Provably Correct Runtime Monitoring [1]

Irem Aktug [a] Mads Dam [b] Dilian Gurov [a]

[a] *School of Computer Science and Communication, KTH, Sweden*

[b] *ACCESS Linnaeus Center, School of Computer Science and Communication, KTH, Sweden*

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 behavior, and (ii) that the program has a copy of the given monitor embedded into it. As the main application of these results we sketch a simple inlining algorithm and show how the two-level annotations can be completed to produce a fully annotated program which is valid in the standard sense of Floyd/Hoare logic. This establishes the mediation property that inlined programs are guaranteed to adhere to the intended policy. Furthermore, validity can be checked efficiently using a weakest precondition based annotation checker, thus preparing the ground for on-device checking of policy adherence in a proof-carrying code setting.

Email addresses: `irem@csc.kth.se` (Irem Aktug), `mfd@kth.se` (Mads Dam), `dilian@csc.kth.se` (Dilian Gurov).

[1] The work was supported by the EU FP6 IST-STREP-27004 project S^{3}MS, Security of Software and Services for Mobile Systems. Additionally, the second author was partially supported by the ACCESS Linnaeus Center at KTH, the Royal Institute of Technology, and by the Swedish Research Council, projects 2003-6108 and 2007-6436.

1 Introduction

Program monitoring is a firmly established and efficient approach to enforce a wide range of program security and safety properties [27,18,14,16,24,26,30,29,23]. 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 [27,24,26]. A variant, examined by Schneider and Erlingsson [17], is monitor inlining, where target programs are rewritten to include the desired monitor functionality, thus making programs essentially self-monitoring [18,17,16,9]. 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 prepare the ground for the certification of correctly inlined programs in a proof-carrying code framework. To this end, we develop a suitable notion of proof to facilitate automatic proof generation and efficient proof checking. By correctness, we mean here the mediation property, namely that inlined programs are guaranteed to adhere to the intended policy.

We propose a formalization of monitoring and monitor inlining where monitors are security automata that operate on calls to some fixed API from a target program given as an abstract Java Virtual Machine (JVM) class file. Automaton transitions are allowed to depend locally on argument values, heap at time of call and (normal or exceptional) return, and return value. In order to handle a large class of inlined programs, we do not fix the inlining procedure. Instead, we use annotations to characterize self-monitoring programs, which would include any correctly inlined program.

Our main contributions are characterizations, in terms of JVM class files annotated by formulae in a suitable Floyd-like program logic, of the following two conditions on a program relative to a given policy:

(1) that the program is policy-adherent; and the stronger condition
(2) that the program contains a method-local monitor for the policy.

By method-local we mean that the updates to the monitor state do not cross the boundaries of the caller method.

We achieve this using a two-level annotation scheme. In level I annotations, illustrated in figure 1, we specify a correct monitor for the given policy in the program by means of "ghost" variables. The monitor state represented by these variables, which we call the ghost state ($\overrightarrow{gs}$), is updated before or after security relevant actions according to the transition function δ of the security

L_0	$\ldots$
$\vdots$	$\vdots$
	$\left\{\begin{array}{l}\overrightarrow{gs} := \delta(\overrightarrow{gs}, a_{c.m}) \cdot \\ \mathit{Defined}(\overrightarrow{gs})\end{array}\right\}$
L_i	`invokevirtual` $c.m$
$\vdots$	$\vdots$
L_n	`return`

Fig. 1. Level I Annotated Method

	$\{\overrightarrow{gs} = \overrightarrow{ms}\}$
L_0	$\ldots$
$\vdots$	$\vdots$
	$\left\{\begin{array}{l}\overrightarrow{gs} := \delta(\overrightarrow{gs}, a_{c.m}) \cdot \\ \mathit{Defined}(\overrightarrow{gs}) \cdot \\ \overrightarrow{gs} = \overrightarrow{ms}\end{array}\right\}$
L_i	`invokevirtual` $c.m$
$\vdots$	$\vdots$
	$\{\overrightarrow{gs} = \overrightarrow{ms}\}$
L_n	`return`

Fig. 2. Level II Annotated Method

automaton. In the example program, such an update is shown for the action $a_{c.m}$, which is the call to method $c.m$. Transitions that violate the policy result in the ghost state becoming "undefined", indicating an illegal monitor state. Level I annonations assert the ghost state to be a legal monitor state whenever a security relevant action is to be executed and are therefore validated only by policy adherent programs, thus establishing the first condition above. In level II, we extend level I annotations to state at all method boundaries that the inlined monitor (represented by global program variables) is "in sync" with the specified monitor (represented by the ghost variables). We show these extended annotations in figure 2, where $\overrightarrow{ms}$ is used for the vector of global variables that represent the inlined monitor state.

The method-local nature of the embedded monitor enables compositional analysis: validity can be checked per method. Being method-local is not an overly restrictive condition on embedded monitors and is satisfied by all general purpose inliners we know of.

The annotations serve as an important intermediate step towards a decidable annotation validity problem, once the inliner is suitably instantiated. By the above characterizations, the problem of showing correct monitor inlining reduces to proving the validity of the corresponding annotations. For practical monitors, this is not a difficult task. We illustrate this by describing a monitor inlining scheme for which we prove the mediation property. We establish this by describing, for programs inlined by the scheme, how the annotations can be completed to produce a fully annotated program. The resulting verification conditions are valid, thus showing that the inlined programs contain a correct method-local monitor for the intended policy and henceforth policy-adherent. Furthermore, validity can be efficiently decided for these annotations using a bytecode weakest precondition checker thus making them suitable to be used in a proof-carrying code setting to certify monitor compliance to a third party such as a mobile device.

Our results can be seen as providing theoretical underpinnings for the earlier work by Schneider and Erlingsson [16]. 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 matching and adherence 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 is described and its correctness characterized. We also sketch how to produce, for this inliner, fully annotated programs with a decidable validity problem. Section 8 summarizes related work, discussing other monitor inliners, methods for specifying policy adherence and several security frameworks based on proof-carrying code. Finally, in Section 9 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 formalizations of the JVM such as [20]). Here we only present components of the JVM that are essential for the definitions in the rest of the text.

2.1 Types and Values

We denote bytecode programs with T. We fix a set of class names $c \in \mathbb{C}$, a set of method names $m \in \mathbb{M}$, and a set of field names $f \in \mathbb{F}$. Primitive type set *PrimType* consist of the types `int` and `string`. The type `Void` is used for methods that do not return a value. A type $\tau \in \mathit{Type}$ is either a primitive type, a class c, the type `Void` or the type `Null`. We let $\gamma \in (\mathit{Type})^*$ range over tuples of types.

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, respectively, and make up the values of type *PrimType*, which are called the primitive values *PrimVal*. The sets determined by the types `Void` and `Null` are singletons consisting of the values *void* and `null`, respectively.

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 the class which this object is a member of, declares f.

We also introduce a static heap $sh : c \times f \rightharpoonup \mathit{Val}$ that stores the values of the static variables of a class. We assume sh_0^{T} to be the initial mapping which maps each static variable of a class reachable through the program T to its initial value as given by its class definition. In this sense, we make two assumptions: the static variables of all reachable classes are assumed to be initialized to constant values (i.e. we disregard static initializers) and the initialization is assumed to have been done *before* the program starts executing. The first assumption can be dropped by extending our approach to handle static initializers, which is straightforward to perform.

Each class determines a set of fields and methods defined for that type through its declaration. The class declarations induce a hierarchy given by the subclassing partial preorder <: on the set $\{\texttt{Null}\} \cup \mathbb{C}$. We write $c_1 <: c_2$ if c_1 is a subclass of (extends) c_2. `Null` is the bottom element with respect to this ordering: $\forall \tau \in \{\texttt{Null}\} \cup \mathbb{C}.\ \texttt{Null} <: \tau$. 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.

2.2 Methods

Method definitions are modeled through an environment Γ taking method references to their definitions. The environment Γ is elided where possible. We assume furthermore a partitioning on the set of methods which divides the set into API methods and application methods. 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 $(c.m)$, $(c.m) : \gamma \rightarrow \tau$ when 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. The method body (the exception handler array) of M is denoted P_M (H_M) when the environment Γ is clear from the context. For each program, we assume that there exists a main method method which does not have a class defining it. We identify this method with the special reference $\langle \texttt{main} \rangle$.

A method body P is a partial function from ω to the set of instructions such that $ADDR_P = Dom(P)$ 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

- M is the currently executing method.
- The *program counter* pc is an index into the currently executing instruction array, i.e. it is a member of $Dom(P)$ where P is the body of M. The configuration C is *calling*, if $P(pc)$ is an invoke instruction, and it is *returning normally*, if $P(pc)$ is a return instruction.
- The *operand stack* s is the stack of values (i.e. 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. For exceptional configurations, the current program counter and executing method is given by the frame below the exceptional frame. Then, C is *returning exceptionally* if there is no handler for this exception and the current instruction label in the currently executing method. Configuration C is *returning* if C is either returning normally or exceptionally. Finally, if C is exceptional and there is a single frame in the activation record, then the program is *exiting exceptionally*.

Machine Transitions We assume a transition relation $\longrightarrow_{\mathrm{JVM}}$ on JVM configurations. Such an operational semantics can be found for instance in [20].

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. [28]).

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 [33]. In this sense, we do not practice *complete mediation* [34]. 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 [35]. Here, we view a *security automaton* over alphabet A as a deterministic automaton $\mathcal{A} = (Q, \delta, q_0)$ where Q is a countable set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times A \rightharpoonup Q$ is a (partial) transition function. All $q \in Q$ are viewed as accepting. A security automaton $\mathcal{A}$ induces a security policy $\mathcal{P}_{\mathcal{A}} \subseteq 2^{A^* \cup A^\omega}$ through its language $L_{\mathcal{A}}$ by $\mathcal{P}_{\mathcal{A}}(X) \Leftrightarrow X \subseteq L_{\mathcal{A}}$.

In this study, we focus on security automata which are induced by policies in the ConSpec language (see Section 4) and therefore are named *ConSpec automata*. The security relevant actions are method calls, represented by the class name and the method name of the method, along with a sequence of values that represent the actual arguments. We partition the set of security relevant actions into *pre-actions* $A^\flat \subseteq \mathbb{C} \times \mathbb{M} \times \mathit{Val}^* \times \mathbb{H}$ and *post-actions* $A^\sharp \subseteq \mathit{RVal} \times \mathbb{C} \times \mathbb{M} \times \mathit{Val}^* \times \mathbb{H} \times \mathbb{H}$, corresponding to method invocations and returns. Both types of actions may refer to the heap prior to method invocation, while the latter may also refer to the heap upon termination and to a return value from $\mathit{RVal} = Val \cup \{\mathrm{exc}\}$ where exc is used to mark exceptional return from a method call[2]. 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 [15] 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. This was a design choice taken to allow formal treatment of monitoring. In particular, the update function induced by the guarded-commands should be effectively recreated by a weakest precondition propagation on inlined code compiled from these commands. For reasons that will become clear later in the text, this property leads to a decidable problem of correct inlining.

As an extension to PSLang, 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. In this work, we focus on policies on a single execution of an application, however.

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[3]. Assume further that method `Open` of class `Connection` is used for

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

[3] The methods used in the policy are not part of any standard Java API but have been chosen for the sake of the example.

```
1    SCOPE Session
2    SECURITY STATE
3         bool accessed   = false;
4         bool permission = false;
5
6    BEFORE File.Open(string path, string mode, string access)
7    PERFORM
8         mode.equals("CreateNew")                            -> { skip; }
9         mode.equals("Open") && access.equals("OpenRead")  -> { accessed = true; }
10
11   EXCEPTIONAL File.Open(string path, string mode, string access)
12   PERFORM
13        FALSE -> { skip; }
14
15   AFTER bool answer = GUI.AskConnect()
16   PERFORM
17        answer  -> { permission = true; }
18        !answer -> { permission = false; }
19
20   BEFORE Connection.Open(string type, string address)
21   PERFORM
22        !accessed || permission -> { permission = false; }
```

Fig. 3. Example Policy in ConSpec

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 is specified in ConSpec as shown in figure 3.

We first specify that the policy applies to each single execution of an application (line 1). 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 (lines 2-4). The example policy contains three *event clauses* that state the conditions for and effect of the security relevant actions: call to the method `File.Open` (lines 6-9), exceptional return from the method `File.Open` (lines 11-13), call to the method `Connection.Open` (lines 15-18) and normal return from the method `GUI.AskConnect` (lines 20-22). 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.

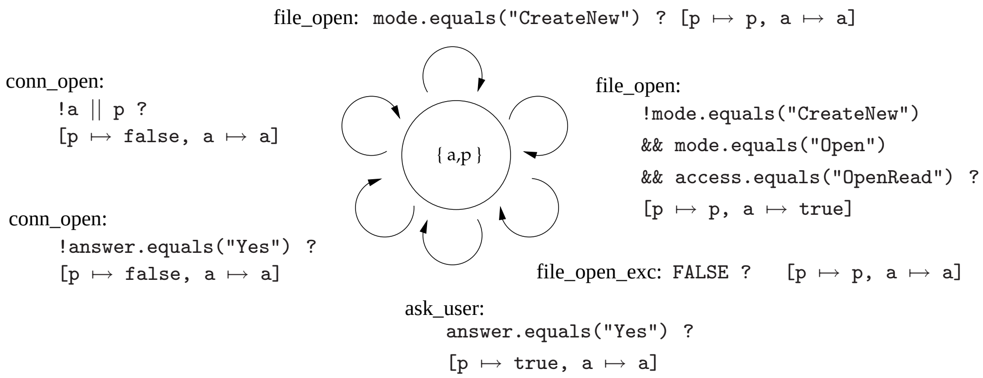
The formal semantics of ConSpec policies is defined in terms of *symbolic security automata*, which in turn induce ConSpec automata. Fix a set *Svar* of security state variables and a set *Var* of program variables.

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) $Init_s : q_s \rightarrow Val$ *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*
$A_s^\sharp \subseteq \{(\{PrimType \cup \mathbb{C}\} \times Var) \cup \texttt{Void} \cup \{\text{exc}\}\} \times \mathbb{C} \times \mathbb{M} \times (Type \times Var)^*$ *are* symbolic post-actions*;*
(iv) $\delta_s = \delta_s^\flat \cup \delta_s^\sharp$ *is a* symbolic transition relation*, where:*
$\delta_s^\flat \subseteq A_s^\flat \times BoolExp \times (q_s \rightarrow Exp)$ *and*
$\delta_s^\sharp \subseteq A_s^\sharp \times BoolExp \times (q_s \rightarrow Exp)$
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 event clause gives rise to one symbolic action, and each guarded command of the clause gives rise to a symbolic transition consisting of the security relevant action 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 figure 4 we illustrate the construction on the earlier example, using ”a” for `accessed` and ”p” for `permission`. For instance, the first event clause of the policy gives rise to the action file_open and the automaton has two transitions for this action, one per guard. The transition on the top center of the figure is for the first guard and does not perform any updates on the current security state. The transition for the second guard (on the right of the figure) sets



$A^\flat_s$={file_open, conn_open}

file_open=(File,Open,(string path, string mode, string access))

conn_open=(Connection,Open,(string type, string address))

$A^\sharp_s$={ask_user, file_open_exc}

ask_user=(string answer, GUI,AskConnect,())

file_open_exc=(exc,File,Open,(string path, string mode, string access))

$Init_s$ =[p ↦ false, a ↦ false]

Fig. 4. Symbolic Automaton for the Example Policy

accessed to true, on the other hand, and applies only if the first guard does not hold.

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 $x = r \in \{void, \mathrm{exc}\}$. 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 $b \in B$.

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

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

Table 1
Security relevant actions induced by configurations

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 first formalize the infinite or finite sequence of security relevant actions induced by a target program execution. 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 table 1. If none of the conditions of the table hold, the corresponding action is ϵ.

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:

$$srt_A(\epsilon) = \epsilon \qquad\qquad srt_A(C) = act^\flat_A(C)$$
$$srt_A(C_1 C_2 \cdot w) = act^\flat_A(C_1) \cdot act^\sharp_A(C_1, C_2) \cdot srt_A(C_2 \cdot w)$$

Then a target program T *adheres* to a policy $\mathcal{P}$, if the security trace of each execution of T is in the 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 assume without loss of generality 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:

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

$$(C, q) \downarrow 1 = C$$

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

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

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

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

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

$$\mathit{extract}(w) \in (E^{\flat} \cup E^{\sharp})^{*} \cup (E^{\flat} \cup E^{\sharp})^{\omega}$$

In other words, an interleaving is a co-execution, if the potential derivation it extracts corresponds to a real derivation.

A monitor is *conservative* if all monitored executions are also executions of the original program, i.e. if the monitor does not introduce new behavior. When monitoring is done by ConSpec automata in the sense captured by the notion of co-execution, the monitor is correct and conservative.

Theorem 5.2 *(Correctness of Monitoring by Co-execution) Let T be a program, and $\mathcal{P}$ a policy. The following holds, where A is the action set of $\mathcal{A}_{\mathcal{P}}$:*

$$\{w \downarrow 1 \mid w \textit{ is a co-execution of } T \textit{ and } \mathcal{A}_{\mathcal{P}}\} = \{E \in \Pi(T) \mid srt_A(E) \in L_{\mathcal{A}_{\mathcal{P}}}\}$$

We present the proofs to the results of the paper in Appendix B.

A corollary of this result is that the set of executions of a program that obeys the policy are identical to the set of executions of the program monitored for the policy.

Corollary 5.3 *Program T adheres to policy $\mathcal{P}$ if, and only if, for each execution E of T there is a co-execution w for the automaton $\mathcal{A}_{\mathcal{P}}$ such that $w \downarrow 1 = E$.*

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 annotation* (or level II), we add assertions that check at all relevant program points that the actual inlined monitor (represented by global program variables) agrees with the specified one (represented by ghost variables).

6.1 Annotation Language

We specify self-monitoring using annotations in a Floyd-style logic for bytecode, which is a specialization of the program logic of Bannwart and Müller [8]. As an extension to their logic, our annotation language makes use of "ghost" variables. These are essentially specification variables that can be assigned values by a multi-assignment statement.

Methods are equipped with annotations consisting of assertions on the extended state (current configuration and current ghost variable environment), and ghost variable assignments. We first introduce the syntax of this annotation language.

Assertions Let g range over ghost variables, i over natural numbers, and let Op range over a standard, not further specified, collection of unary and binary operations on strings and integers, while Bop range over boolean operations. Expressions e and assertions a have the following shape:

$$e ::= \bot \mid v \mid g \mid e.f \mid \mathrm{s}[i] \mid \mathrm{r}i \mid Op\ e \mid e\ Op\ e$$

$$a ::= e\ Bop\ e \mid e : c \mid e <: c \mid \neg a \mid a \wedge a \mid a \vee a$$

Here, $\mathrm{s}[i]$ is the value at the i^{th} position of the current operation stack, if defined, and $\bot$ otherwise, $e : c$ is a class membership test and $e <: c$ is a subclass membership test. The notation $\mathrm{r}i$ denotes the i^{th} local variable. The assertions are evaluated with respect to extended states. Extended states consist of a program configuration C and a ghost environment σ that maps ghost variables to integer values, addresses or the value $\bot$, which captures that the variable is undefined. Referring to standard denotational semantics, we assume a semantic function $\|a\|(C, \sigma)$ that returns, for assertion a and extended state (C, σ), a truth value.

Annotation Example The following example assertion states that if the address at the top of the stack points to an object of type `GUI` in the heap, then the ghost variables g_{a} and g_{p} are both defined.

$$\mathrm{s}[0] : \texttt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)$$

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

$$\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} \mid \cdots \mid a_m \rightarrow \overrightarrow{e_m} \qquad (1)$$

where $\overrightarrow{gs}$ is a vector of ghost variables and $\overrightarrow{e_i}$ $(1 \leq i \leq m)$ are vectors of expressions, such that the arities (and types) of $\overrightarrow{gs}$ and the $\overrightarrow{e_i}$ match. The multi-assignment is performed with vector $\overrightarrow{e_i}$ if guard a_i is the first guard (from left) that holds in the current extended state. If no guard is true, the ghost state is assigned the constant vector with all elements $\bot$ and the arity matching to that of $\overrightarrow{gs}$. This is the case, in particular, when $m = 0$ in (1) above. This case is written as follows: $\overrightarrow{gs} := ()$. The right hand side of conditional assignments are referred to as *conditional expressions* and are denoted by ce.

Method Annotations A target program is annotated by an extended environment Γ^*, which maps method references M to tuples $(P, H, A,$ *Requires*, *Ensures*$)$ such that A is an assignment to each program point $n \in \mathit{Dom}(P)$ of a sequence, ψ, of atomic annotations, i.e. assertions and ghost variable assignments. *Requires* also consists of a sequence of atomic annotations, while *Ensures* is a single assertion. These two clauses do not mention method arguments or return values. The precondition *Requires* is allowed to contain ghost assignments, since we occasionally use these clauses to initialize ghost variables.

Annotation Semantics In the absence of ghost variable assignments the notion of annotation validity is the expected one, i.e. the assertions annotating a given program point (or the point of exceptional return) hold whenever control is at that program point. To extend this account to ghost variables, the ghost variable assignments should be given a suitable semantics. We present such a semantics in this section, which essentially treats ghost variables as program variables. For this purpose, the program state is extended by a store for ghost variables which is altered only by ghost variable assignments, method calls and returns.

The rewrite semantics we use for annotated programs is built on top of the transition relation $\longrightarrow_{\mathrm{JVM}}$ of section 2 and is shown on table 2. The semantics uses extended configurations that are quintuples of the form $(\psi, C, \sigma, \Sigma)$ such that ψ is the sequence of annotations remaining to be evaluated for the current program point of C, and σ is the ghost environment introduced above, mapping ghost variables to values. Each ghost environment σ can be partitioned to the global and local ghost environments σ_l and σ_g where the domain of σ_g is the variables of the ghost state, which are declared and initialized in the beginning of $\langle \texttt{main} \rangle$ and the domain of σ_l is all other ghost variables that have been set values in the current method. Finally Σ is a sequence of local ghost (variable) environments. The top element of Σ is the local ghost environment that belongs to the caller of the current method. Each method call causes a new local ghost environment σ_l^0 to be created, which is defined as $[g_{\mathrm{pc}} \mapsto 0]$. Note that local ghost variables are not allowed to occur in *Requires* or *Ensures* clauses, like it is the case for local program variables.

We overload M, *pc*, A, *Requires*, *Ensures*, to refer to the first, second, third, fourth, and fifth projections on configurations, respectively. *MethodRet* holds of a configuration if the program counter of the top frame points to a return instruction. *MethodCall* holds of a configuration if the program counter of the top frame points to a method invocation instruction, which resolves to an *application* method call. Notice that these predicates can not be satisfied simultaneously. The predicate *Unhandled* holds of a configuration if it has an exceptional frame on top of the frame stack, and Γ^* does not contain a handler

$$(1) \qquad \frac{Assert(a, C, \sigma)}{\Gamma^* \vdash (a\psi, C, \sigma, \Sigma) \rightarrow (\psi, C, \sigma, \Sigma)}$$

$$(2) \qquad \frac{\|a_1\|(C, \sigma) = true, \quad m > 0}{\begin{array}{c}\Gamma^* \vdash ((\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} | \cdots | a_m \rightarrow \overrightarrow{e_m})\psi, C, \sigma, \Sigma) \rightarrow \\ (\psi, C, \sigma[\overrightarrow{gs} \mapsto \| \overrightarrow{e_1} \| (C, \sigma)], \Sigma)\end{array}}$$

$$(3) \qquad \frac{\|a_1\|(C, \sigma) \neq true, \quad m > 0}{\begin{array}{c}\Gamma^* \vdash ((\overrightarrow{gs} := a_1 \rightarrow \overrightarrow{e_1} | \cdots | a_m \rightarrow \overrightarrow{e_m})\psi, C, \sigma, \Sigma) \rightarrow \\ ((\overrightarrow{gs} := a_2 \rightarrow \overrightarrow{e_2} | \cdots | a_m \rightarrow \overrightarrow{e_m})\psi, C, \sigma, \Sigma)\end{array}}$$

$$(4) \qquad \frac{\cdot}{\Gamma^* \vdash ((\overrightarrow{gs} := ())\psi, C, \sigma, \Sigma) \rightarrow (\psi, C, \sigma[\overrightarrow{gs} \mapsto \overrightarrow{\bot}], \Sigma)}$$

$$(5) \qquad \frac{C \longrightarrow_{\mathrm{JVM}} C' \quad \neg(MethodCall(C) \vee MethodRet(C)) \wedge \neg Exc(C'))}{\Gamma^* \vdash (\epsilon, C, \sigma, \Sigma) \rightarrow (A(\Gamma^*(M(C')))(pc(C')), C', \sigma, \Sigma)}$$

$$(6) \qquad \frac{C \longrightarrow_{\mathrm{JVM}} C', \quad MethodRet(C) \vee Unhandled(C)}{\Gamma^* \vdash (\epsilon, C, \sigma_g \uplus \sigma_l, \sigma_l' \cdot \Sigma) \rightarrow (Ensures(\Gamma^*(M(C))), C', \sigma_g \uplus \sigma_l', \Sigma)}$$

$$(7) \qquad \frac{C \longrightarrow_{\mathrm{JVM}} C', \quad MethodCall(C) \wedge \neg Exc(C')}{\begin{array}{c}\Gamma^* \vdash (\epsilon, C, \sigma_g \uplus \sigma_l, \Sigma) \rightarrow \\ (Requires(\Gamma^*(M(C'))) \cdot A_{M(C')}[1], C', \sigma_g \uplus \sigma_l^0, \sigma_l \cdot \Sigma)\end{array}}$$

$$(8) \qquad \frac{C \longrightarrow_{\mathrm{JVM}} C' \quad \neg Exc(C) \wedge Exc(C')}{\Gamma^* \vdash (\epsilon, C, \sigma, \Sigma) \rightarrow (\epsilon, C', \sigma, \Sigma)}$$

Table 2
Operational Semantics of Annotated Programs

for that exception in the current method. Finally, *Exc* holds of a configuration that has an exceptional frame on the top of the stack.

The condition $Assert(a, C, \sigma)$ in Rule (1) always returns true, and does not effect the execution. But as a side-effect causes the predicate argument a to be "asserted", e.g. to appear on some output channel. The asserted predicate is valid if $\|a \| (C, \sigma)\|$ returns *true*. Rules (2), (3) and (4) capture the ghost variable assignment semantics as described above. Rule (5) is for intra-procedural execution, and applies to exception handling steps, but not to exception raising steps, which are handled by rule (8). Rule (6) causes any assertions in *Ensures* to be asserted at times of method exit, which are method returns and exceptional exits. Note also that the values assigned to the local ghost variables σ_l by the current method are discarded as the method terminates, and instead the environment is updated to use the assignments σ_l' of the calling method. Similarly, if the current instruction is a method call to an application method and executes without raising exceptions, rule (7) causes all assertions in the *Requires* clause of the called method to be asserted. When a new method starts executing, the local ghost environment of the caller method are pushed to the stack and the ghost environment uses the environment σ_l^0. Finally, when the execution of an instruction raises an exception, no predicates are asserted, as captured by rule (8).

The initial extended configuration $(\psi_0, C_0, \sigma_0, \Sigma_0)$ of program T is as follows: ψ_0 is $\mathit{Requires}(\Gamma^*(\langle \texttt{main} \rangle)) \cdot A_{\langle \texttt{main} \rangle}[1]$, C_0 is the initial configuration of T, $\sigma_0 = \sigma_l^0 = [g_{\mathrm{pc}} \mapsto 0]$, $\Sigma_0 = \epsilon$.

Definition 6.1 (Validity of an Annotated Program) *A program annotated according to the rules set up above is* valid *for the extended environment* Γ^**, if all predicates asserted as a result of a* Γ^**-derivation* $(\psi_0, C_0, \sigma_0, \Sigma_0) \to \cdots \to (\psi_n, C_n, \sigma_n, \Sigma_n) \to \cdots$ *are valid, where* $(\psi_0, C_0, \sigma_0, \Sigma_0)$ *is the initial extended configuration of the program.*

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. Finally, we assume without loss of generality that there are no jumps to instructions below method invocations, since such jumps can be eliminated by inserting a no-effect instruction (such as `nop`) after the invocation instruction and redirecting the jump to the instruction after.

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 T and a policy $\mathcal{P}$. Let $\mathcal{A}_s = (q_s, A_s, \delta_s, Init_s)$ be the symbolic automaton induced by $\mathcal{P}$, and let $q_s = \{s_1, \ldots, s_n\}$. We define the set $A^e_s \subseteq A^\sharp_s$ 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 $E : q_s \longrightarrow Exp$, 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 Annotations Further below, we define 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 ($k = 2$). The predicate *Handler* holds for a label L and a method M if L is a destination of some exception handler, i.e. $(L_1, L_2, L, c) \in H_M$ for some labels L_1, L_2, and class name c. In addition, we define $\mathit{Exc}(L, M)$ as the sequence of all annotations $A^e_M[L'][2]$ where L' is a security relevant instruction and there exists an exception handler $(L_1, L_2, L, c) \in H_M$ such that $L_1 \leq L' < L_2$, and as ϵ if such an L' does not exist.

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

$$\mathit{Requires}^{\mathrm{I}}_M = \begin{cases} (\overrightarrow{gs} := \overrightarrow{e_{\mathit{Init}_s}}) & \text{if } M = \langle \texttt{main} \rangle \\ \epsilon & \text{otherwise.} \end{cases}$$

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

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

The annotation $\mathit{Requires}_{\langle \texttt{main} \rangle}$ initializes the ghost state using function Init_s of the automaton. In the following we define the annotation arrays mentioned in the above definition.

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 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 is of the form $M[L] = \texttt{invokevirtual}\ (c.m)$ with $(c.m) : (\gamma \rightarrow \tau)$ and $|\gamma| = n$ and $RS((c, m), A^{\sharp}_s \setminus A^{e}_s) = \{c'_1, \ldots, c'_p\}$, then the precondition of the instruction saves the arguments and the object in ghost variables:

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

$A^I[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\left\{\begin{array}{l} g_{\text{this}} := \mathrm{s}[0] \cdot \\ g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \end{array}\right\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\left\{\begin{array}{l} (g_{\mathrm{a}}, g_{\mathrm{p}}) := \\ ((g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \mathrm{s}[0]) \rightarrow (g_{\mathrm{a}}, \mathit{true}) \quad \mid \\ ((g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \neg\mathrm{s}[0]) \rightarrow (g_{\mathrm{a}}, \mathit{false}) \mid \\ (\neg(g_{\text{this}} : \texttt{GUI})) \rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \end{array}\right\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
	L13	`ireturn`

Fig. 5. An application method with level I annotations for the example policy

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

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

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

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

where the α_k are the guarded expressions

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

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

Level I Annotation Example A level I annotated application method for the example policy 3 is shown in figure 5. The ghost state is represented by the ghost variables g_{a} and g_{p}, i.e. $\overrightarrow{gs} = (g_{\mathrm{a}}, g_{\mathrm{p}})$. (The setting of the ghost variable g_{pc} is ignored since the policy does not include an exceptional clause.) The annotations are valid if the class `GUI` does not have any subclasses. The annotations are identical as long as all subclasses of this class overrides `AskConnect`.

The annotation array $A^\flat_M$ is defined similar to $A^\sharp_M$ except that the transitions of the automaton on pre-actions are considered. The values of the arguments of the security relevant instruction can be obtained by accessing the stack directly, so the argument names in the guards and update expressions of the symbolic automaton should be substituted with corresponding stack positions in this case.

The exceptional annotation array $A^e_M[i][k]$ is defined considering transitions of the automaton on exceptional post-actions (the set A^e). The precondition $A^e_M[L][0]$ of an instruction $M[L]$ that may cause an exceptional post-action saves its label L in the ghost variable g_{pc}, in addition to recording the object and arguments to the method. The conditions of the ghost variable update placed in the precondition to the corresponding handler $A^e_M[L][2]$, then include the conjunct $g_{\mathrm{pc}} = L$ to check that the exception was indeed raised by this instruction. The ghost variable g_{pc} is set back to 0 after the update in the annotation (i.e. in $A^e_M[L][2]$) or if the method never raises an exception (i.e. in $A^e_M[L][1]$).

The formalizations are presented here for completeness.

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 is not an `invokevirtual` instruction or is of the form $M[L]$ = `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 is of the form $M[L]$ = `invokevirtual` $(c.m)$ with $(c.m) : (\gamma \rightarrow \tau)$ 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:

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

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

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

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, $\alpha = \neg(\mathrm{s}[n] : c'_1 \vee \ldots \vee \mathrm{s}[n] : c'_p) \ \rightarrow\ \overrightarrow{gs}$.

Exceptional Annotations For every method M, the elements of the exceptional annotation array $A^e_M[L]$ are defined for each label L as follows:

(i) If the instruction is not an `invokevirtual` instruction or is of the form $M[L] =$ `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 is of the form $M[L] =$ `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:

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

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

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

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 caught exception 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

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

and where class c'' defines (c'_i, m) and there exists $a^e_s = (\text{exc}, c'', m, (\tau_0 x_0, \ldots, \tau_{n-1}x_{n-1}))$, $a^e_s \in A^e_s$ 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 *(Level I Characterization) The level I annotation of program T for policy $\mathcal{P}$ is valid if, and only if, T adheres to $\mathcal{P}$.*

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*, that is 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 level II annotations can uniformly treat all methods that can be called, as a result of virtual method resolution, by the same instruction: The specified and the embedded states are synchronized at all call points, not just at the points of security relevant method call.

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 $\mathit{Ensures}(\Gamma^*(M))$.

Level II Annotation Example A level II annotated application method for the example policy of section 4 is shown in figure 6. This is an augmented version of figure 5, where the embedded state consists of the static fields `accessed` and `permission` of the `SecState` class. The *Ensures* clause is the synchronization

$A^{II}[L]$	L	$M[L]$
	L1	`aload r0`
	L2	`getfield gui`
	L3	`dup`
	L4	`astore r1`
$\left\{\begin{array}{l} g_{\text{this}} := \mathrm{s}[0] \cdot \\ g_{\text{this}} : \texttt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \cdot \\ (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}) \end{array}\right\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\left\{\begin{array}{l} (g_{\mathrm{a}}, g_{\mathrm{p}}) := \\ ((g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \mathrm{s}[0]) \rightarrow (g_{\mathrm{a}}, \mathit{true}) \quad \mid \\ ((g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \wedge g_{\text{this}} : \texttt{GUI} \wedge \neg \mathrm{s}[0]) \rightarrow (g_{\mathrm{a}}, \mathit{false}) \mid \\ (\neg(g_{\text{this}} : \texttt{GUI})) \rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \end{array}\right\}$	L6	`istore r2`
	L7	`aload r1`
	L8	`instanceof GUI`
	L9	`ifeq L12`
	L10	`iload r2`
	L11	`putstatic SecState/permission`
	L12	`iload r2`
$\left\{ (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission}) \right\}$	L13	`ireturn`

Fig. 6. An application method with level II annotations for the example policy assertion

$$(g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \texttt{SecState.permission})$$

and *Requires* $= \epsilon$. The annotated method is valid since the embedded state is updated as is described by the policy, after a call to the method `GUI.AskConnect`.

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

Consider a program T with a level II annotated environment Γ^*. Consider an execution $E = C_0 C_1 \cdots$ from an initial configuration C_0 of T. We sample the embedded state $\overrightarrow{ms}$ 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 three conditions holds:

(1) The top frame of C_i has the shape (M, pc, s, lv), and $M[pc]$ is either an `invokevirtual` instruction, or a return instruction.
(2) The configuration C_{i-1} has the shape $(M, pc, s, lv) : R; h$ where $M[pc]$ is an `invokevirtual` instruction, and C_i has the shape $(N, 1, \epsilon, lv')\ (M, pc, s, lv) : R; h$.

(3) Alternatively, C_i is of the shape $(b)_e(M, pc, s, lv) : R; h$ where there is no handler that covers label pc for b in M.

We can then construct a sequence $w(E, \overrightarrow{ms}) = (C_0, q_0)(C_1, q_1) \cdots$ such that:

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

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

The role of the sequence $w(E, \overrightarrow{ms})$ is roughly similar to the role of interleavings in section 5. However, a slightly different treatment is needed here since the sequence $q_0 q_1 \cdots$ may not necessarily correspond to an automaton run. This is so for the case of a postaction followed by a preaction. Then 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, we define the operation $\mathit{extract}_{\mathrm{II}}$, taking sequences w to strings over the alphabet $Q \cup A \cup \{\mathtt{brk}\}$ where $\mathtt{brk}$ is a distinguished symbol, by the following conditions:

- $\mathit{extract}_{\mathrm{II}}((C_1, q_1)(C_2, q_2)w) = q_1\ \mathit{act}^\flat(C_1)\ \mathit{act}^\sharp(C_1, C_2)\ q_2\ \mathit{extract}_{\mathrm{II}}((C_2, q_2)w)$, if C_1 is an API method call.
- $\mathit{extract}_{\mathrm{II}}((C_1, q_1)(C_2, q_2)w) = q_1 \mathtt{brk} q_2\ \mathit{extract}_{\mathrm{II}}((C_2, q_2)w)$, if C_1 is an application method call and C_2 is not exceptional, i.e. C_2 is an entry point to the method.
- $\mathit{extract}_{\mathrm{II}}((C, q)w) = q \mathtt{brk} q\ \mathit{extract}_{\mathrm{II}}(w)$, if C is a return point from an application method, either normal or exceptional.
- $\mathit{extract}_{\mathrm{II}}((C_1, q_1)(C_2, q_2)w) = \mathit{extract}_{\mathrm{II}}((C_2, q_2)w)$, if none of the above conditions hold.
- $\mathit{extract}_{\mathrm{II}}((C, q)) = q\ \mathit{act}^\flat(C)$ if C is a method call.
- $\mathit{extract}_{\mathrm{II}}(\epsilon) = \epsilon$

Definition 6.3 (Method-local Co-execution) *Let*

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

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

$$extract_{\mathrm{II}}(w) \in (\Sigma_1^* \cup \Sigma_1^\omega) \setminus (\Sigma_0^* \cdot \Sigma_2 \cdot (\Sigma_0^* \cup \Sigma_0^\omega))$$

We can then extend theorem 6.2 to the situation where a target program T has a monitor for the given policy inlined into it.

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

In the next section, we describe and prove correct a scheme of inlining, which satisfies both the assumptions of synchronous and method-local update. The execution of a program that is proven to be inlined correctly in this manner then always yields method-local co-executions.

7 Correctness of Inlining

We use the annotation scheme described in the previous section to show that programs inlined by a class of inliners in the flavor of PoET/PSLang [16,15] are self-monitoring. The proof assumes that inlined blocks satisfy a certain property, which we pin down. As an example we first describe a simple, method-local monitor inlining scheme. Then we show for these inlined programs how level II annotations can be efficiently completed to produce fully annotated code, thus reducing the policy adherence problem to checking the validity of the full annotations.

7.1 A Simple Inlining Scheme

The inlining scheme 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.

Storing the Security State The inliner adds a single class definition to the program. The class stores the embedded state in its static fields. Since this new class is not in the previous name space, the embedded state is safe from interference by the target program. Here, we assume that `SecState` is a fresh name for the target program, and use this name for referring to the class storing the embedded state.

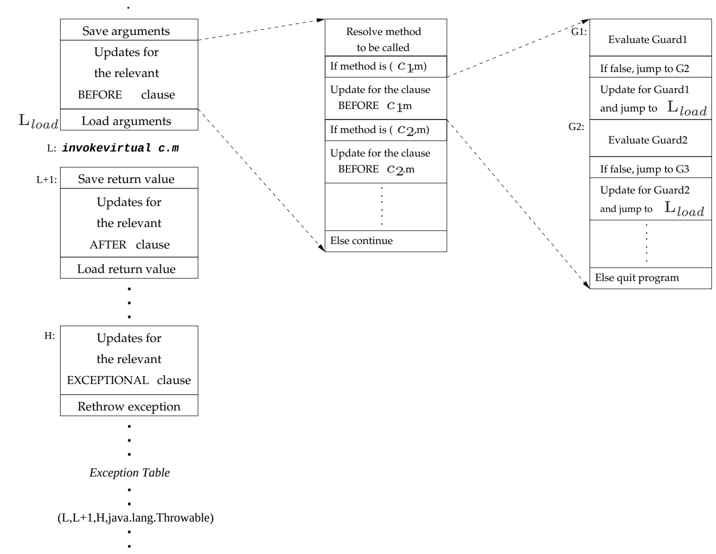


Fig. 7. Inlining of an Instruction

Compiling Policy Body to Bytecode The first part of the transformation compiles the policy to bytecode, and is independent of the method(s) for which inlining is to be performed. For each clause in the policy, a code fragment is produced. These fragments are inlined in application methods in the rewriting stage.

Each clause of a policy consists of an event modifier, an event specification and a list of guarded commands. The created code 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 [4] .

The variables that occur in an event clause are of three kinds: security state variables, method arguments and fields of method arguments. Security state variables are stored in the `SecState` class. Since the fields storing the embedded state are static, they are created and initialized as soon as the class is loaded to the JVM. Actual arguments of a method (including the reference to the object it operates on) reside on top of the stack immediately before the method invocation. In order to use argument values while computing the new values of the security state variables, the arguments are copied from the stack to local variables that are not used by the original program. Since local variables of the calling method are not affected by the execution of the callee, argument values stored in this manner can be accessed even after the control returns to the calling method. Finally, fields of arguments are accessed using the arguments and the heap. Notice that ConSpec policies specify only updates to security state variables. The compiled code uses fresh variables for both the security state and storing arguments. Therefore inlining does not modify the original program behavior beyond forcing its exit upon violation.

[4] In order to abort the program, the method `System.exit()` can be used in standard API's, so that a security violation is distinguished from a normal return.

Rewriting Methods According to Policy The rewriting process consists of identifying method invocation instructions that lead to security relevant actions (security relevant instructions), and for each such instruction, inserting code produced by policy compilation in an appropriate manner. The inlined code is depicted for a single instruction in figure 7. The inliner inserts, immediately before the security relevant instruction, code that records the object the method is called for, and the arguments (and possibly parts of the heap) in local variables. Then, code for the relevant BEFORE clauses of the policy (if any) is inserted. Next, the object and the method arguments are restored on the stack. If there are AFTER clauses in the policy for the instruction, first the return value (if 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[5] .

It is straightforward to implement this scheme as Java bytecode includes instructions or API methods that map to the basic arithmetic and string operations included in the expression language of ConSpec. We have implemented such an inliner, which is available at [1].

7.2 Correctness of Inlining

We first describe how level II annotations for programs inlined with an inliner following the scheme described above can be efficiently completed to (equivalent) level III annotations. We then show that validity of level III annotations

[5] This can be accomplished using the `instanceof` instruction or using the Reflect API.

$M[L]$	$wp(M[L])$
`dup`	$unshift((head(A_M[L+1]))[\mathrm{s}[1]/\mathrm{s}[0]])$
`iload r / aload r`	$unshift((head(A_M[L+1]))[\mathtt{r}/\mathrm{s}[0]])$
`istore r / 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'	$(\mathrm{s}[0]=0 \Rightarrow shift(head(A_M[L'])))\wedge$ $(\neg(\mathrm{s}[0]=0) \Rightarrow shift(head(A_M[L+1])))$
`instanceof` c	$\mathrm{s}[0] <: c \Rightarrow (head(A_M[L+1]))[1/\mathrm{s}[0]]\wedge$ $\neg(\mathrm{s}[0] <: c) \Rightarrow (head(A_M[L+1]))[0/\mathrm{s}[0]]$
`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 3
Weakest precondition function $wp(M[L])$

– and thus policy adherence – holds for such programs and is efficiently checkable.

The level III annotation is produced with the help of the weakest precondition function $wp(M[L])$, adapted for JVM instructions from the weakest precondition function of Bannwart and Müller [8]. Table 3 contains the definition of the function for the instructions occurring in the examples. The function $shift(A)$ denotes the substitution, for all i, of $\mathrm{s}[i]$ by $\mathrm{s}[i+1]$ in assertion A, while function $unshift(A)$ denotes the inverse function. The last two rows of the table refer 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 annotations use a normalizing function *norm* on annotations, with the combined effect of conjuncting consecutive logical assertions and propagating weakest preconditions backward:

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

where γ is an annotation sequence, and $\alpha[ce/\overrightarrow{g}]$ is 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 an annotation sequence.

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

(1) *Requires*$(\Gamma^*(M))$ and *Ensures*$(\Gamma^*(M))$ is the synchronisation assertion, $\overrightarrow{gs} = \overrightarrow{ms}$.

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

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

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

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

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 $M[L]$,

$$A^{\mathrm{III}}_M[L] \;=\; norm(A^{II}_M[L])$$

(5) For all blocks of inlined code, we apply the weakest precondition function $wp(M[L])$ defined in Table 3 to propagate backwards the head assertion of the first instruction following the block (which is the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$). This 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,

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

and otherwise,

$$A^{\mathrm{III}}_M[L] \;=\; wp(M[L])$$

The level III annotated example program can be found in appendix A.

Notice that inlined code blocks are cycle-free and do not contain jumps to any instruction outside of the block (other than the one immediately following it), thus the backward *wp*-propagation in rule (5) above is well-defined. This is thanks to the design of the ConSpec language. Extending the language to allow for (arbitrary) loops in update blocks, for instance, would render this single-pass propagation insufficient for obtaining weakest preconditions of the inlined blocks.

The correctness proof below relies on the following property of inlined code:

Property 7.1 *Given a policy $\mathcal{P}$, and a program T, let T' be the program inlined for the policy by an inliner as described above. Then the following holds for each post-security relevant instruction $M[L]$ of T': Let $M[L] =$* `invokevirtual` *$(c.m)$ for some c and m, $\alpha_1, \ldots, \alpha_m$ be the guarded expressions $g_{\text{this}} : c'_i \wedge\, a_b\rho_i \rightarrow \overrightarrow{e_E}\rho_i$, $1 \leq i \leq m$, and α be $\neg(g_{\text{this}} : c'_1 \vee \ldots \vee g_{\text{this}} : c'_p) \rightarrow \overrightarrow{gs}$, induced by the policy for $M[L]$ as described in section 6.2. Furthermore, let r_{this} be the local variable used by the inliner to record the reference of the object $M[L]$ operates on. Then the weakest pre-condition of the block of code inlined immediately after the instruction $M[L]$ in T' w.r.t. the synchronisation assertion $\overrightarrow{gs} = \overrightarrow{ms}$ is the logical assertion*

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

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

The property essentially expresses that the final postcondition of the inlined blocks can be pushed backwards to produce preconditions that are discharged by the level II annotations. Note that this is the only point for which the level III correctness proof relies on the actual code produced by the inliner. We refrain from giving a fully formalized proof that the property actually holds for the inliner sketched above. The details, however, are not difficult, and in appendix A we give an example in sufficient detail that the reader should be able to convince herself that an inliner can be easily devised for which the property holds. The critical point is that the update expressions contained in the weakest precondition of an inlined block are required by the property to correspond to the update expressions of the corresponding event clause of the policy, up to renaming. An inliner can achieve this by compiling ConSpec expressions using, for each arithmetic (resp. string) operation, the corresponding instruction (resp. API method) of Java bytecode.

In the following result we use *local validity* to refer to logical validity of the verification conditions resulting from a fully annotated program (see appendix B.4 and [8] for details).

Theorem 7.2 *Suppose that I is an inliner satisfying property 7.1. Let T be* *a program, and $\mathcal{P}$ a ConSpec policy. The level III annotated inlined program I(T,$\mathcal{P}$) is locally valid.*

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

Corollary 7.3 *(Correctness of Inlining) Let $\mathcal{P}$ be a ConSpec policy and P be a program. The inlined program I(T,$\mathcal{P}$) adheres to the policy.*

As the example in appendix A indicates it is not hard to produce level III annotations that are efficiently checkable. This makes level III annotations suitable for use as proofs for on-device checking of inlining correctness in a proof-carrying code setting. The verification conditions that arise when checking the local validity of these annotations are trivial for the uninlined parts of the code as the synchronisation annotation is the invariant in this case and annotations inside inlined blocks can be discharged with syntactic manipulations and basic reasoning about the class hierarchy. The level III annotated program of appendix A illustrates this point. A proof-carrying code framework based on the results of this paper that employs our inliner has been developed. Details on the framework, including information on a bytecode precondition checker implementation to be used in this context, can be found in [12].

Another corollary of theorem 7.2 is that for any program T and policy $\mathcal{P}$ the inlined program I(T,$\mathcal{P}$) yields only method-local co-executions. This is so since programs that validate level III annotations validate also level II annotations and thus theorem 6.4 applies to inlined programs.

In contrast to the rest of the paper, the results that we have presented in this section apply to a particular inlining scheme. Similar results can be obtained for other inlining schemes as long as the completion procedure from level II to level III annotations can be adapted so that the resulting full annotations capture the updates to the embedded monitor state and the resulting validity problem is decidable. In this way, different inliners can be employed in a proof-carrying code setting.

8 Related Work

In this section, we discuss different types of inlining and whether these can be handled by our annotation scheme. We relate our approach to other works that propose methods to specify policy adherence and to other security frameworks inspired by proof-carrying code.

Monitor Inliners Monitor inlining has been employed as a security enforcement mechanism in a number of application areas. We account here the basic *types* of monitor inlining implementations (in terms of where the code is inlined) offered in the context of language-based security. We focus on method calls as security relevant actions, although it is possible to monitor many other events with existing tools such as PoET [16].

The inliners that input policies in the form of security automata can be categorized according to where they insert code to perform the inquiry of whether the security relevant action is safe to perform in the current state and the update on the security state. The inlining style of PoET and our tool creates code where security checks and updates are *scattered* in the program: the code is inserted around the security relevant method call at the caller side. An alternative is to inline the program by altering the methods of the untrusted program only through replacing potentially security relevant method invocation instructions with calls to new methods added in the inlining process. Each such new method is dedicated to a particular security relevant method and consists of code that performs the necessary security checks, the call to the method and the corresponding update to the security state. Such a *wrapping* approach is taken in Naccio, one of the first tools for monitor inlining [18]. A rather clean way of implementing monitoring is through *centralizing*, i.e. using a single dedicated component that "implements" the whole policy. The interface of the component includes an evaluator method that takes information about the security relevant method call (e.g. the name of the method and argument values) as argument, performs the necessary operations. The original program code is then only altered by the insertion of calls to this method before (or after) each security relevant method invocation. The inliner of Vanoverberghe and Piessens is an example of centralizing inliners [37]. The Polymer system also practices centralized inlining where a policy is specified as a Java class but API method bodies are altered in the course of the inlining [9].

We can handle all these different types of implementations, with the exception of those that rewrite the body of security relevant methods. It is important to note however that in some cases the procedure described in section 7 for completing level II annotations to level III (full) annotations needs to be adapted. For instance, the evaluator method of the dedicated policy component in centralized inliners is not annotated with the synchronisation assertion. Instead, it should be equipped with annotations that specify that the checks and updates are performed according to the policy and the security relevant action indicated by the arguments, which consist of the security relevant method name and the values of its arguments. As mentioned in section 6.3, those inliners that perform optimizations on inlined code based on a sequence of security relevant actions are not supported with our scheme, since these destroy the "per-action update" principle. An example is the inliner of Martin *et al.* [31].

Monitor inlining can also be implemented through *aspect-oriented programming* (AOP) [25,19]. In such an approach, the security policy is programmed as an aspect, which gets inserted into the program at the compilation stage (*aspect weaving*) resulting in scattered or wrapped code. In [13], an example can be found for policy enforcement on a Java application where the policy is programmed as an aspect in AspectJ [6], an AOP environment. In this manner the two concerns, development and contract declaration, are elegantly separated. The Monitoring-Oriented Programming (MOP) framework of Chen and Roşu [11], the Tracematches tool of Allan *et al.* [4] and the monitor inliner of Hamlen and Jones [21] employ AOP. We have not carried out this study in the context of aspect-orientation as this makes reasoning compositionally on the level of program methods more difficult.

There exist monitor inliners in literature which use other policy specification languages such as temporal logics (e.g. [14,10]), but we do not discuss these further here.

Specifying Policy Adherence In [5], Alpern and Schneider propose a method for showing that a program satisfies a temporal property by producing proof obligations on the global state space of the program, presented as a transition system. The problem we address here can be seen as its restriction to safety properties. Coming up with proof obligations using their method becomes trivial in this case and amounts to asserting that the automaton for the property is in an accepting state throughout the execution of the program. The state space of bytecode programs are prohibitively large, however, thus making the method impractical in this setting. Another result closely related to ours is the recent work on type-based monitor certification by Hamlen *et al.* [22]. Their policies are attached to security relevant classes and restrict the sequences of methods called on instances of these classes. Thus the focus is on "per-object" monitoring, as compared to the "per-session" model we consider in our annotation scheme. Their programs are bytecode programs with typing annotations and type correctly with respect to a security policy if they adhere to the policy. Thus, the authors reduce the problem of correct inlining to that of type-checking, which is an efficient, well-studied procedure. However, their results are restricted to one particular inliner, whereas we give a characterization of a whole class of compositional inliners.

Model Based Certification Many techniques inspired by the PCC approach have been developed for the certification of safe mobile code. In *model-carrying code (MCC)*, introduced by Sekar *et al.* [36], the program is shipped together with a model of its security relevant behavior instead of a proof of policy adherence. Policy adherence checking on the consumer side can be based on this model and used to certify the program for *any* policy that the program adheres to, provided the model is precise enough. In this respect, their approach is more general than ours. However, an additional check is needed on the consumer side to make sure that the model is faithful to the program. Since this may be a costly task due to the size of the model, Sekar *et al.* employs an under-approximation of the program behavior as the model and suggests the policy captured by this model to be *enforced* on the program at runtime by monitoring. Therefore, a trusted monitoring component on the consumer side is needed in MCC. If the model is precise and therefore has a small number of spurious misbehavior, the runtime aborts due to this enforcement are expected to occur infrequently. The main difficulty in applying MCC in practice is to develop a suitable model extraction scheme, which is not a trivial problem. In [36], the authors suggest a method for learning ConSpec automata-like models through executing the program on test cases. Albert *et al.* follow a similar approach but use abstract interpretation to compute the models [3]. This computation involves repeated iterations of the same procedure to reach a fixpoint. A single execution of the procedure on the model is then enough to check on the consumer side that the model is a faithful abstraction of the program. However, the class of policies handled by the approach are restricted to the class of stateless ConSpec policies, hence it is not possible to use the approach for policies restricting the order of the security relevant actions.

9 Conclusion

This paper 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 characterizes a whole class of correctly inlined programs. As the main application of these results we sketch a simple inlining algorithm and show how the two-level annotations can be completed to produce a fully annotated program which is valid. This establishes the mediation property for inlined programs. Furthermore, validity can be checked efficiently using a weakest precondition based annotation checker, thus preparing the ground for on-device checking of policy adherence in a proof-carrying code setting. This idea has been developed within the European S3MS project.

The results of the paper are used to show mediation, namely that inlined monitors guarantee the properties they are meant to enforce. Another desirable result, which we leave to future work, is to state the complementary ”conservativity” property: that the behavior of an inlined program is identical to the behavior of the original program, up until points of policy violation in which case the inlined program terminates. 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 scheme of section 6 to programs with multi-threading to develop a more comprehensive proof-carrying code setting. Current efforts focus on solving this problem.

References

[1] I. Aktug and J. Linde. An inliner tool for mobile platforms. Available at `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, *Proc. of The First Int. Workshop on Run Time Enforcement for Mobile and Distributed Systems (REM'07)*, volume 197-1 of *Electronic Notes in Theoretical Computer Science*, pages 45–58, 2007.

[3] E. Albert, G. Puebla, and M.V. Hermenegildo. Abstraction-carrying code. In *Proc. 11th Int. Conf. on Logic for Programming Artificial Intelligence and Reasoning (LPAR'04)*, volume 3452 of *Lecture Notes in Artificial Intelligence*, pages 380–397. Springer Verlag, 2004.

[4] C. Allan, P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam and J. Tibble. Adding trace matching with free variables to AspectJ. *SIGPLAN Notices*, 40(10):345–364, ACM, 2005.

[5] B. Alpern and F. B. Schneider. Verifying temporal properties without temporal logic. *ACM Transactions on Programming Languages and Systems*, 11(1):147–167, 1989.

[6] AspectJ Project Home Page. `http://eclipse.org/aspectj`.

[7] F. Y. Bannwart and P. Müller. A logic for bytecode. Technical Report 469, ETH Zurich, 2004. Available at
`http://www.sct.inf.ethz.ch/publications/`

[8] F. Y. Bannwart and P. Müller. A logic for bytecode. In *Proc. of Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE'05)*, volume 141-1 of *ENTCS*, pages 255–273, 2005.

[9] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 305–314, 2005.

[10] E. Bodden. *J-LO, a tool for runtime-checking temporal assertions.* PhD thesis, RWTH Aachen University, 2005.

[11] F. Chen and G. Roşu. Java-MOP: A Monitoring Oriented Programming Environment for Java. In *Proc. of the Eleventh International Conference on Tools and Algorithms for the construction and analysis of systems (TACAS'05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 546–550, 2005.

[12] M. Dam and A. Lundblad. A Proof Carrying Code Framework for Inlined Reference Monitors in Java Bytecode. In *Public Deliverable D4.2, S3MS*, April, 2008. Available at
`http://www.csc.kth.se/~mfd/Papers/s3ms_pcc_final.pdf`

[13] F. Diotalevi. Contract enforcement with AOP. Available at
`http://www-128.ibm.com/developerworks/library/j-ceaop/`

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

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

[16] Ú. Erlingsson and F. B. Schneider. IRM enforcement of Java stack inspection. In *Proc. of the IEEE Symposium on Security and Privacy*, page 246. IEEE Computer Society, 2000.

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

[18] D. Evans and A. Twyman. Flexible policy-directed code safety. In *Proc. of the IEEE Symposium on Security and Privacy*, pages 32–45, 1999.

[19] R. Filman, T. Elrad, S. Clarke, and M. Aksit. *Aspect-Oriented Software Development.* Addison-Wesley, 2004.

[20] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1196–1250, 1999.

[21] K.W. Hamlen and M. Jones. Aspect-oriented in-lined reference monitors In *Proc. of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20. ACM, 2008.

[22] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In *Proc. of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS'06)*, pages 7–16, June 2006.

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

[24] K. Havelund and G. Roşu. Synthesizing monitors for safety properties. In *Proc. of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '02)*, pages 342–356, London, UK, 2002. Springer Verlag-Verlag.

[25] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proc. of the European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer Verlag-Verlag, Berlin, Heidelberg, and New York, 1997.

[26] M. Kim, M. Viswanathan, S. Kannan, I. Lee and O. Sokolsky. Java-MaC: A Run-Time Assurance Approach for Java Programs. *Formal Methods System Design*, 24:129–155,2004. Kluwer Academic Publishers.

[27] 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.

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

[29] 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.

[30] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In *Proc. of the 10th European Symposium on Research in Computer Security (ESORICS'05)*, pages 355–373, Sep 2005.

[31] M. Martin, B. Livshits and M.S. Lam. Finding application errors and security flaws using PQL: a program query language. *SIGPLAN Notices*, 40(10):365–383, ACM, 2005.

[32] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential java. In S. D. Swierstra, editor, *Proc. of the 8th European Symposium on Programming (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer, March 1999.

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

[34] J.H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, 1974.

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

[36] R. Sekar, V.N. Venkatakrishnan, S. Basu, S. Bhatkar, and D. C. DuVarney. Model-carrying code: a practical approach for safe execution of untrusted

applications. In *Proc. of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pages 15–28, New York, NY, USA, 2003. ACM.

[37] D. Vanoverberghe and F. Piessens. A caller-side inline reference monitor for an object-oriented intermediate language. In *Proc. of the 10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*, Lecture Notes in Computer Science. Springer Verlag, 2008. to appear.

$A^{\mathrm{III}}[L]$	L	$M[L]$
$\{\, \mathit{INV} \,\}$	L1	`aload r0`
$\{\, \mathit{INV} \,\}$	L2	`getfield gui`
$\{\, (\mathrm{s}[0] : \mathtt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)) \wedge \mathit{INV} \wedge (\mathrm{s}[0] = \mathrm{s}[0]) \,\}$	L3	`dup`
$\{\, ((\mathrm{s}[1] : \mathtt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)) \wedge \mathit{INV} \wedge (\mathrm{s}[1] = \mathrm{s}[0])) \,\}$	L4	`astore r1`
$\{\, ((\mathrm{s}[0] : \mathtt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)) \wedge \mathit{INV} \wedge (\mathrm{s}[0] = \mathrm{r1})) \cdot$ $(g_{\mathrm{this}} := \mathrm{s}[0]) \cdot$ $((g_{\mathrm{this}} : \mathtt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)) \wedge \mathit{INV} \wedge (g_{\mathrm{this}} = \mathrm{r1})) \,\}$	L5	`invokevirtual GUI/AskConnect()Z`
$\{\, (g_{\mathrm{this}} = \mathrm{r1}) \wedge (\mathrm{r1} <: \mathtt{GUI} \Rightarrow (\Phi = (\mathtt{SecState.accessed}, \mathrm{s}[0]))) \wedge$ $(\neg(\mathrm{r1} <: \mathtt{GUI}) \Rightarrow (\Phi = (\mathtt{SecState.accessed}, \mathtt{SecState.permission}))) \cdot$ $((g_{\mathrm{a}}, g_{\mathrm{p}}) := \Phi \cdot$ $((\mathrm{r1} <: \mathtt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\mathtt{SecState.accessed}, \mathrm{s}[0]))) \wedge$ $(\neg(\mathrm{r1} <: \mathtt{GUI}) \Rightarrow \mathit{INV}))) \,\}$	L6	`istore r2`
$\{\, \neg(\mathrm{r1} <: \mathtt{GUI}) \;\Rightarrow\; \mathit{INV} \wedge$ $(\mathrm{r1} <: \mathtt{GUI}) \;\Rightarrow\; (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\mathtt{SecState.accessed}, \mathrm{r2}) \,\}$	L7	`aload r1`
$\{\, \neg(s[0] <: \mathtt{GUI}) \;\Rightarrow\; \mathit{INV} \wedge$ $(s[0] <: \mathtt{GUI}) \;\Rightarrow\; (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\mathtt{SecState.accessed}, \mathrm{r2}) \,\}$	L8	`instanceof GUI`
$\{\, (s[0] = 0) \;\Rightarrow\; \mathit{INV} \wedge$ $\neg(s[0] = 0) \;\Rightarrow\; (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\mathtt{SecState.accessed}, \mathrm{r2}) \,\}$	L9	`ifeq L12`
$\{\, (g_{\mathrm{a}}, g_{\mathrm{p}}) \;=\; (\mathtt{SecState.accessed}, \mathrm{r2}) \,\}$	L10	`iload r2`
$\{\, (g_{\mathrm{a}}, g_{\mathrm{p}}) \;=\; (\mathtt{SecState.accessed}, \mathrm{s}[0]) \,\}$	L11	`putstatic SecState/permission`
$\{\, \mathit{INV} \,\}$	L12	`iload r2`
$\{\, \mathit{INV} \,\}$	L13	`ireturn`

Fig. A.1. An application method with level III annotations for the example policy

A Level III Example

The level III annotations of the level II annotated method of figure 6 is shown in figure A.1. The synchronization assertion, denoted by INV, is $(g_{\mathrm{a}}, g_{\mathrm{p}}) = (\mathtt{SecState.accessed}, \mathtt{SecState.permission})$. The pre- and post-conditions of the method are the synchronization assertion:

$$\mathit{Requires} = \mathit{Ensures} = \mathit{INV}$$

The symbol Φ denotes the following multi-assignment expression:

$$((g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \wedge g_{\mathrm{this}} : \mathtt{GUI} \wedge \mathrm{s}[0]) \longrightarrow (g_{\mathrm{a}}, \mathit{true}) \quad \mid$$
$$((g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot) \wedge g_{\mathrm{this}} : \mathtt{GUI} \wedge \neg\mathrm{s}[0]) \longrightarrow (g_{\mathrm{a}}, \mathit{false}) \mid$$
$$(\neg(g_{\mathrm{this}} : \mathtt{GUI})) \longrightarrow (g_{\mathrm{a}}, g_{\mathrm{p}})$$

The reader may get detailed information about verification conditions in the definition of local validity (def. B.7). In this example, the inlined block in which we propagate the synchronisation annotation is $L6 - L12$. The most involved verification condition arises from the condition $\mathit{Ensures}_M \Rightarrow \mathit{head}(A^{\mathrm{III}}[L6])$:

$$\begin{aligned} \mathit{INV} \Rightarrow {} & (g_{\mathrm{this}} = \mathrm{r1}) \wedge (\mathrm{r1} <: \mathtt{GUI} \Rightarrow (\Phi = (\mathtt{SecState.accessed}, \mathrm{s}[0]))) \wedge \\ & (\neg(\mathrm{r1} <: \mathtt{GUI}) \Rightarrow (\Phi = (\mathtt{SecState.accessed}, \mathtt{SecState.permission}))) \end{aligned}$$

Using the first conjunct of $\mathit{head}(A^{\mathrm{III}}[L6])$, we can replace g_{this} by r1 in Φ. Notice that `GUI` does not have any subclasses in this example, so $\mathrm{r1} <: \mathtt{GUI} \Leftrightarrow \mathrm{r1} : \mathtt{GUI}$. Φ updates `permission` to the value of s[0] if g_{this} is of type `GUI` and does not change it otherwise.

Notice that what we do here is in some sense to unify Φ (the updates comign from the ghost annotations) to $\mathit{head}(A^{\mathrm{III}}[L6])$ (the effectual update of the inlined block), as we had also noted in section 7. In any target program, $\mathit{Ensures}_M$ is the synchronisation annotation and Φ and the head of the weakest precondition of the inlined block looks similar. What is left to the theorem prover is simply to match expressions in the annotations to those in the weakest precondition.

Below are the details of the computation of level III annotations.

Requires and *Ensures* are determined by rule 1

$\mathit{Requires} = \mathit{Ensures} = \mathit{INV}$

Annotations of $L1 - L2$ are computed using rule 2.

$$\begin{aligned} A^{\mathrm{III}}[L1] &= \mathit{norm}(\mathit{INV}) = \mathit{INV} \\ A^{\mathrm{III}}[L2] &= \mathit{norm}(\mathit{INV}) = \mathit{INV} \end{aligned}$$

Annotation of $L5$ (used for computing the annotations $L3 - L4$) is computed using rule 3.

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

Annotations of $L3 - L4$ are computed using rule 5 using wp computation.

$$\begin{aligned} A^{\mathrm{III}}[L4] &= \mathit{wp}(M[L4]) \\ &= (\mathit{shift}(\mathit{head}(A^{\mathrm{III}}_M[L5])))[\mathrm{s}[0]/\mathrm{r1}] \\ &= ((\mathrm{s}[1] : \mathtt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\perp, \perp)) \wedge \mathit{INV} \wedge (\mathrm{s}[1] = \mathrm{r1}))[\mathrm{s}[0]/\mathrm{r1}] \\ &= ((\mathrm{s}[1] : \mathtt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\perp, \perp)) \wedge \mathit{INV} \wedge (\mathrm{s}[1] = \mathrm{s}[0])) \end{aligned}$$

$$\begin{aligned}
A^{\mathrm{III}}[L3] &= \mathit{wp}(M[L3] \\
&= \mathit{unshift}((\mathit{head}(A^{\mathrm{III}}_{M}[L+1]))[\mathrm{s}[1]/\mathrm{s}[0]]) \\
&= \mathit{unshift}((\mathrm{s}[1] : \texttt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)) \wedge \mathit{INV} \wedge (\mathrm{s}[1] = \mathrm{s}[1])) \\
&= (\mathrm{s}[0] : \texttt{GUI} \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) \neq (\bot, \bot)) \wedge \mathit{INV} \wedge (\mathrm{s}[0] = \mathrm{s}[0])
\end{aligned}$$

Annotation of $L13$ (used for computing the annotations of $L6 - L12$) is computed using rule 4.

$$A^{\mathrm{III}}[L13] = \mathit{norm}(\mathit{INV}) = \mathit{INV}$$

Annotations of $L6 - L12$ are computed using rule 5 using wp computation.

$$\begin{aligned}
A^{\mathrm{III}}[L12] &= \mathit{wp}(M[12]) \\
&= \mathit{unshift}(\mathit{head}(A^{\mathrm{III}}_{M}[L13])[\mathrm{s}[0]/\mathrm{r2}]) \\
&= \mathit{unshift}(\mathit{INV}) \\
&= \mathit{INV} \\
A^{\mathrm{III}}[L11] &= \mathit{wp}(M[11]) \\
&= (\mathit{shift}(\mathit{head}(A^{\mathrm{III}}_{M}[L12])))[\mathrm{s}[0]/\texttt{SecState.permission}] \\
&= \mathit{INV}[\mathrm{s}[0]/\texttt{SecState.permission}] \\
&= (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{s}[0]) \\
A^{\mathrm{III}}[L10] &= \mathit{wp}(M[10]) \\
&= \mathit{unshift}(\mathit{head}(A^{\mathrm{III}}_{M}[L13])[\mathrm{r2}/\mathrm{s}[0]]) \\
&= \mathit{unshift}((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2})) \\
&= (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2}) \\
A^{\mathrm{III}}[L9] &= \mathit{wp}(M[9]) \\
&= (\mathrm{s}[0] = 0 \Rightarrow \mathit{shift}(\mathit{head}(A^{\mathrm{III}}_{M}[L12]))) \ \wedge (\neg(\mathrm{s}[0]) = 0 \Rightarrow \mathit{shift}(\mathit{head}(A^{\mathrm{III}}_{M}[L10]))) \\
&= (\mathrm{s}[0] = 0 \Rightarrow \mathit{INV}) \ \wedge (\neg(\mathrm{s}[0]) = 0 \Rightarrow (g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2})) \\
A^{\mathrm{III}}[L8] &= \mathit{wp}(M[8]) \\
&= (\mathrm{s}[0] <: \texttt{GUI} \Rightarrow (\mathit{head}(A^{\mathrm{III}}_{M}[9]))[1/\mathrm{s}[0]]) \wedge \\
&\quad (\neg(\mathrm{s}[0] <: \texttt{GUI}) \Rightarrow (\mathit{head}(A^{\mathrm{III}}_{M}[9]))[0/\mathrm{s}[0]]) \\
&= (\mathrm{s}[0] <: \texttt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2}))) \wedge \\
&\quad (\neg(\mathrm{s}[0] <: \texttt{GUI}) \Rightarrow (\mathit{INV})) \\
A^{\mathrm{III}}[L7] &= \mathit{wp}(M[L7]) \\
&= \mathit{unshift}((\mathit{head}(A^{\mathrm{III}}_{M}[L8]))[\mathrm{r1}/\mathrm{s}[0]]) \\
&= \mathit{unshift}((\mathrm{r1} <: \texttt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2}))) \wedge \\
&\quad (\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow (\mathit{INV}))) \\
&= (\mathrm{r1} <: \texttt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2}))) \wedge \\
&\quad (\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow \mathit{INV})
\end{aligned}$$

We denote with Φ below the right hand side of the ghost assignment of $A^{II}_{M}[L6]$.

$$\begin{aligned}
A^{\mathrm{III}}[L6] &= \mathit{norm}(g_{\mathrm{this}} = \mathrm{r1} \ \cdot A^{II}_{M}[L6] \cdot \mathit{wp}(M[L6])) \\
&= \mathit{norm}(g_{\mathrm{this}} = \mathrm{r1} \ \cdot \ A^{II}_{M}[L6] \cdot \\
&\quad (\mathit{shift}(\mathit{head}(\mathrm{r1} <: \texttt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{r2}))) \wedge \\
&\quad (\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow \mathit{INV})))[\mathrm{s}[0]/\mathrm{r2}]) \\
&= \mathit{norm}(g_{\mathrm{this}} = \mathrm{r1} \ \cdot (g_{\mathrm{a}}, g_{\mathrm{p}}) := \Phi \cdot \\
&\quad (\mathrm{r1} <: \texttt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{s}[0])) \wedge \\
&\quad (\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow \mathit{INV})) \\
&= (g_{\mathrm{this}} = \mathrm{r1}) \wedge (\mathrm{r1} <: \texttt{GUI} \Rightarrow (\Phi = (\texttt{SecState.accessed}, \mathrm{s}[0]))) \wedge \\
&\quad (\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow (\Phi = (\texttt{SecState.accessed}, \texttt{SecState.permission}))) \cdot \\
&\quad ((g_{\mathrm{a}}, g_{\mathrm{p}}) := \Phi \cdot \\
&\quad ((\mathrm{r1} <: \texttt{GUI} \Rightarrow ((g_{\mathrm{a}}, g_{\mathrm{p}}) = (\texttt{SecState.accessed}, \mathrm{s}[0]))) \wedge \\
&\quad (\neg(\mathrm{r1} <: \texttt{GUI}) \Rightarrow \mathit{INV})))
\end{aligned}$$

B Proofs

B.1 Proof of Theorem 5.3

Theorem 5.2 (Correctness of Monitoring by Co-execution) *Let T be a program, and $\mathcal{P}$ a policy. The following holds, where A is the action set of $\mathcal{A}_\mathcal{P}$:*

$$\{w \downarrow 1 \mid w \text{ is a co-execution of T and } \mathcal{A}_\mathcal{P}\} = \{E \in \Pi(\text{T}) \mid srt_A(E) \in L_{\mathcal{A}_\mathcal{P}}\}$$

Proof.

($\supseteq$) We prove that for all executions E of the program such that the security relevant trace of E is in the language of the policy automaton, there is an execution w of the program and the policy automaton, where $w \downarrow 1 = E$.

Let $q_0q_1\ldots$ be the run of the automaton for $srt_A(E)$. We (1) construct a configuration-automaton state pair sequence w for E, using the automaton run and (2) prove that w is a co-execution with $w \downarrow 1 = E$.

(1) Intuitively, we begin the construction with the initial configuration C_0 of E and the initial state q_0. We add the following configurations, paired with this state until a security relevant action (s.r.a.) is produced. Whenever an s.r.a. is produced, 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, for infinite executions the process is repeated infinitely many times. Security relevant actions are detected by using the act_A functions on consecutive configurations of the execution.

Formally, let w_n denote the sequence constructed for the (finite) prefix $C_0 \ldots C_{n-1}C_n$ of E. The sequence w_0 is defined as (C_0, q_0) if $act^\flat(C_0) = \epsilon$ and $(C_0, q_0)(C_0, q_1)$ if $act^\flat(C_n) \in A^\flat$. When constructing the sequence w_n for longer executions, we use the current state as the state component of the last pair of w_{n-1}, denoted below by q_k. The sequence w_n for $n > 0$ is defined as follows:

$$w_n = \begin{cases} w_{n-1} \cdot (C_n, q_k) & \text{if } act^\sharp(C_{n-1}, C_n)act^\flat(C_n) = \epsilon \\ w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) & \text{if } act^\sharp(C_{n-1}, C_n)act^\flat(C_n) \in A^\flat \\ w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) & \text{if } act^\sharp(C_{n-1}, C_n)act^\flat(C_n) \in A^\sharp \\ w_{n-1} \cdot (C_n, q_k) \cdot (C_n, q_{k+1}) \cdot (C_n, q_{k+2}) & \text{if } act^\sharp(C_{n-1}, C_n) \in A^\sharp, \\ & \quad act^\flat(C_n) \in A^\flat \end{cases}$$

(2) We prove that w_i is a co-execution and $w_i \downarrow 1 = C_0 \ldots C_i$ for all finite prefixes $C_0 \ldots C_i$ of E. The result then follows since this is a continuous predicate on configuration sequences with respect to the immediate prefix ordering and $\Pi(\mathrm{T})$ is prefix-closed.

In the proof, we use the fact that ConSpec automata are deterministic (by definition) and their language is prefix-closed (since each ConSpec automaton is a security automaton as defined by Schneider [35]). We can then conclude for each prefix E' of E that E' is in the language of the automaton and the run of the automaton which accepts E' is a prefix of the run accepting E.

(*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 C_0 is $act^\flat(C_0)$. Then $w_0 = (C_0, q_0)(C_0, q_1)$ by construction. This sequence is an interleaving since : $(C_0, q_0) \longrightarrow_{\mathrm{AUT}} (C_0, q_1)$. By definition then, $w_0 \downarrow 1 = C_0$ and $extract(w_0) = q_0 q_1\ act^\flat(C_0)$. Clearly $extract(w_0) \in E^\flat$. On the other hand, if $act^\flat(C_0) = \epsilon$, the security relevant trace is empty. The accepting run then consists of q_0 and the constructed sequence w_0 of (C_0, q_0). Again by definition, $w_0 \downarrow 1 = C_0$, and $extract(w_0) = \epsilon$.

(*Induction Hypothesis*) Assume that w_i is a co-execution and $w_i \downarrow 1 = C_0 \ldots C_i$ for all $i < n$.

(*Inductive Step*) Consider the sequence w_n constructed for the prefix $C_0 \ldots C_{n-1} C_n$ using the automaton run $q_0 \ldots q_m$ where m is the number of security relevant actions of $C_0 \ldots C_{n-1} C_n$. By definition, the following holds:

$$srt_A(C_0 \ldots C_{n-1} C_n) = srt_A(C_0 \ldots C_{n-1})\ act^\sharp(C_{n-1}, C_n)\ act^\flat(C_n)$$

We consider the most difficult 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) Since $q_0 \ldots q_m$ is an accepting run for this execution:

$$\begin{aligned} \delta(q_{m-2}, act^\sharp(C_{n-1}, C_n)) &= q_{m-1}\ (i) \\ \delta(q_{m-1}, act^\flat(C_n)) &= q_m \quad (ii) \end{aligned}$$

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):

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

We prove that:

- w_n *is an interleaving:* The sequence w_{n-1} 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:

$$(C_{n-1}, q_{m-2}) \longrightarrow_{\text{JVM}} (C_n, q_{m-2}) \longrightarrow_{\text{AUT}} (C_n, q_{m-1}) \longrightarrow_{\text{AUT}} (C_n, q_m)(*)$$

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

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

  By (i), $\mathit{act}^\sharp_A(C_{n-1}, C_n)q_{m-2}q_{m-1} \in E^\sharp$ and by (ii), $q_{m-1}q_m\,\mathit{act}^\flat_A(C_n) \in E^\flat$. Hence $\mathit{extract}(w_n) \in (E^\flat \cup E^\sharp)^m$.
- $w \downarrow 1 = E_n$*:* This simply follows from the induction hypothesis and applying the first projection function to w_n.

($\subseteq$) We prove that for all co-executions w of the program and the policy automaton, the projection to the first component is (i) an execution of the program and that (ii) its security relevant trace is in the language of the policy automaton.

(i) We prove this by induction on the length of w.

(*Base Case*) If $w = (C, q)$, since w is an interleaving, $C = C_0$ and $q = q_0$. Then, $w \downarrow 1 = C_0$, which is an execution.

(*Induction Hypothesis*) We assume the statement for w_n of length n.

(*Inductive Case*) We prove the statement for w_{n+1}, where $w_{n+1} = w_n \bullet [(C_{n+1}, q_{n+1})]$ for some q_{n+1}. Let the last element of w_n be (C_n, q_n) and $w_n \downarrow 1 = E_n$. By the definition of $\downarrow 1$ (page 5), $E_n = E' \bullet [C_n]$ for some sequence of configurations E'. Again by the definition of $\downarrow 1$, $w_{n+1} \downarrow 1 = E'C_n \bullet [C_{n+1}]$ if $C_n \longrightarrow_{\text{JVM}} C_{n+1}$ and $w_{n+1} \downarrow 1 = E' \bullet [C_{n+1}]$ otherwise.

(1) If the first case applies, $w_{n+1} \downarrow 1 = (w_n \downarrow 1) \bullet [C_{n+1}]$ and everything but the last element is an execution by the induction hypothesis and the last two configurations are related with the JVM transition relation. Hence this is an execution of T. We also note the observation here that $\Pi(\text{T})$ is closed under the transitive closure of the suffix relation built using the JVM transition relation.

(2) If the second case applies, by the definition of interleaving, $C_{n+1} = C_n$ and therefore $w_{n+1} \downarrow 1 = w_n \downarrow 1$. The result follows from the inductive hypothesis.

(ii) We prove this also by induction on the length of w.

(*Base Case*) If $w = (C, q)$, since w is an interleaving, $C = C_0$ and $q = q_0$. Then, $w \downarrow 1 = C_0$. According to the table of page 5, $srt_A(C_0) = a^\flat \in A^\flat$ for some pre-action $a^\flat$, or $srt_A(C_0) = \epsilon$. The statement trivially holds in the latter case, as ϵ is in the language of all security automata with at least one state. Let us assume the first case. We will prove that such a co-execution does not exist, thus reaching a contradiction and hence the statement will hold vacuously for the first case. By the definition of *extract*, $extract w = a^\flat$ if the first case applies. But by the definition of being a co-execution $a^\flat$ should be in the set $(E^\flat \cup E^\sharp)^* \cup (E^\flat \cup E^\sharp)^\omega$. This is not possible as there is no string of length 1 in this set. Hence we reach a contradiction.

(*Induction Hypothesis*) We assume the statement for all w_i of length i, where $i < n + 1$.

(*Inductive Case*) We prove the statement for w_{n+1}. We have to consider the cases of how a co-execution is produced by extending another co-execution. We prove the statement for w_{n+1}, where $w_{n+1} = w_n \bullet [(C_{n+1}, q_{n+1})]$ for some q_{n+1}. Notice that since both are co-executions, the function *extract* maps both to the same set. It can not be however that $extract w_{n+1} = extract w_n \bullet E'$ for some $E^\flat$ or $E^\sharp$, for these extensions contain always three elements, two automata states and an action, and therefore can not be extracted when the co-execution is extended with only one pair. This means that the security relevant trace of E_{n+1} is the same with that of E_n and the result holds by induction hypothesis. The other cases are proved similarly. □

B.2 Proof of Theorem 6.2

The proof of this theorem is quite complicated as it brings together many concepts of the paper such as the symbolic and the ConSpec automaton, co-execution, and operational semantics of annotations.

Let us first assume that the level I annotated program is valid. Intuitively, our goal is to show that any execution of the program can be "completed" with automaton states to form a co-execution of the program with the policy automaton (the ConSpec automaton for the policy). This means for each configuration in the execution such an automaton state should be found that the pair sequence that is formed is a co-execution. We use the value of the ghost state as the automaton state. Remember the conditions for a configuration-automaton state pair sequence to be a co-execution. The first condition is that the automaton component of the first pair is the initial automaton state. When execution begins, the ghost state is the initial state, thus satisfies this condition. The second is that for a pre-action, the automaton state is updated sometime before the action takes place, but after the previous action in the series. The ghost state is updated immediately before the execution of a pre-action, since a ghost assignment is placed before each instruction which may yield a preaction when executed. (Similarly for post-actions but with an update to automaton state/ghost state immediately after.) We call a co-execution where the monitor updates are done immediately before (or after) a s.r.a. a *closest updating co-execution*. For the ghost state to be a monitor for the program, it should also be updated to the correct automaton state. We prove this using the way a ConSpec automaton is induced by a symbolic automaton and the way the (same) symbolic automaton induces the level I annotations. There is one catch: if at some configuration of the execution, the ghost state is undefined and a security relevant action is performed, the ghost state remains undefined; but such a sequence can never be co-execution. The reason is that the "undefined" state of the automaton does not have any outgoing transitions, thus the automaton sequence extracted from such a pair sequence would not be a run of the automaton. The validity assumption is used to rule out this possibility. So we also show in the course of the proof that if the annotated program is valid, then a security relevant action is not executed when the ghost state is undefined.

We first present some new definitions that will be used in the course of the proof like extended execution and closest updating co-execution.

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

The following property follows from the definition of level I annotations.

Property B.1 *Let C be an unexceptional configuration of program T. If $A^{\flat}_{M}[pc(C)] = \epsilon$ in $T_{\mathcal{P}}$, then $act^{\flat}_{A}(C) = \epsilon$. Let C' be configuration following C in an execution of program T. If C' is unexceptional and $A^{\sharp}_{M}[pc(C)] = \epsilon$ in $T_{\mathcal{P}}$, then $act^{\sharp}_{A}(C, C') = \epsilon$.*

Definition B.2 (Extended Execution) *Given an annotated program T_A, a sequence of extended configurations $(\psi_0, C_0, \sigma_0, \Sigma_0)(\psi_1, C_1, \sigma_1, \Sigma_1)\ldots$ is termed an* extended execution *of T_A, if:*

- $(\psi_0, C_0, \sigma_0, \Sigma_0)$ *is the initial extended configuration as defined on page 6.1, and*
- $\forall i.\ \Gamma^* \vdash (\psi_i, C_i, \sigma_i, \Sigma_i) \rightarrow (\psi_{i+1}, C_{i+1}, \sigma_{i+1}, \Sigma_{i+1})$

That is, any Γ^*-derivation that definition 6.1 refers to is an extended execution.

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. An extended execution is called *complete* if it executes the precondition (if any) of the instruction at the program counter of its last configuration to completion.

Definition B.3 *(Complete Extended Execution) Given a finite execution* $E = C_0 \ldots C_{n-1} C_n$ *of program* T*, the extended execution* $X_E = (\psi_0, C'_0, \sigma_0, \Sigma_0) \ldots (\psi_{m-1}, C'_{m-1}, \sigma_{m-1}, \Sigma_{m-1})(\psi_m, C'_m, \sigma_m, \Sigma_{m-1})$ *of the annotated program* $T_{\mathcal{P}}$ *is the* complete extended execution *of* E *if* $X_E \downarrow 2 = E$ *and* $\psi_m = \epsilon$*.*

Given a finite execution $E_n = C_0 \ldots C_n$ of program T, notice that the following hold for the execution $E_{n+1} = C_0 \ldots C_n C_{n+1}$, where $(\epsilon, C_n, \sigma, \Sigma)$ is the last element of X_{E_n}:

(1) If C_n is not an application method call or a return, and C_{n+1} is not exceptional, i.e. rule (5) of table 2 applies:

$$X_{E_{n+1}} = X_{E_n} \bullet (A_{M(C_{n+1})}[pc(C_{n+1})], C_{n+1}, \sigma, \Sigma) \ldots (\epsilon, C_{n+1}, \sigma', \Sigma) \quad \text{(B.1)}$$

for some σ'.

(2) If C_n is a return or is exceptional with an exception that can not be handled in the current method, i.e. rule (6) applies:

$$X_{E_{n+1}} = X_{E_n} \bullet \\ (\mathit{Ensures}(\Gamma^*(M(C_n))), C_{n+1}, \sigma_g \uplus \sigma'_l, \Sigma') \ldots \ (\epsilon, C_{n+1}, \sigma', \Sigma') \quad \text{(B.2)}$$

where $\sigma = \sigma_g \uplus \sigma_l$ for some σ_g and σ_l and $\Sigma = \sigma'_l \Sigma'$.

(3) If C_n is an application method call and C_{n+1} is not exceptional, i.e. rule (7) applies:

$$X_{E_{n+1}} = X_{E_n} \bullet \\ \mathit{Requires}(\Gamma^*(M(C_{n+1}))) \cdot A_{M(C_{n+1})}[1], C_{n+1}, \sigma_g \uplus \sigma_l^0, \sigma_l \cdot \Sigma) \quad \text{(B.3)} \\ \ldots \ (\epsilon, C_{n+1}, \sigma', \sigma_l \cdot \Sigma)$$

where $\sigma = \sigma_g \uplus \sigma_l$ for some σ_g and σ_l.

(4) Finally, if C_n was not exceptional but C_n is exceptional, i.e. rule (8) applies:

$$X_{E_{n+1}} = X_{E_n} \ \bullet \ (\epsilon, C_{n+1}, \sigma, \Sigma) \quad \text{(B.4)}$$

Constructing the Co-execution A sequence of configuration-automaton state pairs are constructed 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 If the current extended configuration is the last in the sequence, then the sequence is not extended further. 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 obtained by sampling the ghost state some time during the extended execution that ends with C'. The reason is that there is no annotation associated with exceptional configurations and the ghost update is done in this case at the precondition of the first instruction of the handler. This precondition is executed *after* at the extended execution of the configuration following C'. In order to sample the ghost value in such a situation, we consider a maximal execution of which the finite execution is a prefix of. This way we get to "peek" to the new value of the ghost state.

Let $E = C_0 \dots C_{j-1}C_j$ be a finite execution and let $X_E = (\psi_0, C'_0, \sigma_0, \Sigma_0) \dots (\psi_k, C'_k, \sigma_k, \Sigma_k)$ be its corresponding extended execution. Notice that the first extended configuration correspond to the execution of $\mathit{Requires}^I_{\langle\mathtt{main}\rangle}$. If the last two configurations (C_{j-1}, C_j) of E do not induce an exceptional action, the sequence of configuration-automaton state pairs corresponding to this extended execution is defined as

$$w(X_E) = (C_0, q_0)\ \mathit{subw}((\psi_1, C'_1, \sigma_1, \Sigma_1) \dots (\psi_k, C'_k, \sigma_k, \Sigma_k))$$

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 X' of $E' = C_0 \dots C_jC_{j+1}$. The value of the ghost state at the last element of X' is taken in this case.

- $\mathit{subw}((\psi_1, C_1, \sigma_1, \Sigma_1)\cdot(\psi_2, C_2, \sigma_2, \Sigma_2)\cdot X') = (C_2, \sigma_2(\overrightarrow{gs}))\cdot \mathit{subw}((\psi_2, C_2, \sigma_2, \Sigma_2)\cdot X')$ if $C_1 \longrightarrow_{\mathrm{JVM}} C_2$
- $\mathit{subw}((\psi_1, C_1, \sigma_1, \Sigma_1)\cdot(\psi_2, C_2, \sigma_2, \Sigma_2)\cdot X') = (C_2, \sigma_2(\overrightarrow{gs}))\cdot \mathit{subw}((\psi_2, C_2, \sigma_2, \Sigma_2)\cdot X')$ if $\psi_1 = (\overrightarrow{gs} := \alpha_1| \dots |\alpha_k) \cdot \psi_2$ for some $k \neq 1$
- $\mathit{subw}((\psi_1, C_1, \sigma_1, \Sigma_1) \cdot (\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X') = \mathit{subw}((\psi_2, C_2, \sigma_2, \Sigma_2) \cdot X')$ otherwise.
- $\mathit{subw}(\psi, C, \sigma, \Sigma) = \epsilon$

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 for a level I annotated program, 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.4 (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)*:*

- $act^{\flat}_{A}(C_1) \in A^{\flat} \wedge (C_2, q_2) \longrightarrow_{\mathrm{JVM}} (C_3, q_3) \Rightarrow (C_1, q_1) \longrightarrow_{\mathrm{AUT}} (C_2, q_2)$
- $act^{\sharp}_{A}(C_1, C_2) \in A^{\sharp} \;\wedge\; \neg Exc(C_2) \Rightarrow (C_2, q_2) \longrightarrow_{\mathrm{AUT}} (C_3, q_3)$
- $act^{\sharp}_{A}(C_1, C_2) \in A^{\sharp} \;\wedge\; Handled(C_2) \Rightarrow (C_2, q_2) \longrightarrow_{\mathrm{JVM}} (C_3, q_3) \;\wedge$ $(C_3, q_3) \longrightarrow_{\mathrm{AUT}} (C_4, q_4)$

The Proof 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.5 *$T_{\mathcal{P}}$ 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 length of E.

(*Base Case*) When the number of configurations in E is 1, the complete extended execution is the execution of $Requires^{I}_{\langle \mathtt{main} \rangle}$ and the precondition of the first instruction of $\langle$`main`$\rangle$. The more involved case arises if this precondition is not empty. Otherwise, $w(C_0) = (C_0, q_0)$ by construction. Similarly, if $A^{I}_{\langle \mathtt{main} \rangle}[1]$ includes a ghost assignment then the constructed sequence depends on which condition the assignment was done for. Let us consider the case when $k \neq 1$.

In this case, the constructed sequence is $w(C_0) = (C_0, q_0)(C_0, \sigma_0(\overrightarrow{gs}))$, where σ_0 is the mapping at the end of the extended execution. By definition, this is a co-execution if $act^{\flat}_{\langle\mathtt{main}\rangle}(C_0) \in A^{\flat}$ and $\delta^{\flat}(q_0, act^{\flat}_{\langle\mathtt{main}\rangle}(C_0)) = \sigma_0(\overrightarrow{gs})$. This can be proven using the definition of before annotations and the way ConSpec automaton is extracted from symbolic automaton.

(*Induction Hypothesis*) For all executions $E_i = C_0 \ldots C_{i-2}C_{i-1}$ of length i such that $i \leq n$ and $act^{\sharp}_{A}(C_{i-2}, C_{i-1})$ is not an exceptional post action, we assume that $w(X_i)$ is a co-execution where $w(X_i) \downarrow 1 = E_i$ if and only if all boolean formulae asserted in the complete extended execution X_i holds except possibly the assertions *Defined*$^{\sharp}$ and *Defined*^{e} asserted in the course of the execution of the precondition of $pc(C_{i-1})$.

Notice that this induction hypothesis 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 *Defined*$^{\sharp}$ or *Defined*^{e} is asserted in the course of the execution of the precondition of $pc(C_{i-1})$. If the maximal execution is one which returns from the $\langle\mathtt{main}\rangle$, then $pc(C_{i-1})$ 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.

(*Inductive Step*) Consider the execution $E_{n+1} = C_0 \ldots C_{n-1}C_n$ 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 $E_n = C_0 \ldots C_{n-1}$. 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 $(\epsilon, C_{n-1}, \sigma, \Sigma)$ for some σ and 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 $(A_M[L], C_n, \sigma, \Sigma)$, and its last element is $(\epsilon, C_n, \sigma', \Sigma)$ 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:

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

By the definition of level I annotations,

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

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:

· $A^\sharp_M[L-1][1] = \epsilon$, $A^\flat_M[L] = \epsilon$

In this case there are no ghost assignments in $A_M[L]$ and so $subw(X) = \epsilon$ by definition. Then, $w(X_{E_{n+1}}) = w(X_{E_n})(C_n, \sigma(\overrightarrow{gs}))$.

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:

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

By property B.1, $\mathit{act}^\flat_A(C_n) = \epsilon$ and $\mathit{act}^\sharp_A(C_{n-1}, C_n) = \epsilon$. By the induction hypothesis, $w(X_{E_{n+1}})$ is a co-execution.

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 closest updating co-execution, and $\mathit{act}^\flat_A(C_n) = \epsilon$, $\mathit{act}^\sharp_A(C_{n-1}, C_n) = \epsilon$. (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 $\mathit{Defined}^\sharp$ and $\mathit{Defined}^e$ executed for $pc(C_{n-1})$) hold. Since $A^\flat_M[L] = \epsilon$, there is no $\mathit{Defined}^\flat$ that is asserted in the precondition of $pc(C_{n-1})$, hence the only assertions that should be shown to hold are $\mathit{Defined}^\sharp$ and $\mathit{Defined}^e$ 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 $\mathit{Defined}^\sharp$ and $\mathit{Defined}^e$ 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 achieved 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.

· $A^\sharp_M[L-1][1] \neq \epsilon$, $A^\flat_M[L] = \epsilon$

Then the suffix X is as follows:

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

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:

(1) $k = 1$:

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

If only the second holds however $act^{\sharp}_{A}(C_{n-1}, C_n) \in A^{\sharp}$ and $\mathit{extract}(w(X_{E_{n+1}}))$ 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 $\sigma(\overrightarrow{gs}) \neq \perp$. 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 $RS((c, m), A^{\sharp}_{s} \setminus A^{e}_{s}$, then $\sigma(\overrightarrow{gs}) \neq \perp$. Hence it can not be the case only the second conjunct holds.

(2) $k > 1$:

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

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

In order to show that $w(X_{E_{n+1}})$ is an interleaving, we should prove that there exists 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 $\delta^{\sharp}(q, \mathit{act}^{\sharp}_{A}(C_{n-1}, C_n)) = q'$. (This is the only possibility since by property B.1, $\mathit{act}^{\flat}_{A}(C_n) = \epsilon$)

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: $(\overrightarrow{gs} \neq \overrightarrow{\perp}) \wedge g_{\text{this}} : c'_i \wedge a \ \rightarrow \ \overrightarrow{e}$, where $c'_i \in RS((c, m), A^{\sharp}_{s} \setminus A^{e}_{s})$. Note that α_1 holds at (2). This implies that $\overrightarrow{gs} \neq \overrightarrow{\perp}$ 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 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* $(\mathit{type}(d, h), m)$ 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 it is not 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 $c' \in RS((c, m), A^{\sharp}_{s} \setminus A^{e}_{s})$, 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 $(c.m) : (\gamma \rightarrow \tau)$. 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 $\rho = [v/x, g_0/x_0, \ldots g_{k-1}/x_{n-1}, g_{this}/\mathtt{this}]$ by this 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),

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

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

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

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

(3) $k = 0$:

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 property B.1, $act_A^\flat(C_n) = \epsilon$)

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 $RS((c, m), A^\sharp_s \setminus A^e_s$. 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 $\delta^\sharp(q, act^\sharp_A(C_{n-1}, C_n)) = q'$, where $q' = \overrightarrow{\perp}$ and we are done.

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

· The cases where $A^\sharp_M[L-1][1] = \epsilon$, $A^\flat_M[L] \neq \epsilon$ and where $A^\sharp_M[L-1][1] \neq \epsilon$, $A^\flat_M[L] \neq \epsilon$ 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 is similar to the one above when C_{n-1} is not an application method call.
- 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 $act^\sharp_A(C_{n-2}, C_{n-1}) \neq \epsilon$. 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 $act^\sharp_A(C_{n-2}, C_{n-1}) \neq \epsilon$. 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.

□

Proposition B.6 *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 (Level I Characterization) *The level I annotated program T for policy $\mathcal{P}$ is valid, if and only if, T adheres to $\mathcal{P}$.*

Proof.

The result follows in one direction from theorem 5.3, proposition B.6 and lemma B.5; the other direction follows from lemma B.5 and theorem 5.3. □

B.3 Proof of Theorem 6.4

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

Proof. (Sketch) 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 postaction followed, some time later, by a preaction, 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, exceptional configurations with an unhandled exception, 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 6.2 that the ghost state and machine configurations constitute a co-execution if and only if level I annotated program is valid. If the level II annotated program 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 $extract_{\mathrm{II}}$ applied to this sequence falls in the set stated in definition 6.3.

($\Leftarrow$) In this direction, we show the result by taking any execution E of a valid level II annotated program. Since level II annotations include level I annotations, by theorem B.5 one can construct a co-execution of this program and the automaton $\mathcal{A}_{\mathcal{P}}$, in the sense of section 5, using the ghost state. By the placement of the synchronisation annotations, the value of the embedded state can be inferred at sampling points, using the value of the ghost state. Then, it is left to show that for the embedded state to be a monitor for the policy, it is sufficient that the embedded state is in synch with the ghost state at the points where level II annotations are asserted. For instance, the ghost state gets updated for a preaction, immediately before the action and by the validity of the level II annotations, at this point the embedded state is equal to the ghost state, hence if the embedded state has been a monitor until this point, this property will be preserved for the next action.

The proof is by induction on the length of the execution:

(*Base Case:*) The sequence produced for an execution C_0 depends on whether it is a sampling point or not. C_0 is not preceded by any configuration, and is not exceptional. Therefore, if $\mathrm{pc}(C_0)$ is an `invokevirtual` or a `return` C_0 is a sampling point and the sequence is $w(C_0, \overrightarrow{ms}) = (C_0, q)$ where $q = C_0(\overrightarrow{ms})$. Otherwise, $w(C_0, \overrightarrow{ms}) = (C_0, q_0)$. Let us carry out the case when this is a `return`. By the synchronisation annotation asserted by the *Ensures* clause of $\langle \mathtt{main} \rangle$, $\overrightarrow{ms} = \overrightarrow{gs}$ at C_0. Since there are no ghost assignments associated with a `return` instruction, the ghost state is still the initial state of the automaton at C_0. The result of applying the extract function is then $\mathit{extract}_{\mathrm{II}}(w) = q_0 \mathtt{brk} q_0$.

(*Induction Hypothesis:*) For all executions E_k of length $k \leq n$, if the level II annotation of T with embedded state $\overrightarrow{ms}$ is valid, then $w(E_k, \overrightarrow{ms})$ is a method-local co-execution.

(*Inductive Step:*) Assume that the level II annotation of T with embedded state $\overrightarrow{ms}$ is valid and consider the execution $E_{n+1} = C_0 \ldots C_n$. The sequence $w(C_0 \ldots C_n, \overrightarrow{ms})$ is built by extending $w(E_n, \overrightarrow{ms})$ with the pair (C_n, q). Notice that since $w(E_n, \overrightarrow{ms})$ is a method-local co-execution, the result of applying the $\mathit{extract}_{\mathrm{II}}$ function returns a sequence ending with some state q, except the case where C_{n-1} is an API method call that induces a preaction. If C_n is a sampling point, the state component q of this pair is $C_n(\overrightarrow{ms})$; the state component is the same as the state component of the last pair of $w(C_0 \ldots C_{n-1}, \overrightarrow{ms})$, if C_n is not a sampling point. By lemma B.5, we know that $w(X_{E_n})$ is a closest updating co-execution and $w(X_{E_n}) \downarrow 1 = E_n$. Let the last element of X_{E_n} be $\epsilon, C_n, \sigma, \Sigma$ for some σ and Σ.

We consider the different cases for the pair C_{n-1}, C_n. Notice that for all cases except the last, C_n is a sampling point.

- C_n is an API method call and C_{n-1} is not a method call: By the definition of $extract_{\mathrm{II}}$,

$$extract_{\mathrm{II}}(w(E_n, \overrightarrow{ms})) = extract_{\mathrm{II}}(w(C_0 \ldots C_{n-1}, \overrightarrow{ms}))\, C_n(\overrightarrow{ms})\, act^{\flat}(C_n)$$

By the validity assumption and the way level II annotations are inserted, $C_n(\overrightarrow{ms}) = \sigma(\overrightarrow{gs})$.

- $Unhandled(C_n)$: `invokevirtual` instruction or if $n = 0$.
- C_n is not of the above: In this case, C_n is not a sampling point. Furthermore, by the definition of $extract_{\mathrm{II}}$: hence $extract_{\mathrm{II}}(w(E_n, \overrightarrow{ms})) = extract_{\mathrm{II}}(w(E_{n-1}, \overrightarrow{ms}))$ and the claim holds by the induction hypothesis.
- The other cases are similar.

($\Rightarrow$) The argument goes as follows: we take an arbitrary method-local co-execution and show that any execution that yields such a co-execution validates its assertions. For instance, as a base case, take q`brk`q. By definition of $extract_{\mathrm{II}}$, the sequence that yields this co-execution includes one and only one configuration which is an application method call. There are no other method calls (if this was the case the resulting co-execution would contain more automaton states.). Since the sampling begins with the initial automaton state q_0, q should be q_0. There are no s.r.a's and the ghost state is also the initial state throughout the execution, thus validating both the assertions on the ghost state being defined (if any) and the synchronisation assertion immediately before the method call. (Notice that there are no other assertions as there are no other states extracted and hence has no other sampling points.)□

B.4 Proof of Theorem 7.2

Before we proceed with the proof of the theorem, we clarify our notion of local validity.

For a fully annotated program, checking validity can be reduced to the simpler problem of checking local validity by referring to the axiomatic semantics of instructions. Local validity can be checked by generating verification conditions for each instruction and for method entry and exit points, by using the corresponding pre- and post-conditions and checking that these verification conditions hold. As mentioned before, our logic is an adaptation of the logic of Bannwart and Müller [7,8], which in turn is a specialization of the logic of Poetzsch-Heffter and Müller [32] to bytecode. In this section, we first introduce this logic briefly, noting the differences it has with the one presented in section 6.1. Then we define a notion of local validity, the correctness of which is based on the results of Bannwart and Müller. In this way, the definition of local validity is clear and can be used in the proof of the theorem.

Bannwart-Müller Logic The logic is for a bytecode language with object-oriented features such as classes and objects, inheritance, fields, and virtual method resolution, as well as unstructured control flow with conditional and unconditional jumps, which makes it suitable for our purposes. Instead of using triples for instruction specifications as in classic Hoare logic, programs are annotated by associating a single assertion with each instruction, interpreted as its precondition. For an instruction I, its precondition has to be established by all predecessors of I, which usually includes the instruction that precedes I in the program text as well as all instructions that jump to I. We also follow this approach for specifying instructions.

Bannwart and Müller allow different specifications to be attached to the method implementation and the method body. What is more, it is possible to pose a common pre- and post-condition to all methods that can be invoked using the same method invocation instruction. Properties of methods are expressed by Hoare triples of the form $\{P\}\mathit{comp}\{Q\}$, where P and Q are first-order formulae and *comp* is a method body, a method implementation, or a "virtual" method, explained in more detail below. The triple $\{P\}\mathit{comp}\{Q\}$ expresses the following refined partial correctness property: if the execution of *comp* starts in a state satisfying P, then (1) this computation terminates in a state in which Q holds, or (2) *comp* aborts due to errors beyond the semantics of the programming language (for instance, internal JVM errors), or (3) *comp* does not terminate.

In both our logic and this logic, individual instruction specifications can be combined at the level of method bodies. This is due to the following guarantees on the structure of Java bytecode: the instruction sequence constituting a method body is always entered at the first instruction and left after the last instruction[6] and all jumps are local within a method body. The body of the method $c.m$ is denoted with $\mathit{body}(c.m)$. A *method body specification* is then written as $\{P\}\mathit{body}(c.m)\{Q\}$, where the precondition P is the precondition of the first instruction and the postcondition Q is the precondition of the return instruction. *Method implementation specifications* play a similar role to that of *Requires* and *Ensures* in our logic, except that the postcondition of a method implementation need not hold at an exceptional exit from the method. These are denoted with $\{P\}\mathit{imp}(c.m)\{Q\}$ for method $c.m$.

The proofs are constructed in this logic using rules that combine specifications on a lower level to infer specifications on a higher level and language independent rules. Here we only present a few rules from the version of this logic where exceptions not considered (i.e. [8]) in order to give an intuition to the user. The details of the logic, including reasoning on programs that

[6] Note that methods in which return instructions occur earlier can be rewritten so to redirect all returns to the last return instruction in order to satisfy this condition.

contain exception handling can be found in [7]. A sequent in this proof system has the following form:

$$\Phi \vdash \{P\}comp\{Q\}$$

where Φ is a set of method specifications needed for dealing with recursive methods. The rule about method body specifications, which combines the assertions occurring in the method body as explained above is as follows:

$$\frac{\forall i \in \{|body(c.m)|\}.\ (\Phi \vdash \{A[i]\}Li : I_i)}{\Phi \vdash \{A[1]\}body(c.m)\{A[|body(c.m)|]\}}$$

To infer the method implementation specifications, the following rule is used:

$$\frac{\Phi, \{P\}imp(c.m)\{Q\} \vdash \{P \wedge \mathrm{r0} \neq \mathtt{null}\}body(c.m)\{Q\}}{\Phi \vdash \{P\}imp(c.m)\{Q\}}$$

The assertion $\mathrm{r0} \neq \mathtt{null}$ guarantees that, at the point where the method body starts executing, the address to the object the method is called on is stored in the local variable r0 and that it is not `null`. The rule also shows how assertions such as $\{P\}imp(c.m)\{Q\}$ are added to the set of assumptions.

Specifications on virtual methods are meant to capture method specifications imposed by the specification of an `invokevirtual` instruction on any method that can be called as a result of the execution of this instruction. In order to prove the precondition P for the instruction `invokevirtual` $c.m$, it has to be proven that (i) $\{P'\}virtual(c.m)\{Q'\}$, i.e. the methods that can be called by this instruction satisfy their method specification, (ii) that P implies the precondition P' of the virtual method specification, with actual arguments substituted for the formal parameters, and that (iii) the postcondition Q' of the method specification implies the precondition of the instruction following `invokevirtual`. In turn, to be able to prove $\{P\}virtual(c.m)\{Q\}$, it has to be proven that the specification holds for each method that can be called. This is done through proving $\{P \wedge \mathrm{r0} : c'\}imp(c'.m)\{Q\}$ for the implementation of each method $c'.m$ where $c' <: c$ and where the assertion $\mathrm{r0} : c'$ guarantees that the method is called on an object of type c'.

Our method specifications and the related rules are simplifications of this logic. While Bannwart-Müller logic is both sound and complete [7] with respect to the language presented above, we only aim at soundness for program specifications of a particular shape, specifically for specifications described in detail in sections 6.2, 6.3 and 7. Here we only note that when the program is fully annotated in our scheme, all methods (with the exception of $\langle$`main`$\rangle$ which is not to be called from inside the program) have the same specification. Furthermore, both method specifications (the pre- and post-condition of methods) and the pre- and post-condition of method invocation instructions mention the same (invariant) assertion. Finally, this invariant does not mention formal arguments. Our *Requires* and *Ensures* clauses correspond to method implementation pre- and post-conditions of Bannwart-Müller logic, respectively. Notice that method body specifications do not correspond to our *Requires* and *Ensures* clauses, as a method body pre-condition is asserted each time there is a jump to the first instruction from within the body, since it is identical to the precondition of the first instruction. Therefore the rules of Bannwart-Müller logic become superfluous. Our only extension to this logic is the use of ghost variables and ghost assignments. Ghost variables can be seen as regular variables which are not affected by program code. We treat ghost assignments the same way as program instructions as these are not boolean expressions.

Local Validity In section 7, we described how a program inlined for a policy with a simple inliner can be fully annotated so that the validity of the annotations implies adherence of the program to the policy. Consequently, the problem of policy adherence for an inlined program is reduced to checking local validity. In this section, we introduce a suitable notion of fully annotated programs and conditions for a fully annotated program to be locally valid.

A *fully annotated program* is, then, a program where a sequence of annotations γ are associated with each instruction and with the *Requires* clause, and where only a single assertion is associated with the *Ensures* clause; each instruction specification and *Requires* clause consists of a single boolean expression or an alternating sequence of ghost assignments and boolean expressions α, with the first and last elements being a boolean expression. This definition guarantees that each ghost assignment is preceded and succeeded by a boolean expression. The expression before a ghost assignment can then be used as its specification, like it is done for program instructions.

In the definition of local validity, we use the function $wp(M[L])$ for computing the local weakest precondition of an instruction presented for a subset of JVM instructions in table 3. The notion of local validity is defined as expected, namely that (i) the method precondition implies the annotation associated with the first instruction, (ii) the precondition of the return instruction implies the method post-condition, (iii) the pre-condition of an instruction implies the weakest precondition of the instruction provided it is not a method invocation, (iv) the last assertion before a ghost assignment implies the first assertion after the ghost assignment where the ghost values are replaced with the conditional expression of the assignment, (v) the pre-condition of an instruction implies the pre-condition of any handler that covers the instruction and it implies the post-condition of the method if it can raise an exception not covered by any handler of the method, (vi) for all method invocation instructions L, there exists an assertion α such that the pre-condition of the instruction implies the conjunction of the pre-condition of any method, which can be called by the instruction and α, while the conjunction of α and the postcondition of any method, which can be called by the instruction, imply the pre-condition of $L + 1$, furthermore if a method that can be called by this instruction raises exceptions, then the post-condition of the called method implies the pre-condition of the corresponding handler if any, or implies the postcondition of the caller method, (vii) finally, the initializations to the static variables done by the initial static heap is sufficient to make the pre-condition of $\langle \texttt{main} \rangle$ valid.

In the definition below, the function *head* returns the first element of an annotation sequence and *last*, the last element. $\mathit{Statics}_{\mathrm{T}}$ denotes the set of static variables $c.f$ of T and for all $c.f \in \mathit{Statics}_{\mathrm{T}}$, $v_{c.f}$ is equal to $sh_0^{\mathrm{T}}(c.f)$. We let $\models$ denote standard first-order logic validity.

Definition B.7 (Local Validity) *A fully annotated program T is* locally valid *if for every virtual method $M = (P, H, A, \mathit{Requires}, \mathit{Ensures})$ the following holds:*

(i) $\models \mathit{last}(\mathit{Requires}) \Rightarrow \mathit{head}(A[1])$,

(ii) $\models \mathit{last}(A[|P|]) \Rightarrow \mathit{Ensures}$,

(iii) for all $L \in \mathit{Dom}(P)$ where $M[L]$ is not a method invocation instruction: $\models \mathit{last}(A[L]) \Rightarrow \mathit{wp}(M[L])$,

(iv) whenever $\gamma \cdot \alpha \cdot (\overrightarrow{g} := ce) \cdot \alpha' \cdot \gamma'$ is an instruction specification,

$$\models \alpha \Rightarrow \alpha'[ce/\overrightarrow{g}]$$

(v) for all $L \in \mathit{Dom}(P)$, if $M[L]$ can raise an exception with type c, one of the following holds:

a) There exists a handler (L_1, L_2, L', c') that handles this exception, and

$$\models \mathit{last}(A[L]) \Rightarrow \mathit{head}(A[L'])$$

b) There does not exist a handler for label L and exception c, and

$$\models \mathit{last}(A[L]) \Rightarrow \mathit{Ensures}$$

*(vi) for a label $L \in \mathit{Dom}(P)$ where $M[L] =$ **`invokevirtual`** $c.m$, and for all methods $c'.m$ that can be invoked as a result of the execution of this instruction and virtual method resolution, let $c'.m = (P', H', A', \mathit{Requires}', \mathit{Ensures}')$ and $c'.m$ be of arity n. Then there exists the assertion α which mentions only local and (local) ghost variables, such that:*

- $\cdot$ $\models \mathit{last}(A[L]) \Rightarrow (\mathit{head}(\mathit{Requires}') \wedge \alpha)$,
- $\cdot$ $\models \alpha \wedge \mathit{Ensures}' \Rightarrow \mathit{head}(A[L + 1])$,
- $\cdot$ *If $c'.m$ raises an exception then either there exists a handler with destination L' and $\models \mathit{Ensures}' \Rightarrow \mathit{head}(A[L'])$ or there is no such handler and $\models \mathit{Ensures}' \Rightarrow \mathit{Ensures}$.*

(vii)

$$\models \bigwedge_{c.f \in Statics_T} c.f = v_{c.f} \quad \Rightarrow \; head(Requires_{\langle \mathtt{main} \rangle})$$

The assertion α mentioned in item (vi) is used for assertions that are preserved by the method call, that is assertions on local program and ghost variables. Note that this notion of local validity includes recursive methods, since we do not require that the called method be a different method from the caller method. This, in effect, means that the method specification can be assumed at the point of the recursive call, which is in line with [8].

Showing that a locally valid program is valid also in the sense of definition 6.1 can be done based on the soundness result[7] of [8], detailed and extended to exceptions in [7]. Such a proof consists of extending Bannwart-Müller logic with a rule for ghost assignments, extending the soundness proof they present with this rule and showing that a proof tree can be constructed to infer $\{Requires_M\}imp(M)\{Ensures_M\}$ for each method M of the program in this logic. The proof construction is straightforward as our local validity definition is simply an application of the proof rules in a restricted setting. A rule for handling ghost variables is not hard to develop either, guided by the fact that ghost assignments are very much like ordinary assignment statements that do not alter the machine configuration, and would resemble rule (iv) above.

The Proof

Theorem 7.2 *Suppose that I is an inliner satisfying property 7.1. Let T be a program, and $\mathcal{P}$ a ConSpec policy. The level III annotated inlined program I(T,$\mathcal{P}$) is locally valid.*

Proof. (Sketch) We show that the verification conditions resulting from the level III annotation of I(T,$\mathcal{P}$) 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 alternating with ghost variable

[7] Their soundness result states that whenever $\vdash \{P\}body(c.m)\{Q\}$, it is the case that for all initial configurations C that $c.m$ can start running with (i.e. configurations with the right number of values for arguments are on the stack etc.) and all configurations C', if C' is a terminating configuration with the current method $c.m$, C is related to C' with the reflexive, transitive closure of the transition relation of the operational semantics and P holds at C, then Q holds at C'. In order to prove this result, they prove a similar result for single steps of the machine.

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(T,$\mathcal{P}$) 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 $\mathit{last}(A_M^{III}[L]) \Rightarrow \mathit{Ensures}(\Gamma^*(M))$

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.

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 $\mathit{last}(A^{III}_M[L])$ and $\mathit{wp}(M[L])$, and the only other conjuncts in the latter must be either of the shape $\mathit{Defined}^\sharp$ or s[i] = s[i]; (c) if $M[L]$ is an inlined instruction, $\mathit{last}(A^{III}_M[L])$ must contain a conjunct $\mathit{wp}(M[L])$ by construction. In all three cases, the verification condition is valid, validity being efficiently checkable; the only interesting case here is presented by $\mathit{Defined}^\sharp$, the consequent $\overrightarrow{gs} \neq \overrightarrow{\perp}$ 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 $\mathit{last}(A^{III}_M[L]) \Rightarrow \mathit{head}(A^{III}_M[H])$. 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:

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

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 $\mathit{last}(A^{III}_M[L])$ should be replaced by $\mathit{Ensures}(\Gamma^*(M'))$.

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

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

where ϕ is the formula $(g_0 = r_0) \wedge \ldots \wedge (g_{n-1} = r_{n-1}) \wedge (g_{\mathit{this}} = r_{\mathit{this}})$. 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 [8].

The first of these conditions is again easy to show valid, since $\mathit{Requires}(\Gamma^*(M'))$ and all conjuncts in ϕ also appear as conjuncts in $\mathit{last}(A^{III}_M[L])$ 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 $A^{III}_M[L+1]$ 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 pre-condition 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

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

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

$$\begin{aligned} &\phi \\ \wedge\ &\mathit{Defined}^\sharp \\ \wedge\ &\textstyle\bigwedge_{1\leq i\leq m} r_{\text{this}} : c'_i \wedge\ a_b\rho'_i \ \rightarrow\ ce = \overrightarrow{e_E}\rho'_i \\ \wedge\ &\neg(r_{\text{this}} : c'_1 \vee \ldots \vee r_{\text{this}} : c'_p) \ \rightarrow\ \ ce = \overrightarrow{ms} \end{aligned}$$

where ϕ is as explained above, and where ce is the tuple of conditional expressions $\overrightarrow{ce_i}$, 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 $\mathit{Ensures}(\Gamma^*(M')) \wedge \phi \Rightarrow \mathit{head}(A^{III}_M[L+1])$ is established as follows. The first conjunct ϕ (actually a set of conjuncts) of $\mathit{head}(A^{III}_M[L+1])$ 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 $\overrightarrow{gs} \neq \overrightarrow{\bot}$. Every conjunct $r_{\text{this}} : c'_i \wedge\ a_b\rho'_i \ \rightarrow\ ce = \overrightarrow{e_E}\rho'_i$ 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 $\overrightarrow{e_E}\rho'_i$. Validity can thus be easily checked mechanically by simple equational reasoning and (syntactic) guard matching. Finally, validity of the conjunct $\neg(r_{\text{this}} : c'_1 \vee \ldots \vee r_{\text{this}} : c'_p) \;\rightarrow\; ce = \overrightarrow{ms}$ 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:

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

where ϕ is the formula $(g_0 = r_0) \wedge \ldots \wedge (g_{n-1} = r_{n-1}) \wedge (g_{\mathit{this}} = r_{\mathit{this}})$. 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. □