invocation instruction, either C_{n-1} is either an application method call or an API method call. In the former case, both $Defined^\sharp$ and $Defined^e$ hold vacuously since the premise of the boolean formula does not hold, that is the object that the method is invoked on is not one of those mentioned in these assertions.
   Let us consider the case where C_{n-1} is an API method call. Since there are no jumps to instructions after method calls, $pc(C_{n-1})$ should be $L-1$. By the definition of AFTER annotations, $A^\sharp_M[L-1][1] = \epsilon$ implies that $A^\sharp_M[L-1][0] = \epsilon$, so there is no $Defined^\sharp$ for $pc(C_{n-1})$. If it is also the case that $A^e_M[L-1][0] = \epsilon$, we are done. If there is a $Defined^e$ however, we have to show that this also holds. Suppose that $Defined^e$ which comes from $A^e_M[L-1][0]$ does not hold. Then an alternative execution of the program can be constructed by replacing C_n with C'_n where C'_n is exceptional. Since $L-1$ is exceptionally security relevant (otherwise there would be no $Defined^e$ asserted for C_{n-1}), there is a handler H for $L-1$. Now consider the alternative execution that is archived by extending the execution with C'_{n+1} where $pc(C_{n-1}) = H$: $E' = C_0 \ldots C_{n-1}C'_nC'_{n+1}$. Then $w(X_{E'})$ can not be a co-execution. We reach a contradiction.
2. $A^\sharp_M[L-1][1] \neq \epsilon$, $A^\flat_M[L] = \epsilon$
   Then the suffix X is as follows:

$$
\begin{array}{ll}
 & ((\overrightarrow{gs} := ce) \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0], C_n, \sigma) \quad (1)\\
\rightarrow_* & ((\overrightarrow{gs} := \alpha_1| \cdot \cdots \cdot |\alpha_k) \cdot A^\sharp_M[L][0] \cdot A^e_M[L][0], C_n, \sigma) \quad (2)\\
\rightarrow & (A^\sharp_M[L][0] \cdot A^e_M[L][0], C_n, \sigma') \quad (3)\\
\rightarrow_* & (\epsilon, C_n, \sigma'') \quad (4)
\end{array}
$$

By definition, $subw(X) = \epsilon$ if $k = 1$ and $subw(X) = (C_n, \sigma'(\overrightarrow{gs}))$ otherwise. Notice that $\sigma''(\overrightarrow{gs}) = \sigma'(\overrightarrow{gs})$ since there are no assignments to the ghost state in the steps between (3) and (4) and furthermore if $k = 1$, $\sigma'(\overrightarrow{gs}) = \sigma(\overrightarrow{gs})$, by the definition of level I annotations.

By the definition of AFTER annotations, $A^\sharp_M[L-1][1] \neq \epsilon$ if $M[L-1] =$ `invokevirtual` $(c.m)$ for some class c and method m. That is the instruction at above the current program counter is a method invocation instruction. By the assumption that there are no direct jumps to instructions immediately below method calls, the previous configuration is either a method call (to an API method) or a method return (from an application method).

($\Leftarrow$) This direction is similar to the argument for the case above.

($\Rightarrow$) As is apparent from the execution of X, $subw(X)$ is determined by the value of k above:

(a) $k = 1$:

This corresponds to the case where we have a stuttering if the ghost state is defined when the assignment begins executing. This type of stuttering is meant to occur when the current call is not to a security relevant action, in order to not to update the state unnecessarily with this assignment. This last condition can be satisfied also if the ghost state is not defined when the assignment begins executing. In this case, for the extracted sequence to be a co-execution, the method return should not be a postaction.

By the definition of *subw*, $subw(X) = \epsilon$ and $w(X_{E_{n+1}})$ is the same as case 1 above. The argument that this is an interleaving and that its first projection is E_n is also identical. The equation 1 also holds. For $w(X_{E_{n+1}})$ to be a co-execution then, we should show that no security relevant actions are induced by the addition of configuration C_n to the execution E_{n-1}. By lemma B.1, $act^\flat_A(C_n) = \epsilon$. If C_{n-1} is a return from an application method, $act^\sharp_A(C_{n-1}, C_n) = \epsilon$. The case where C_{n-1} is a method call to an API is more complicated. This case is to prove that, although this instruction has been annotated, in this case the method called as a result of virtual method resolution turned out not to be security relevant.

Let C_{n-1} be $((M, L-1, s \cdot d \cdot s', lv) \cdot R, h^\flat)$ and C_n be $((M, L, v \cdot s', lv) \cdot R, h^\sharp)$ for some actual arguments s, some location d, some stack s' and return value v. Notice that there exists a class c' such that c' *defines* $type(h^\flat, d).m$ and $type(h^\flat, d) <: c$. (If this was not the case, C_n would be exceptional.) Now suppose $(v, c', m, s, h^\flat, h^\sharp)$ is a postaction of the induced automaton $A_\mathcal{P}$. Then there should exist, for some names $x, x_1, \ldots x_n$, a symbolic postaction $a^\sharp_s = (\tau x, c', m, ((\tau_1\ x_1), \ \ldots, \ (\tau_n\ x_n)))$ of A_s such that the type of v is τ, the type of $s[0]$ is τ_1 etc. It would then be the case that $type(h^\flat, d) \in RS((c, m), A^\sharp_s \setminus A^e_s)$, by the definition of RS. Notice that $\sigma(g_{\text{this}}) = d$ by the execution of $A^\sharp[L-1][0]$ in $X_{E_{n-1}}$.

Since $k = 1$, either $\neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p)$ or $\overrightarrow{gs} = \overrightarrow{\perp}$ or both of

them holds at (2) where $RS((c,m), A_s^\sharp \setminus A_s^e) = \{c'_1, \ldots, c'_p\}$. If only the first holds, at (2), d is not an object of one of these classes, $type(h^\flat, d) \notin RS((c,m), A_s^\sharp \setminus A_s^e)$. (We assume that $type(h^\flat, d) = type(h^\sharp, d)$, that is an API call does not change the type of the object it is called on) We reach a contradiction, showing that $act_A^\sharp(C_{n-1}, C_n) = \epsilon$. Hence, $extract(w(X_{E_{n+1}}))$ is a co-execution. If both holds, then the return is again not security relevant and $extract(w(X_{E_{n+1}}))$ is a co-execution.

If only the second holds however $act_A^\sharp(C_{n-1}, C_n) \in A^\sharp$ and $extract(w(X_{E_{n+1}}))$ can not be a co-execution since there is no outgoing transitions from the undefined state in a ConSpec automaton induced from the symbolic automaton of the policy. In order to rule out this case, we should prove that $\sigma(\overrightarrow{gs}) \neq \bot$. Now we use the assumption that all assertions in $X_{E_{n+1}}$ holds. This is only the case if all assertions of X_{E_n} holds. By the definition of AFTER annotations, $A^\sharp[L-1][0]$ asserts that if g_{this} is of a class which is a member of $RS((c,m), A_s^\sharp \setminus A_s^e$, then $\sigma(\overrightarrow{gs}) \neq \bot$. Hence it can not be the case only the second conjunct holds.

(b) $k \neq 1$:
Let $\sigma(\overrightarrow{gs}) = q$ and $\sigma'(\overrightarrow{gs}) = q'$, by the definition of *subw* and of *extract*:

$$\begin{aligned} w(X_{E_{n+1}}) &= w(X_{E_n})(C_n, q)(C_n, q') \\ extract(w(X_{E_{n+1}})) &= extract(w(X_{E_n}))act_A^\sharp(C_{n-1}, C_n)qq' act_A^\flat(C_n) \end{aligned}$$

In order to show that $w(X_{E_{n+1}})$ is an interleaving, we should prove that there exits an action $a \in A$ such that $\delta(q,a) = q'$. From this, it will also follow that $w(X_{E_{n+1}}) \downarrow 1 = E_{n+1}$. To prove that $w(X_{E_{n+1}})$ is a co-execution, however, we should prove a stronger statement, namely that $\delta^\sharp(q, act_A^\sharp(C_{n-1}, C_n)) = q'$. (This is the only possibility since by lemma B.1, $act_A^\flat(C_n) = \epsilon$)

- $k > 1$: This is the case when one of the conditions (other than the last condition) of the conditional assignment is satisfied and the ghost state is set accordingly. We show that this is the case only if C_{n-1} is a return from a post security relevant method call and that the ghost state is set correctly. Since $k > 1$, in the execution segment above, α_1 has the following form: $(\overrightarrow{gs} \neq \overrightarrow{\bot}) \wedge g_{\text{this}} : c'_i \wedge a \rightarrow \overrightarrow{e}$, where $c'_i \in RS((c,m), A_s^\sharp \setminus A_s^e)$. Note that α_1 holds at (2). This implies that $\overrightarrow{gs} \neq \overrightarrow{\bot}$ at σ.
  We first show that C_n can not be a return from an application method. (If this was the case the return would be from an application method, hence not security relevant). Assume that this is the case, let this method which is returning be c', m and the object it was called on to be d. (That is the second frame in the activation stack of C_{n-1} is $(M, L-1, s \cdot d \cdot s')$ for some actual arguments s, and some stack s') Since the call was made by the instruction `invokevirtual` $c.m$, it should be the case that c' *defines* $(type(d,h), m)$ where h is the heap at the time of the method call. Notice that $g_{\text{this}} = d$, since it was set to this value by $A^\sharp[L-1][0]$ just before the method call was made and since it is local so could not have been changed during the execution of the application method. (We further assume that the application method does not change the type of the object it is called on) This means that $c' \in RS((c,m), A_s^\sharp \setminus A_s^e)$, which can not be the case since it is an application method. Hence we reach a contradiction, showing that C_n can not be a return from an application method.

The only possibility left is that C_n is a return from an API method. Let C_{n-1} be $((M, L-1, s \cdot d \cdot s', lv) \cdot R, h^\flat)$ and C_n be $((M, L, v \cdot s', lv) \cdot R, h^\sharp)$ for some actual arguments s, some location d, some stack s', return value v and heaps $h^\flat, h^\sharp$. Let $(c.m) : (\gamma \to \tau)$. Since α_1 is a part of the ghost assignment, the symbolic automaton should include the action $a_s^\sharp = (\tau x, c_i', m, ((\tau_1 x_1), \ldots, (\tau_{|\gamma|} x_{|\gamma|})))$ for some names $x, x_1, \ldots$ and types $\tau, \tau_1, \ldots$ such that the type of v is τ, the type of $s[0]$ is τ_1 etc. What is more there exists a predicate b and an expression tuple E such that $(a_s^\sharp, b, E) \in \delta_s^\sharp$ and $a = a_b\rho$ where a_b is the boolean formula for predicate b and $\overrightarrow{e_E}$ as defined in section 6.2. The substitution $\rho = [v/x, g_0/x_0, \ldots g_{k-1}/x_{n-1}, g_{this}/\texttt{this}]$ by construction. Notice that $\sigma(g_{\text{this}}) = d$ by the execution of $A^\sharp[L-1][0]$ in $X_{E_{n-1}}$ and hence c_i' *defines type*$(h^\flat, d).m$. Thus $(v, c', m, s, h^\flat, h^\sharp)$ is a postaction of the induced automaton $A_\mathcal{P}$. We have proven that $act_A^\sharp(C_{n-1}, C_n) \in A^\sharp$.
We are left to prove that $\delta^\sharp(q, act_A^\sharp(C_{n-1}, C_n)) = q'$. Since α_1 holds at (2),

$$\| a_b\rho \| (C_n, \sigma) = true \Leftrightarrow \| b \| qIh^\flat h^\sharp = true$$

where $I = [x \mapsto v, x_1 \mapsto s[0], \ldots]$. Using the same interpretation,

$$\| \overrightarrow{e_E}\rho \| (C_n, \sigma) = q' \Leftrightarrow \| E(sv) \| qIh^\flat h^\sharp = q'(sv)$$

for all security state variables sv of $\overrightarrow{gs}$. The result then follows from the way a ConSpec automaton is induced by a symbolic automaton.

- $k = 0$ It is possible to show in this case that C_n is a return from a security relevant method call by a similar argument. The idea is that if C_n was a return from an application method call, the last condition of the conditional assignment would instead have been satisfied, hence k would have been 1. Since this is not the case, we know that C_n is a return from an API call. What is more, let this method be $c'.m$. Then $c' \in RS((c, m), A_s^\sharp \setminus A_s^e$. Notice that none of the conditions in the assignment hold, that is $k = 0$, if either $\sigma(\overrightarrow{gs}) = \perp$ or $\sigma(\overrightarrow{gs}) \neq \perp$ but the guards are not satisfied. In both cases, after this assignment the ghost state is undefined: $\sigma'(\overrightarrow{gs}) = \perp$.
The case that $k = 0$ may only occur if the ghost state becomes undefined since the return from the API method was a violation. Since the last condition does not hold, we know that the ghost state was not undefined at σ and we know that the object the method was called is of one of the classes in $RS((c, m), A_s^\sharp \setminus A_s^e$. This means that the call is security relevant. Since none of the conditions before the last was satisfied, this is a violating postaction. By the definition of the way a ConSpec automaton is extracted from a symbolic automaton, any such state has a transition to the undefined state. Hence $\delta^\sharp(q, act_A^\sharp(C_{n-1}, C_n)) = q'$, where $q' = \overrightarrow{\perp}$ and we are done.

Hence, $w(X_{E_{n+1}})$ is a co-execution.

3. The cases where $A_M^\sharp[L-1][1] = \epsilon$, $A_M^\flat[L] \neq \epsilon$ and where $A_M^\sharp[L-1][1] \neq \epsilon$, $A_M^\flat[L] \neq \epsilon$ are proved similar to the case above.

- C_{n-1} and C_n are both not exceptional, and C_{n-1} is an application method call:
This case can be viewed as a special case of the case when C_{n-1} is not an application method call (above). The difference is that the *Requires* of the called method is also executed, making sure that when the execution of a method begins g_{pc} is set to 0.

- C_{n-1} is exceptional, while C_n is not exceptional: The only interesting subcase of this case is when C_{n-2} is an API method call and $act_A^\sharp(C_{n-2}, C_{n-1}) \neq \epsilon$. In this case, notice that $w(X_{E_{n+1}})$ is not an extension of $w(X_{E_n})$, but rather of $w(X_{E_{n-1}})$, by the definition of w function.

- C_{n-1} is not exceptional, while C_n is exceptional: The only interesting subcase of this case is when C_{n-2} is an API method call and $act_A^\sharp(C_{n-2}, C_{n-1}) \neq \epsilon$. Then the special construction described for $w(X)$ when X has an exceptional configuration as last element and the element before the last is an API call is used.

□

Lemma B.5. *Given a program T and a policy $\mathcal{P}$, if for every execution E of T there exists a co-execution w of T and $\mathcal{A}_\mathcal{P}$ such that $w \downarrow 1 = E$, then the sequence $w(X_E)$ extracted from the extended execution X_E corresponding to this execution is also a co-execution such that $w(X_E) \downarrow 1 = E$ and $w(X_E)$ is closest updating.*

Proof. For each co-execution, a closest updating co-execution can be constructed by postponing the transition of the monitor for a preaction until the configuration which calls this security relevant method is reached and by performing the transition of the monitor right after the return of the security method call if the update is for a postaction.

□

Theorem 6.2 (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 $\mathcal{P}$.*

Proof. ($\Rightarrow$) The result follows in this direction from theorem 5.2, lemma B.5 and lemma B.4.
($\Leftarrow$) The result follows in this direction from lemma B.4 and theorem 5.2. □

B.2 Proof of Theorem 6.4

Theorem 6.4 *The level II annotation of P with embedded state $\overrightarrow{ms}$ is valid if and only if for each execution E of P, the sequence $w(E, \overrightarrow{ms})$ is a method-local co-execution.*

Proof. ($\leftarrow$)
By induction on the length of the execution:

Base Case

Induction Hypothesis: For all executions E_k, $k < n$, $w(E_k, \overrightarrow{ms})$ is a method-local co-execution.

Inductive Step: We consider the shape of C_n:

- C_n is an entry point of an application method: This is the case only if C_{n-1} is an
- C_n is an application method call:
- C_n is an API method call:
- *Unhandled*(C_n): `invokevirtual` instruction or if $n = 0$.
- C_n is not of the above: In this case, C_n is not a sampling point, hence $w(E_n, \overrightarrow{ms}) = w(E_{n-1}, \overrightarrow{ms})$ and the claim holds by the induction hypothesis.

($\rightarrow$)

□

B.3 Proof of Theorem 7.1

Theorem 7.1 *Let P be a program, $\mathcal{P}$ a ConSpec policy, and I(P,$\mathcal{P}$) denote program P inlined for policy $\mathcal{P}$. The level III annotation of I(P,$\mathcal{P}$) is locally valid, and validity is efficiently checkable.*

Proof. (Sketch) We show that the verification conditions resulting from the level III annotation of I(P,$\mathcal{P}$) are valid and efficiently checkable. To simplify the presentation, we consider here post-actions only; the argument is easily adapted to pre-actions and exception actions.

Notice that for level III annotated programs, every instruction is annotated by a non-empty sequence of logical assertions alternated with ghost variable assignments, always starting and ending with a logical assertion. Notice also that $\mathit{Ensures}(\Gamma^*(M))$ and $\mathit{Exsures}(\Gamma^*(M))$ are all equal to the synchronization assertion $\overrightarrow{gs} = \overrightarrow{ms}$ for fully annotated programs. The first and last elements of the annotation sequence of $\mathit{Requires}(\Gamma^*(M))$ is also the synchronization assertion (except for $\langle\texttt{main}\rangle$, in which case $\mathit{last}(\mathit{Requires}(\Gamma^*(M)))$ is again $\overrightarrow{gs} = \overrightarrow{ms}$). Similarly, notice that for all instructions L, where L is not the label of an inlined instruction and is not a security relevant action, $\mathit{last}(A_M^{III}[L])$ is the synchronization assertion.

We assume that the return instruction is not the first instruction of an exception handler, the last element in its annotation sequence is the synchronisation annotation. We also assume that the inlined instructions do not raise exceptions.

Then, a level III annotation of I(P,$\mathcal{P}$) gives rise to a set of verification conditions described as follows.

First, there are three types of verification conditions arising from method compositionality, namely:

- $\mathit{last}(\mathit{Requires}(\Gamma^*(M))) \Rightarrow \mathit{head}(A_M^{III}[1])$,
- $\mathit{last}(A_M^{III}[R]) \Rightarrow \mathit{Ensures}(\Gamma^*(M))$,
- For all instructions L that is not a method call and that can raise an unhandled exception $\mathit{last}(A_M^{III}[L]) \Rightarrow \mathit{Exsures}(\Gamma^*(M))$

where R is the label of the return instruction in method M, and where *last* is a function on sequences returning the last element. The inlined instructions are assumed not to raise any exceptions, so no verification condition for exception raising is generated by these. Additionally, only inlined instructions and method calls change the embedded monitor state, hence the simple form of the verification conditions of the latter type. In the first two cases and in the last case when L is not the label of a method call, the antecedent and the consequent are (syntactically) equal to the synchronisation assertion. These verification conditions are therefore valid, and validity is efficiently checkable.

last case, method call?

Second, every ghost variable assignment $\overrightarrow{g} := ce$ gives rise to a verification condition. If $\alpha \cdot (\overrightarrow{g} := ce) \cdot \alpha'$ is a subsequence of $A_M^{III}[L]$ for some L where α and α' are logical assertions, then $\alpha \Rightarrow \alpha'[ce/\overrightarrow{g}]$ is a verification condition. Due to the normalization performed in the construction of the level III annotation, α must contain a conjunct $\alpha'[ce/\overrightarrow{g}]$. Such verification conditions are therefore valid, validity being efficiently checkable.

Third, every non-method-call instruction $M[L]$ gives rise to a verification condition $\mathit{last}(A_M^{III}[L]) \Rightarrow \mathit{wp}(M[L])$. There are three cases to be considered: (a) if $M[L]$ is a non-inlined instruction with non-inlined successor instructions only, $\mathit{last}(A_M^{III}[L])$ is syntactically equal to $\mathit{wp}(M[L])$ by construction; (b) if $M[L]$ is a non-inlined instruction followed by an inlined instruction (in the case of post-actions only, the latter indicates the beginning of

an inlined block serving to record the current values of the parameters and the object with which the following potentially security relevant instruction is called), then the synchronization assertion $\overrightarrow{gs} = \overrightarrow{ms}$ must appear as a conjunct in both $last(A_M^{III}[L])$ and $wp(M[L])$, and the only other conjuncts in the latter must be either of the shape $Defined^\sharp$ or s[i] = s[i]; (c) if $M[L]$ is an inlined instruction, $last(A_M^{III}[L])$ must contain a conjunct $wp(M[L])$ by construction. In all three cases, the verification condition is valid, validity being efficiently checkable; the only interesting case here is presented by $Defined^\sharp$, the consequent $\overrightarrow{gs} \neq \overrightarrow{\bot}$ of which is implied by $\overrightarrow{gs} = \overrightarrow{ms}$. Similarly, every non-method call instruction $M[L]$ that can raise an exception which is handled by the handler at label H gives rise to the verification condition $last(A_M^{III}[L]) \Rightarrow head(A_M^{III}[H])$. By the assumption that the inlined instructions do not raise an exception, this instruction can not be an inlined instruction. There are two cases to consider: (a) if $M[H]$ is a non-inlined instruction, then both the antecedent and the consequent are the synchronisation annotation; (b) if the handler $M[H]$ is an inlined instruction (which is possible only if it is the first instruction a code inlined for a potentially preaction occuring in the original handler) this case becomes a subcase of the proof for pre-actions, handled similar to the final part of this proof.

Finally, every method-call instruction $M[L]$ calling some method M' gives rise to three types of verification conditions. If the method call is not potentially post-security relevant, these are:

- $last(A_M^{III}[L]) \Rightarrow Requires(\Gamma^*(M'))$,
- $Ensures(\Gamma^*(M')) \Rightarrow head(A_M^{III}[L+1])$, and
- For all handler instructions H of L, $Exsures(\Gamma^*(M')) \Rightarrow head(A_M^{III}[H])$

In the first two formulas, the antecedent and the consequent are (syntactically) equal by construction, and hence valid. The last set of verification conditions are valid and efficiently checkable by the argument for the case when $M[L]$ is not a method-call presented above where $last(A_M^{III}[L])$ should be replaced by $Exsures(\Gamma^*(M'))$.

If $M[L]$ is calling some method M' which is potentially security relevant, the three types of verification conditions are

- $last(A_M^{III}[L]) \Rightarrow Requires(\Gamma^*(M')) \wedge \phi$,
- $Ensures(\Gamma^*(M')) \wedge \phi \Rightarrow head(A_M^{III}[L+1])$, and
- For all instruction handler instructions H of L, $Exsures(\Gamma^*(M')) \wedge \phi \Rightarrow head(A_M^{III}[H])$

where ϕ is the formula $(g_0 = r_0) \wedge \ldots \wedge (g_{n-1} = r_{n-1}) \wedge (g_{this} = r_{this})$. Notice that the invoked method does not change the local variables and the evaluation stack of the caller method (except for popping arguments from the stack and pushing its return value). Then a formula mentioning variables not changed by the invoked method (such as ϕ) can be added to both the pre-and postconditions of the invoked method [3].

The first of these conditions is again easy to show valid, since $Requires(\Gamma^*(M'))$ and all conjuncts in ϕ also appear as conjuncts in $last(A_M^{III}[L])$ by construction. The third set of verification conditions are similar to the last cases of the argument above, when $M[L]$ is calling a non-potentially security relevant action. The only really involved case in the whole proof is the second verification condition.

Let $\alpha_1, \ldots, \alpha_m$ 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 $\neg(g_{\text{this}} : c_1' \vee \ldots \vee g_{\text{this}} : c_p') \ \rightarrow\ \overrightarrow{gs}$, all induced by the policy for the instruction $M[L] =$ `invokevirtual` $(c.m)$ as described in Section 6.2 (cf. After Annotations). Then

the second element of $A_M^{III}[L+1]$ must be a ghost assignment $\overrightarrow{gs} := ce$ where ce is the conditional expression $\alpha_1 \mid \cdots \mid \alpha_m \mid \alpha$. The block inlined immediately after the (potentially post-security relevant) instruction $M[L]$ has the important property that its weakest precondition w.r.t. the head assertion of the first instruction following the block (which is the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$) is the logical assertion

$$\begin{array}{ll} & \bigwedge_{1\leq i\leq m} r_{\text{this}} : c_i' \wedge\ a_b\rho_i' \ \rightarrow\ \overrightarrow{gs} = \overrightarrow{e_E}\rho_i' \\ \wedge & \neg(r_{\text{this}} : c_1' \vee \ldots \vee r_{\text{this}} : c_p') \ \rightarrow\ \overrightarrow{gs} = \overrightarrow{ms} \end{array}$$

where the substitution ρ_i' is defined as $[s[0]/x, r_0/x_0, \ldots r_{n-1}/x_{n-1}, r_{this}/\texttt{this}, \overrightarrow{ms}/\overrightarrow{gs}]$ if $r = (\tau\, x)$ and as $[r_0/x_0, \ldots r_{n-1}/x_{n-1}, r_{this}/\texttt{this}, \overrightarrow{ms}/\overrightarrow{gs}]$ if $r = void$. Therefore, $head(A_M^{III}[L+1])$ must be the logical assertion

$$\begin{array}{ll} & \phi \\ \wedge & \mathit{Defined}^\sharp \\ \wedge & \bigwedge_{1\leq i\leq m} r_{\text{this}} : c_i' \wedge\ a_b\rho_i' \ \rightarrow\ \overrightarrow{ce} = \overrightarrow{e_E}\rho_i' \\ \wedge & \neg(r_{\text{this}} : c_1' \vee \ldots \vee r_{\text{this}} : c_p') \ \rightarrow\ \overrightarrow{ce} = \overrightarrow{ms} \end{array}$$

where ϕ is as explained above, and where $\overrightarrow{ce}$ is the tuple of conditional expressions ce_i, obtained from ce by replacing each expression vector $\overrightarrow{e_E}$ occurring in ce with its i-th component. Now, validity of the verification condition $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi \Rightarrow head(A_M^{III}[L+1])$ is established as follows. The first conjunct ϕ (actually a set of conjuncts) of $head(A_M^{III}[L+1])$ appears as a conjunct in $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi$. The second conjunct $\mathit{Defined}^\sharp$ is implied by $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi$ because $\mathit{Ensures}(\Gamma^*(M'))$ is $\overrightarrow{gs} = \overrightarrow{ms}$, which implies $\overrightarrow{gs} \neq \overrightarrow{\perp}$. Every conjunct $r_{\text{this}} : c_i' \wedge\ a_b\rho_i' \rightarrow \overrightarrow{ce} = \overrightarrow{e_E}\rho_i'$ is valid under the equalities of $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi$, since then every guard $r_{\text{this}} : c_i' \wedge\ a_b\rho_i'$ matches exactly the guard of α_i, and $\overrightarrow{e_E}\rho_i$ is equal to $\overrightarrow{e_E}\rho_i'$. Vality can thus be easily checked mechanically by simple equational reasoning and (syntactic) guard matching. Finally, validity of the conjunct $\neg(r_{\text{this}} : c_1' \vee \ldots \vee r_{\text{this}} : c_p') \rightarrow \overrightarrow{ce} = \overrightarrow{ms}$ is established similarly.

When $M[L]$ can give rise to an exceptional postaction, the last set of verification conditions look slightly different. Notice that our inliner inserts a handler for each such potentially security relevant instruction that handles all types of exceptions. Let the label of the first instruction of this handler to be H for the instruction $M[L]$, then the three verification conditions are:

- $last(A_M^{III}[L]) \Rightarrow \mathit{Requires}(\Gamma^*(M')) \wedge \phi \wedge (g_{\text{pc}} = L)$ and
- $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi \wedge (g_{\text{pc}} = L) \Rightarrow head(A_M^{III}[L+1])$, and
- , $\mathit{Exsures}(\Gamma^*(M')) \wedge \phi \wedge (g_{\text{pc}} = L) \Rightarrow head(A_M^{III}[H])$

where ϕ is the formula $(g_0 = r_0) \wedge \ldots \wedge (g_{n-1} = r_{n-1}) \wedge (g_{this} = r_{this})$. The non-trivial case is then to show that the third verification condition is valid and efficiently checkable. This argument is similar to the argument made above for the non-exceptional case. □