A Proof Carrying Code Framework for Inlined Reference Monitors in Java Bytecode

Mads Dam, Andreas Lundblad
School of Computer Science and Communication
Royal Institute of Technology (KTH)
Stockholm, Sweden

Abstract

We propose a lightweight approach for certification of Java bytecode monitor inlining using proof-carrying code. The main purpose of such a framework is to enable development use of monitoring for quality assurance, while minimizing the runtime overhead of monitoring, minimizing the need for changes to the load- and runtime tcb, and eliminating the need for post-shipping code rewrites with the resulting loss of liability. Policies to be enforced are specified in the ConSpec policy specification language which, roughly, express regular sequences of method calls to some fixed API. Proofs are represented as Java class files augmented with logical assertion in Floyd/Hoare logic style: Assertions are associated to each program point as well as to method entry and (exceptional and normal) exit points using standard, JML-style pre- and post-conditions. Such a proof representation is adequate in our case, as all proofs generated in our framework can be recognized in time linear in the size of the associated program. The basic proof generation strategy is to compare the effects of an actual, untrusted, inliner, with the effects of a trusted "ghost" inliner which is never actually executed, but is nonetheless present for analysis purposes. At time of receiving a program with proof annotations, it is sufficient for the receiver to plug in its own trusted ghost inliner and check the resulting, given, verification conditions, to be sure inlining has been performed correctly, of the correct policy. We have proved correctness of the approach at the Java bytecode level. A prototype implementation has been produced. We finally report on an example based on a J2ME snake game with a simple two-state policy.

1 Introduction

Program monitoring is a well-established and efficient approach to prevent potentially misbehaving software clients from causing harm, for instance by violating system integrity properties, or by accessing data to which the client is not entitled. The conceptual model is simple and familiar from policy-based management: Potentially dangerous actions by a client program are intercepted and routed to a policy decision point (pdp) in order to determine whether the actions should be allowed to proceed or not. In turn, these decisions are routed to a policy enforcement point (pep), responsible for ensuring that only policy-compliant actions are executed.

Explicit Policy Monitoring Often, clients, pdp's, and pep's, are viewed as independently executing, possibly distributed, agents, that use OS and network services for communication and synchronization. For application programs that execute in a local environment such as a Java Runtime Environment, this model, however, has a number of drawbacks:

- Concepts needed for policy enforcement such as application program, state, thread, session, and "potentially dangerous action" may be evident at the level of a Java runtime system, but not at the system, or OS level.

- The machinery needed for policy enforcement needs to interfere with both application program execution (i.e. the Java runtime system), and the relevant OS and network service primitives. It is far from obvious that this can be done without affecting program behaviour, even in the case of policy-compliant programs, for instance in the case of threading.

Monitor Inlining A complementary approach is to use monitor inlining [12]. This approach uses program rewriting to push some or all policy enforcement functionality into the client programs themselves, by code rewriting. When applicable, this has a number of advantages:

- Overhead for marshalling and demarshalling policy information between the various decision and enforcement points in the system is eliminated.

- All information needed for policy enforcement is directly available to the pdp and the pep.

- Extensions to the runtime trusted computing base (tcb) needed for policy enforcement are localized to the client program.

- By proving the inliner correct, in the sense that it enforces the policy correctly, and that it interferes with program execution only when necessary, the need for extensions (trust) can to a large extent be eliminated.

This, however, presupposes that inlining is performed under the jurisdiction of the executing agent. Otherwise, some mechanism is needed to transfer evidence that the inlining has been performed correctly from the site performing the inlining to the site using the inlined code. One method for doing this is code signing, but this reintroduces the need for trust, and the need for a large and complex tcb to support this. In this paper we examine the possibility of instead using a proof-carrying code architecture to certify policy compliance at a semantical level. This approach has some interesting advantages:

- The need for extra trust relationships on the runtime tcb is eliminated.
- The enforcement architecture can be realized in a way which is backwards compatible, in the sense that pcc-aware client programs can be executed without modification in a pcc-unaware host environment.

The cost, of course, is the need for a loadtime proof checker which in this case, however, is quite simple.

Certified Monitoring The goal is to make monitor inlining available as a quality assurance tool to the application developer in a transparent and fully automated way. Given a policy to be enforced we provide the means to automatically inline a monitor for this policy, along with a proof, as a set of annotations to the Java bytecode, that the resulting inlined program is indeed policy compliant. Upon receiving such an annotated application program, the receiving agent can then, again without user intervention, perform two essential checks:

1. A check that the statement proved is correct, i.e. that the policy claimed for the bytecode is indeed a policy accepted by the receiving agent.
2. A check that the proof itself is correct, in the sense that the conclusion —that the application program is policy adherent—indeed follows from the bytecode according to sound steps of reasoning provided by the bytecode annotations.

Inliner Our approach is as follows. Policies are given as security automata in the style of Schneider [26] in a special-purpose language, ConSpec [4], developed in the context of the EU FP6 S3MS project [1]. The policies express constraints on the allowed sequences of calls to some selected set of API methods. An inliner for such a policy converts the corresponding security automaton to snippets of Java bytecode, to be inserted at selected points in the instruction sequence for each application method. The inliner examined in this paper follows the principles of previous work in this area, including [12, 3, 9] and Erlingsson's PoET/PSLang toolset [12].

Proof Generation The key idea in proof generation is to compare the effects of the embedded, untrusted, inliner with a trusted inliner, inserted as explicit ghost instructions in each method instruction sequence. The two inliners are compared through a *monitor invariant*, expressing that the state of the embedded inliner is in synchrony with that of the ghost inliner. This monitor invariant is inserted, as annotations, at method entry, method exit, and at each method call point. In principle we would then use a wp-based proof completion procedure to propagate verification conditions to the remaining program points. Unfortunately, there is no guarantee that such an approach would be feasible. However, it turns out that it is sufficient to perform the wp propagation for the inlined code snippets and not for the client code, under some critical assumptions:

- The inlined code appears as contiguous subsequences of the entire instruction sequences in the inlined methods.
- Control transfers out of these contiguous snippets are allowed only when the monitor invariant is guaranteed to hold.
- Control transfers into the inlined snippets are allowed only when the wp-generated assertions follow from the monitor invariant.
- The embedded monitor state is represented in such a way that a simple syntactic check suffices to determine if some non-inlined instruction can have an effect on its value.

We claim that our approach can handle any inliner which adheres to these four constraints. The last constraint can be handled, in particular, by implementing the embedded monitor state as a static member of a final security state class. The important consequence is that, for instructions that do not appear in the inlined snippets, and not including `putstatic` instructions to the security state field, it is sufficient to annotate the corresponding program point by the monitor invariant to obtain a fully annotated method which is locally valid in the sense that method pre- and post-conditions match, and that each program point annotation follows from successor point annotations by simple and local reasoning alone.

Proof Recognition This approach to proof generation allows a correspondingly simple approach to proof recognition at the receiving end. Transmitted applications will be constructed from fully annotated class files from which the ghost instructions have been removed. Property (1) and (2) above can then be checked by the receiving agent simply by inserting its own ghost inliner in place of the ghost inliner used during proof generation, and by checking local validity of the resulting fully annotated program. Note that this approach makes proof recognition quite independent of the actual inliner used: As long as the proof generation principles using static final ghost states is adhered to, any inliner producing locally valid annotations will do.

Contributions The contribution of this paper is to show formally the soundness of this approach at the level of Java bytecode. We also report briefly of a prototype implementation of proof generation and proof recognition tools, the latter executable on Java enabled mobile devices.

1.1 Related Work

Proof-Carrying Code For background on proof-carrying code we refer to [23]. Our approach is based on simple Floyd-like program point annotations in the style of Bannwarth and Müller [5], and method specifications extended by pre- and post-conditions in the style of JML [15].

Security Automata Schneider [26] proposed the use of automata as a tool to formalize security policies, and monitor inlining to enforce such policies was examined in [12, 11]. The PoET/PSLang toolset by Erlingsson [11] implements monitor inlining for Java. That work represents security automata directly in terms of Java code snippets, making it difficult to formally prove correctness properties of the approach. Subsequent work on monitor inlining includes [3] and [9]. Edit automata [21, 22] are examples of security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event insertions, for instance to recover gracefully from policy violations.

Type-Based Enforcement A number of authors have considered type systems for security policy enforcement, e.g. [27, 28, 16, 8, 13]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al [16]. The Mobile system uses a simple library extension to Java bytecode to help managing updates to the security state. The use of linear types allows a type system to localize security-relevant actions to objects that have been suitably unpacked, and the type system can then use this property to check for policy compliance.

Mainly, in comparison to [16], we consider our work as examining to which extent a pcc approach could be a suitable alternative to the use of types. In principle, a pcc approach could be useful, as it might allow richer classes of programs and inlining strategies to be admitted. Although it is premature to draw any firm conclusions, we believe this may indeed be the case. Compared to our work, the approach of [16] addresses threads, but not inheritance. An extension of our work to threads is left to future work. The security policies considered are not directly comparable. Mobile enforces per-object policies, whereas the policies enforced in our work (as in most work on IRM enforcement) are per session. Since Mobile leaves security state tests and updates as primitives, it is quite likely that Mobile could be adapted, at least to some forms of per session policies. On the other hand, to handle per-object policies our approach would need to be extended to track object references. Finally, it is worth noting that Mobile relies on a specific inlining strategy, whereas our approach is less sensitive to this. Indeed, [3] shows how to generate annotations under rather modest assumptions on the inliner, by fixing control points immediately before and after each method call at which the embedded state must be correctly updated.

1.2 Overview of the Paper

After a few notational preliminaries we present the JVM execution model in section 3. In section 4 the state assertion language is introduced, and in section 5 we address method and program annotations, give the conditions for (local and global) validity used in the paper, and sketch a soundness proof for local validity. We briefly describe the ConSpec language in section 6, and (our version of) security automaton is introduced in section 7. The inlining algorithm is specified is section 8. Section 9 introduces the ghost inliner, and section 10, then, presents the main results of the paper, namely the algorithms for proof generation and proof recognition, including soundness proofs. Finally, section 11 reports briefly on our prototype implementation, and we conclude by discussing some open issues and directions for future work. A standard JVM semantics for instructions used by the inliner and some proofs are deferred to the appendix.

1.3 Acknowledgements

Thanks to Irem Aktug, Dilian Gurov, Bart Jacobs and Johan Linde for useful discussions on many topics related to monitor inlining.

2 Preliminaries

A stack is an element $s \in A^*$, $s[i]$ is the i'th element of s counting from the top, if defined, $v :: s$ is the stack obtained by pushing v on the top of s, and $\mathrm{lth}(s)$ is the length of s. We write $\mathbf{a}$ for sequences $a_0, \ldots, a_n$.

3 Program Model

We assume that the reader is familiar with Java bytecode syntax, the Java Virtual Machine (JVM), and formalisations of the JVM such as [14]). Here, we only present components of the JVM, that are essential for the definitions in the rest of the text. A few simplifications have been made in the presentation. In particular, we disregard static initializers and to ease notation a little we ignore issues concerning overloading.

Classes and Types We use c for class names, m for method names, and f for field names. For our purpose it suffices to think of class names as fully qualified. A type τ is either a primitive type, including integers, or an object type, determined by a class name c. An object type determines a set of fields and methods defined for that type through its class declaration. The class declarations induce a class hierarchy, and we write $c_1 <: c_2$ if c_1 is a subclass of c_2. If c defines m (declares f) explicitly, then c defines (declares) $c.m$ ($c.f$). Otherwise, c defines $c'.m$ (declares $c'.f$) if c is the smallest superclass of c' that contains an explicit definition (declaration) of $c.m$ ($c.f$). Single inheritance ensures that definitions/declarations are unique, if they exist.

Values and Objects Each type τ determines a set $\|\tau\|$ of values of that type, and $\mathit{Val} = \bigcup_\tau \|\tau\|$ is the set of values, ranged over by v. Integers are infinite precision. Values of object type are (typed) locations $\ell \in \mathit{Loc}$, mapped to objects, or arrays, by a heap h. The typing assertion $h \vdash v : c$ asserts that v is location Loc, and that in the typed heap h, Loc is defined and of type c, and similarly for arrays. Typing preserves the subclass relation, in the sense that if $h \vdash v : c$ and $c <: c'$ then $h \vdash v : c'$ as well. For objects, it suffices to assume that if $h \vdash v : c$ then the object $h(v)$ determines a field $h(v).f$ (method $h(v).m$) whenever f (m) is declared (defined) in c. Non-static fields are values. Static fields are identified with field references of the form $c.f$. To handle those, heaps are extended to assignments of values to field references.

Methods A method environment is a mapping Γ, usually elided, taking method references to their definitions. To simplify notation, method overloading is not considered, so a method is uniquely identified by a method reference of the form $M = c.m$. A method $c.m$ has type $\gamma \to \tau$, written $c.m : \gamma \to \tau$, if γ is the list of argument types and τ is the return type of the method. A method definition is a pair (I, H) consisting of an instruction array I and an exception handler array H. We use the notation $M[L] = \iota$ to indicate that $\Gamma(M) = (I, H)$ and I_L is defined and equal to the instruction ι. 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.

Configurations and Transitions 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, r), where:

- M is a reference to the currently executing method.
- The *program counter* pc is an index into the instruction array of M.
- The *operand stack* $s \in Val^*$ is the stack of values currently being operated on, including local variables.
- r is an array of *registers*, or local variables.

We assume a transition relation $\rightarrow_{\mathrm{JVM}}$ on JVM configurations. In appendix A we give details for instructions used in our inliner. A configuration of the shape $C = ((M, pc, s, r) :: R, h)$ is *calling*, if $M[pc]$ is an invoke instruction, and it is *returning normally*, if $M[pc]$ is a return instruction. For exceptional configurations the top frame has the form (b) where b is the location of an exceptional object, i.e. of class THROWABLE. Such a configuration is called *exceptional*. With reference to the semantics shown in appendix A we say that C is *returning exceptionally* if C is exceptional, and if $C \rightarrow_{\mathrm{JVM}} C'$ implies that C' are exceptional as well. I.e. the normal frame immediately succeeding the top exceptional frame in C is popped in C', if C' is exceptional as well.

Programs and Types For the purpose of this paper we can view a *program* P as a collection of class declarations determining types of fields and methods belonging to classes in P, and a method environment Γ giving a method definition of each method in P. An *execution* E of a program P is a (possibly infinite) sequence of JVM configurations $C_0 C_1 \ldots$ where C_0 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, $pc = 0$, Γ set up according to P, and for each $i \geq 0$, $C_i \rightarrow_{\mathrm{JVM}} C_{i+1}$. 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. [19]).

API Method Calls The only non-standard aspect of $\rightarrow_{\mathrm{JVM}}$ 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 in, e.g. [24]. In this sense, we do not practice *complete mediation* [25]. 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.

4 Assertions

In the appendix we present a transition semantics for a collection of instructions $\mathcal{I}$ used by the inliner. The main use of this semantics is to serve as justification of a wp characterization, also shown in the appendix. The wp characterization uses Floyd-style annotations of each program point (possible value of *pc* for the given method) in a standard fashion.

Assertion Syntax The annotations are given in an assertion language similar to that introduced by Bannwarth and Müller [5]. Assertions a and expressions e used in this language are given in the following abstract syntax:

$$\begin{aligned} e \in \mathit{Exp} &::= v \mid e.f \mid c.f \mid \mathrm{s}_i \mid \mathrm{r}_i \mid e\ \mathit{BOp}\ e \\ a \in \mathit{Ast} &::= e\ \mathit{BRel}\ e \mid a \wedge a \mid \neg a \mid e : c \end{aligned}$$

where $i \in \omega$. We use BOp as a generic binary operator among integer addition, subtraction, and multiplication, and similarly BRel ranges over integer less-than and equality. The expression s_i accesses the i'th element of the operation stack, and r_i accesses the i'th register; a *heap assertion* is one that does not reference the stack, or any of the registers. Disjunction ($\vee$), implication ($\Rightarrow$) and unary minus are defined as usual.

Assertion Semantics Expressions and assertions are evaluated relative to a configuration $C = (R, h)$. Expressions return elements in the lifted domain of integers and locations as follows:

- $|e.f|C = \begin{cases} h(|e|C).f, & |e|C \text{ is defined and of} \\ & \text{object type } c \text{ such that } c \text{ declares } f \\ \perp, & \text{otherwise} \end{cases}$

- $|c.f|C = \begin{cases} h(c.f), & \text{if } c \text{ declares } f \\ \perp, & \text{otherwise} \end{cases}$

- $|\mathrm{s}_i|C = \begin{cases} s[i], & \text{if } \exists R, h, M, pc, s, r. C = ((M, pc, s, r) :: R, h), \text{ and} \\ & \mathrm{lth}(s) \geq i \\ \perp, & \text{otherwise} \end{cases}$
- $|\mathrm{r}_i|C = \begin{cases} r[i], & \text{if } \exists R, h, M, pc, s, r. C = ((M, pc, s, r) :: R, h), \text{ and} \\ & \mathrm{lth}(r) \geq i \\ \perp, & \text{otherwise} \end{cases}$
- $|e_1 \; BOp \; e_2|C = |e_1|C \; BOp \; |e_2|C$

Assertions are then evaluated as expected:

- $|e_1 \; BRel \; e_2|C = \begin{cases} \texttt{TRUE} & \text{if } |e_1|C \text{ and } |e_2|C \text{ are both defined} \\ & \text{and, if they are, } |e_1|C \; BRel \; |e_2|C \\ \texttt{FALSE} & \text{otherwise} \end{cases}$
- $|e : c|C = \begin{cases} \texttt{TRUE} & \text{if } |e|C \text{ is defined, of object type, and} \\ & h \vdash |e|C : c \\ \texttt{FALSE} & \text{otherwise ,} \end{cases}$

and boolean operations are evaluated as usual.

We define a syntactical macro for conditional expressions $\mathrm{IF}(a_0, a_1, a_2) = (a_0 \Rightarrow a_1) \wedge (\neg a_0 \Rightarrow a_2)$ as well as a generalized form which takes an array of conditions, an array of values and an "else"-value

$$\mathrm{SELECT}(\mathbf{a_1}, \mathbf{a_2}, a_{else}) = \mathrm{IF}(a_{1,0}, a_{2,0}, \mathrm{IF}(a_{1,1}, a_{2,1}, \ldots, \mathrm{IF}(a_{1,n}, a_{2,n}, a_{else}) \ldots)).$$

5 Extended Method Definitions

In this section we extend method definitions by assertion annotations at each program point, and by pre- and post-conditions at method entry and (normal or exceptional) return.

Definition 1 (Extended Method Definition) *An* extended method definition *is a tuple (I, H, A, req, ens, exs) where*

1. *(I, H) is a method definition,*
2. *A is an array of assertions such that $|I| = |A|$, representing a proof outline of the local validity of the method,*
3. *req (requires), ens (ensures), exs (exsures) are heap assertions.*

Extended method environments *take method references to extended method definitions, and an* extended program *is a program with an extended environment.*

We often refer to assertions *req* as the precondition, and to *ens* and *exs* as the (normal and exceptional) postconditions of the given method. The pre- and post-conditions are assumed to be heap assertions for simplicity: This is sufficient for the purpose of the present paper, and it makes for a much simpler "calling convention". For extended programs, the notions of transition and execution are not affected by the presence of annotations. An extended program is valid, if all annotations are validated by their corresponding configurations in any execution starting in a configuration satisfying the initial precondition. Or, in other words:

Definition 2 (Extended Program Validity) *An extended program P is* valid *if for each maximal execution $E = C_0C_1\cdots$ of P such that $|req_{\texttt{main}}|C_0 = \texttt{TRUE}$, for each $i : 0 \leq i (\leq |E|)$, if C_i has the shape $((M, pc, s, r) :: R, h)$ and $\Gamma(M) = (I, H, A, req, ens, exs)$ then*

1. $|A_{pc}|C_i = \texttt{TRUE}$
2. *If $pc = 1$ then $|req|C_i = \texttt{TRUE}$*
3. *If C_i is returning normally then $|ens|C_i = \texttt{TRUE}$*
4. *If C_i is returning exceptionally, and C_i has the shape $((b) :: (M, pc, s, r) :: R, h)$ then $|exs|C_i = \texttt{TRUE}$.*

Validity is the basic semantic notion of interest. It is, however, neither modular, nor tractable. In its place we work with the following notion of local validity expressed using weakest preconditions and local conditions on control transfer. To explain this let $\Gamma(M) = (I, H, A, req, ens, exs)$. The weakest precondition function $wp_M : dom(I) \rightarrow Ast$ computes weakest preconditions according to table 1. The definition uses the auxillary functions *shift* and *unshift* which increments, resp. decrements, each stack index by one. Also, by convention, $A_L = \texttt{FALSE}$ whenever $L > |I|$.

The wp_M function takes method pre- and post-conditions from M, and M:s assignment of assertions to program points, to compute the weakest assertion required at a given program point, for the "successor assertions" to hold at "successor configurations", including, in particular, the exsures assertion in case the instruction concerned is `ATHROW`.

Our approach to verification condition generation is based on weakest preconditions for instructions produced by the inliner, and an easily computable approximation of the weakest precondition for other instructions. For this reason it is useful to generalize the setting slightly. In the context of an extended method definition $\Gamma(M) = (I, H, A, req, ens, exs)$, an *annotation transformer* is a mapping $f_M : Dom(I) \rightarrow Ast$ which typically, for a given program point, iterates the *wp* computation one step.

The correctness of the wp definition is captured by the following key per-instruction property:

I_L	A_L
`INSTANCEOF` c	$A_{L+1}[\mathrm{s}_0 : c/\mathrm{s}_0]$
`ALOAD` n	$\mathit{unshift}(A_{L+1}[\mathrm{r}_n/\mathrm{s}_0])$
`ASTORE` n	$(\mathit{shift}(A_{L+1})) \wedge \mathrm{s}_0 = \mathrm{r}_n$
`ATHROW`	$\mathrm{SELECT}((\mathrm{s}_0 : c \wedge b \leq L < e)_{(b,e,L',c)\in H}, (A_{L'})_{(b,e,L',c)\in H}, \mathit{exs})$
`DUP`	$\mathit{unshift}(A_{L+1}[\mathrm{s}_1/\mathrm{s}_0])$
`GETFIELD` f	$\mathit{unshift}(A_{L+1}[\mathrm{s}_0.f/\mathrm{s}_0])$
`GETSTATIC` $c.f$	$\mathit{unshift}(A_{L+1}[c.f/\mathrm{s}_0])$
`GOTO` L'	$A_{L'}$
`ICONST_`n	$\mathit{unshift}(A_{L+1}[n/\mathrm{s}_0])$
`IF_ICMPEQ` L'	$\mathrm{IF}(\mathrm{s}_0 = \mathrm{s}_1, \mathit{shift}^2(A_{L'}), \mathit{shift}^2(A_{L+1}))$
`IFEQ` L'	$\mathrm{IF}(\mathrm{s}_0 = 0, \mathit{shift}(A_{L'}), \mathit{shift}(A_{L+1}))$
`PUTSTATIC` $c.f$	$\mathit{shift}(A_{L+1})[\mathrm{s}_0/c.f]$
`LDC` v	$\mathit{unshift}(A_{L+1}[v/\mathrm{s}_0])$
`INVOKESTATIC System.exit`	`TRUE`

Table 1: Specification of the wp_M function

Property 1 *Let $\Gamma(M) = (I, H, A, req, ens, exs)$. The annotation transformer f_M is* valid *on a configuration $C = ((M, pc, s, r) :: R, h)$ which is neither calling nor returning, if the following properties hold:*

1. *Suppose that $C \rightarrow_{JVM} C' = ((M', pc', s', r') :: R'h'$. Then $|A_{pc'}|C' = |wp_M(A)_{pc}|C$.*
2. *Suppose that $C \rightarrow_{JVM} C' = ((b') :: (M, pc', s', r') :: R', h') \rightarrow_{JVM} C'' = ((M, pc'', s'', r'') :: R'', h'')$ (so that C' is not returning exceptionally). Then $|A_{pc''}|C'' \equiv |wp_M(A)_{pc}|C$.*
3. *Suppose that $C \rightarrow_{JVM} C' = ((b') :: (M', pc', s', r') :: R', h') \rightarrow_{JVM} ((b'') :: (M'', pc'', s'', r'') :: R'', h'')$ (so that C' is returning exceptionally). Then $|exs|C' \equiv |wp_M(A)_{pc}|C$.*

We phrase this as a property rather than as a lemma as we for now only establish it for instructions that are used by the inliner.

Lemma 1 *The annotation transformer wp_M defined in table 1 is valid on all configurations $C = ((M, pc, s, r) :: R, h)$ for which wp_M is defined.*

Proof The proof is a case analysis on the instruction $M[pc]$. Here we address only the case $M[pc] =$ `ATHROW`. The remaining cases are straightforward.

We first prove 2. If $C \rightarrow_{\mathrm{JVM}} C'$ then C' has the shape $((l) :: (M, pc, s, r) :: R, h)$, and if C' is not returning exceptionally then $C'' = ((M, pc', l, r) ::$

$R, h)$ such that (pc, l, h) matches an earliest entry (b, e, pc', c) in the exception handler table. We then just need to use the assumption $|A_{pc'}|C''$ to conclude that $|wp_M(A)_{pc}|C = \texttt{TRUE}$. For the converse implication, if $|wp_M(A)_{pc}|C = \texttt{TRUE}$, since C' is not returning exceptionally, this can only be because there is some earliest entry (b, e, pc', c) that matches l and pc, making $|A_{pc'}|C'' = \texttt{TRUE}$ as required. $\square$

We can then define a suitable notion of local validity in the following way:

Definition 3 (Local Validity) *Let an extended method definition $M = (I, H, A, req, ens, exs)$ be given.*

1. *M is* locally valid*, if there is a valid annotation transformer f_M such that*
   - *(a) $req \Rightarrow A_0$*
   - *(b) $A_L \Rightarrow f(A, L)$ for all program points $L \in Dom(I)$.*
2. *The extended program P is* locally valid*, if each extended method definition in P is locally valid, and if, whenever $M = (I, H, A, req, ens, exs)$ and I_L is an invoke instruction with (static) method reference c.m then, whenever c' is any class defining m, and the method definition of $c'.m$ is $M' = (I', H', A', req', ens', exs')$, then*
   - *(a) $A_L \Rightarrow req'$,*
   - *(b) $ens' \Rightarrow A_{L+1}$,*
   - *(c) if there is a first entry (b, e, L', c'') in H such that $b \le L < e$, and $c' <: c''$ then $exs' \Rightarrow A_{L'}$, and*
   - *(d) if no (b, e, L', c'') as in (c) exists, then $exs' \Rightarrow exs$.*

The account of dynamic call resolution in def. 3.2 is quite crude, but the details are unimportant since, in this paper, pre- and post-conditions are always identical and common to all methods in P. The important property is the local validity is sufficient to prove (global) validity:

Theorem 1 *For any extended program P, if P is locally valid then P is valid.*

Proof (Sketch) We assume that P is locally valid, and that f_M has the required property. To show that P is valid it suffices to verify properties 1.–4. for each maximal execution E of P that validates the initial precondition of `main`, and each $i : 0 \le i \le |E|$. The proof is by induction on i. If C_{i-1} is normal (to allow us to use the induction hypothesis) and neither calling, nor returning, we use lemma 1 and local validity to conclude. If C_{i-1} is calling then, by def. 3.2.a (since pre- and postconditions are unable to address the stack and the registers) we obtain that the precondition req' of the called method holds, and hence, by local validity, also the relevant program point annotation. If C_{i-1} is returning normally, we use property 3.2.b instead. Finally assume C_{i-1} is exceptional. Then there must exist a $j < i - 1$ such that C_j is normal and all frames $k : j < k < i$ are exceptional. There are two cases: Either $j = i - 2$ and C_i is normal (so the exception is caught by the method throwing the exception), or not. In the former case we conclude by lemma 1.2, local validity, and the induction hypothesis. In the latter case, by local validity, the induction hypothesis, and lemma 1.3, the exceptional postcondition for the top frame holds for C_j. By condition 3.4 it follows that the exceptional postcondition holds for the top frame for all $k : j < k < i$, and the result the follows by condition 3.3 or 3.4, depending on whether C_i is normal or exceptional. $\square$

6 Security Specifications

We distinguish between two types of security specifications: *policies* and *contracts*. The only difference between policies and contracts is in the way they are used. A policy resides in the host-system and is defined by the consumer. It specifies what behaviors are acceptable for programs executing on the host. A contract on the other hand is a specification how the program behaves. It is defined by the producer, and is distributed with the program.

ConSpec In this paper we consider security specifications written in the ConSpec language [4, 2]. ConSpec is similar to PSlang [11], but more constrained, in order to allow for a decidable entailment (containment) problem. An example of a ConSpec specification is given in fig. 1. The syntax is intended to be largely self-explanatory: The specification in figure 1 states that the program has to ask the user for permission each time it intends to send a file over bluetooth. No exception may arise during the user query. A ConSpec specification tells when and with what arguments an API method may be invoked. If the specification has one or more constraints on a method, the method is said to be *security relevant* and we refer to invocations of this method as *security relevant calls*. In the example, there are two security relevant methods, `GUI.approveSend` and `Bluetooth.obexSend`. The specification expresses the constraints on security relevant actions in terms of guarded commands where the guards are side-effect free and terminating boolean expressions. These expressions may involve constants, method call parameters, objects fields, and values returned by accessor or test methods that are guaranteed to be side-effect free and terminating.

Scope Definitions The set of API's considered is assumed to be fixed. API calls, whether security relevant or not, are trusted. Non-API calls, on

```
SCOPE Session

SECURITY STATE boolean sendApproved = false;

AFTER answer = GUI.approveSend()
    PERFORM
         answer -> { sendApproved = true;  }
        !answer -> { sendApproved = false; }

EXCEPTIONAL GUI.approveSend()
    PERFORM
        false -> { }

BEFORE Bluetooth.obexSend(String file)
    PERFORM
        sendApproved -> { sendApproved = false; }
```

Figure 1: A security specification example written in ConSpec.

the other hand, are untrusted. The *scope declaration* tells whether the specification should hold on a single object, for each session, for multiple sessions, or globally, at the level of the entire virtual machine. In this paper we focus on the session scope. That is, monitoring is kicked off together with some client program, monitors all security-relevant calls of untrusted methods belonging to that program, and stops once the program is terminated.

Security State The *security state* declaration is a list of variable declarations which specifies the variables in the monitor state. The work reported here is not very sensitive to the choice of state variables, but in order to decide policy entailment by a standard language inclusion test it is important that the state variables range over finite domains.

Events and Clauses An *event clause* defines an action related to a security relevant action. The event modifiers `BEFORE`, `AFTER` and `EXCEPTIONAL` specifies if the action should be taken before the call to the method, upon normal return from the method or upon exceptional return from the method. Guards are evaluated top to bottom, in order to obtain a deterministic semantics and hence, when it applies, a tractable entailment problem. If no clause guards hold, the call is violating.

7 Security Automata

ConSpec policies are formalized in terms of security automata. The notion of security automata was introduced by Schneider [26]. In this paper we view a *security automaton* as an automaton $\mathcal{A} = (Q, \delta, q_0)$ where Q is a countable (not necessarily finite) set of states, Σ is the alphabet of security relevant actions (sra's, typically α), $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \rightharpoonup Q$ is a (partial) transition functions. All states $q \in Q$ are viewed as accepting.

Notation 1 *For a security automaton $\mathcal{A} = (Q, \delta, q_0)$, $q \xrightarrow{\alpha} q'$ abbreviates the condition $q' = \delta(q, \alpha)$.*

All security automata considered in this paper are assumed to be generated by a ConSpec policy, in a manner which we do not make precise here. The details are not complicated. We refer to [3] for details. For the purpose of this paper it suffices to note that security automaton actions are induced by clauses in a corresponding ConSpec specification in the style of 1, involving guard evaluation and security state updates.

Pre-actions and Post-actions The alphabet Σ is partitioned into pre-actions and (normal or exceptional) post-actions. Pre-actions $\alpha \in \Sigma^{\uparrow}$ have the form $(c.m, (v_1, \ldots, v_n), h)$ where $(v_1, \ldots, v_n)$ is the sequence of parameters to API method $c.m$, and h is the heap at time of call. Normal post-actions $\alpha \in \Sigma^{\downarrow}$ have the form $(c.m, (v_1, \ldots, v_n), h_1, h_2, v)$ where $(v_1, \ldots, v_n)$ is the sequence of parameters to API method $c.m$, h_1 is heap at time of call, h_2 is heap at time of normal return, and v is returned value. Exceptional post-actions $\alpha \in \Sigma^{\Downarrow}$ have the same form, but now the call returns exceptionally, h_2 is the heap at time of exceptional return, and v is the returned THROWABLE object. Then, $\Sigma = \Sigma^{\uparrow} \cup \Sigma^{\downarrow} \cup \Sigma^{\Downarrow}$.

Security-relevant Actions We explain how security-relevant actions are produced under execution. Consider an execution $E = C_0 C_1 \cdots$. Let $C_i = ((M_i, pc_i, s_i :: l :: s_i', r_i) :: R_i, h_i)$ be a configuration such that $M[pc_i] =$ `invokevirtual` m (other invoke instructions are handled similarly). Then $sra(C_i)$, the security-relevant pre-action of C_i, is $(c.m, s_i, h_i)$, if the smallest superclass defining m of (the class of) $h(l)$ under <: is the API class c. Note that the type and arity of $c.m$ is determined since overloading is disregarded, and since the program has been bytecode verified.

For post-actions we assume for simplicity that all API methods return a value. Let $C_i = ((M_i, pc_i, s_i :: l :: s_i', r_i) :: R_i, h_i)$ be a configuration as above such that $M[pc_i] =$ `invokevirtual` m, and suppose that $C_{i+1} = ((M_i, pc_{i+1}, v :: s_i', r_i) :: R_i, h_i')$. Then $sra(C_i, C_{i+1})$, the security-relevant (normal) post-action of the transition $C_i \rightarrow_{\mathrm{JVM}} C_{i+1}$, is $(c.m, s_i, h_i, h_{i+1}, v)$, again provided that c is an API class, and that c is the smallest superclass defining m of $h_1(l)$. Similarly, if $C_{i+1} = ((l) :: (M_i, pc_i, s_i :: l :: s'_i, r_i) :: R_i, h'_i)$ then $sra(C_i, C_{i+1})$, the security-relevant (exceptional) post-action of the transition $C_i \rightarrow_{\text{JVM}} C_{i+1}$, is $(c.m, s_i, h_i, h_{i+1}, l)$, where c is determined as above.

In cases other than the above three, $sra(C) = sra(C, C') = \varepsilon$, the empty string.

Policy Adherence The security-relevant trace of E, $srt(E)$, now, is determined co-inductively in the following way:

$$\begin{aligned} srt(\varepsilon) &= \varepsilon \\ srt(C) &= sra(C) \\ srt(C_0C_1E) &= sra(C_0)sra(C_0, C_1)srt(C_1E) \end{aligned}$$

Definition 4 (Policy Adherence) *The program P adheres to security policy $\mathcal{P}_{\mathcal{A}}$, if for all executions E of P, either $srt(E) \in \mathcal{P}_{\mathcal{A}}$ or $srt(E) = w sra(C, C') \wedge w \in \mathcal{P}_{\mathcal{A}}$ for some w.*

8 Inlining

By *inlining* we refer to the procedure of compiling a policy into a JVM based reference monitor and embedding this monitor into a target program. A program with a correctly inlined reference monitor (IRM) is guaranteed to adhere to the policy which the program was rewritten to comply with.

Inlining is in this paper assumed to be *compositional* in the sense that one method can be treated in isolation. This means that proper security state updates, checks and interventions due to security relevant actions must take place before the method returns or calls another method.

Our inliner achieves this by initializing the security state before anything else in the main method and replacing each `invokevirtual` $c.m$ by JVM code corresponding to the pseudo code in table 2 (see appendix B). The replacement is referred to as a block of inlined code. The only way to enter this block of code is through the first instruction. We assume that no instructions in a block of inlined code (except explicit throw instructinos) will raise any exceptions. The inliner stores arguments to the virtual call for use in event handler code, once virtual call resolution has been performed. Once the dynamic call has been identified each piece of event code evaluates guards by reference to the embedded monitor state and the stored arguments, and updates the state according to the matching clause, alt. exits, if no matching clause is found.

The embedded monitor state is represented as a static field **ms** of a final security state class, named to avoid clashes with classes in the target program. This choice of representation relies on the following fact of JVM

```
               L:  Save parameters in local variables
                   Assign target object to local variable $t$
                   If $t : c^N$ then $c^N.m$-before event code
                   else if $t : c^{N-1}$ then $c^{N-1}.m$-before event code
                                 ⋮
                   else if $t : c^1$ then $c^1.m$-before event code
     $L_{ExcStart}$:  Invokevirtual $c.m$
       $L_{ExcEnd}$:  Goto $L_{HandlerEnd}$
 $L_{HandlerStart}$:  If $t : c^N$ then $c^N.m$-exceptional event code
                   else if $t : c^{N-1}$ then $c^{N-1}.m$-exceptional event code
                                 ⋮
                   else if $t : c^1$ then $c^1.m$-exceptional event code
                   Rethrow exception
   $L_{HandlerEnd}$:  If $t : c^N$ then $c^N.m$-after event code
                   else if $t : c^{N-1}$ then $c^{N-1}.m$-after event code
                                 ⋮
                   else if $t : c^1$ then $c^1.m$-after event code

 where $t$ represents the target object of the method call and
 $c^1, \ldots, c^N$ are all the API-classes defining or overriding $m$
 ordered such that for all $i < j$, $c^i <: c^j$.
```

Figure 2: Pseudo-code for the inlining replacement of L: `invokevirtual` $c.m$.

execution which we state as an assumption due to our open-ended treatment of large parts of the JVM instruction set.

Assumption 1 *Suppose that c is final and that f is static. Assume that $C = ((M, pc, s, r) :: R, h) \rightarrow_{JVM} C'$ and $M[pc] \neq$* `putstatic` *$c.f$. If $|c.f = v|C =$ **TRUE** then $|c.f = v|C' =$ **TRUE**.*

In other words, the only instruction which can affect the value stored in a static field f of a final class c is an explicit assignment to $c.f$, and in particular, the assumption ensures that instructions originating from the target program are unable to affect the embedded monitor state.

We refer to the method resulting from inlining a method M with a policy $\mathcal{P}_{\mathcal{A}}$ as $I(M, \mathcal{A})$, and $I(P, \mathcal{A})$ is the result of inlining each method in P with

$\mathcal{P}_{\mathcal{A}}$.

Theorem 2 (Soundness of Inliner) *The inlined method $I(M, \mathcal{P})$ adheres to the policy $\mathcal{P}$.*

Proof This is a consequence of corollary 3 and theorem 5 proved below.□

9 Ghost Inlining

The first step in proof generation is to produce a trusted inlined ghost monitor with which the actually inlined monitor can be compared. The ghost inliner uses special-purpose ghost instructions operating on a ghost state that is accessible exclusively to the ghost monitor. These ghost instructions are never actually executed, but they appear in intermediate representations of the method bytecodes in order to produce and recognize the adherence proofs.

9.1 Ghost Instructions

The ghost instructions are guarded multi-assignment commands for updating the ghost state, and for evaluating and storing method call arguments and dynamic class identities. A ghost instruction is written in the following way

$$\langle \mathbf{x}^g := a_1 \rightarrow \mathbf{e}_1 \mid \ldots \mid a_n \rightarrow \mathbf{e}_n \rangle \tag{1}$$

where $\mathbf{x}^g$ is a vector of ghost variables, a_i are guard assertions and $\mathbf{e_i}$ are expression vectors of the same type and dimension as $\mathbf{x}^g$. The guards a_i and expressions $\mathbf{e}_{i,j}$ may refer to the ghost variables. Typically, each $\mathbf{x}_i$ will determine a unique static field of some final security state object with state $\mathbf{x}$. When such a ghost assignment is executed the first value, $\mathbf{e}_i$, whose guard, a_i, is satisfied is assigned to $\mathbf{x}^g$. In case no guard is satisfied the undefined state, $\perp$, is assigned to $\mathbf{x}^g$. Given the postcondition A, the weakest precondition for the instruction (1) above, labeled L, is computed as

$$wp_M(L) = \text{SELECT}((a_1, \ldots, a_n), (\phi[\mathbf{e}_1/\mathbf{x}^g], \ldots, \phi[\mathbf{e}_n/\mathbf{x}^g]), \phi[\perp/x^g])$$

9.2 Ghost Inlining

The policy adherence proofs are closely related to the notion of ghost inlining. A ghost inlining is an extra (trusted) inlining, on top of the actual inlining, implemented with ghost instructions. The ghost instructions are just as the embedded IRM, constructed from the guarded updates in the event clauses and the insertion points are determined by the method signatures and corresponding invoke instructions. The sole purpose of the ghost inlining is to act as a reference when verifying the actual inlining. In other words, the proof shows that a security relevant method is called if and only if this is permitted by the ghost IRM. It does this by showing that the following properties hold:

1. The state of the embedded IRM is in sync with the ghost state before each security relevant method call.

2. The security relevant method is called only when permitted in the state of the embedded IRM.

The ghost inlining for a method is generated by the function GI. This function takes a method definition and a policy, represented by a security automaton $\mathcal{A}$, and returns an array of instructions with ghost instructions inserted. Similarly, $GI(P, \mathcal{P}_{\mathcal{A}})$ is the result of ghost inlining each method in P. The ghost inlining is done according to figure 3 which is an implementation of the schema in figure 2. Special care is taken to the main method in which a ghost update is inserted which initializes all ghost security state fields.

In fig. 3, t is the target object, arg_i the identifier for the i'th argument, and **ms** is the embedded monitor state.

First, ghost representations of the target and argument are stored as new local variables t^g, $arg_1^g, \ldots, arg_n^g$. Then, a ghost assignment is enacted corresponding to the pre-action applying to the dynamically resolved method call. Similar to figure 2, each *BEFORE*, *AFTER*, and *EXCEPTIONAL* clause evaluates the guards relevant to the resolved method call and updates the ghost state accordingly. This evaluation uses ghost versions of the target and arguments. We do not need to worry about the representation of the ghost state itself, other than it is mapped to an actual automaton state value by the heap after ghost inlining.

An event clause in ConSpec updates the security state variables through a list of asignments while a ghost instruction does this in one vector assignment. It is however easy to convert the former to the latter through back substitution. For this purpose we define

$$collapse((x_1 := e_1, \ldots, x_n := e_n), \mathbf{e}) = \begin{cases} \mathbf{e} & \text{if } n = 0 \\ \mathbf{e}[e_n/x_n] & \text{otherwise} \end{cases}$$

Given a monitor state $\mathbf{ms} = (x, y, z)$, a ConSpec state update $s = \mathtt{x} := 7; \mathtt{z} := \mathtt{z} + 2; \mathtt{y} := \mathtt{x} + \mathtt{z};$ can be collapsed to the ghost update $\mathbf{ms} := collapse(s, \mathbf{ms})$ and written as $(x, y, z) := (7, 7 + 2, z + 2)$.

The ghost inliner does not add any entry in the exception table but assumes that an exception handler $(L, L+1, Throwable, L_{ExcHandler})$ exists for each instruction L:`invokevirtual` $c.m$ whose exceptional return could be security relevant.

```
$L$:              $\langle (t^g, arg_1{}^g, \ldots, arg_n{}^g) :=$ TRUE $\rightarrow (s_n, \ldots, s_0) \rangle$

                  $\langle \mathbf{ms}^g$ := $t^g : c^N$      $\rightarrow BEFORE[c^N.m]$
                              | $t^g : c^{N-1}$  $\rightarrow BEFORE[c^{N-1}.m]$
                                                 $\vdots$
                              | $t^g : c^1$      $\rightarrow BEFORE[c^1.m]$
                              | TRUE             $\rightarrow \mathbf{ms}^g \rangle$

                  invokevirtual c.m

                  $\langle \mathbf{ms}^g$ := $t^g : c^N$      $\rightarrow AFTER[c^N.m]$
                              | $t^g : c^{N-1}$  $\rightarrow AFTER[c^{N-1}.m]$
                                                 $\vdots$
                              | $t^g : c^1$      $\rightarrow AFTER[c^1.m]$
                              | TRUE             $\rightarrow \mathbf{ms}^g \rangle$

                  $\vdots$

$L_{ExcHandler}$: $\langle \mathbf{ms}^g$ := $t^g : c^N$      $\rightarrow EXCEPTIONAL[c^N.m]$
                              | $t^g : c^{N-1}$  $\rightarrow EXCEPTIONAL[c^{N-1}.m]$
                                                 $\vdots$
                              | $t^g : c^1$      $\rightarrow EXCEPTIONAL[c^1.m]$
                              | TRUE             $\rightarrow \mathbf{ms}^g \rangle$
```

Figure 3: Ghost implementation of pseudo code in figure 2

The key property of the ghost inliner is that it exactly reflects the behaviour of the underlying automaton running on the programs security relevant actions. We go on to make this intuition precise.

Use $\mathbf{ms}^g(C)$ to abbreviate the value of $\mathbf{ms}^g$ in the heap of C. Let $E = C_0 \cdots C_{i_0} \cdots C_{i_n} \cdots C_m$ be an execution of $GI(P, \mathcal{P}_{\mathcal{A}})$ such that $\mathbf{ms}^g(C_m) \neq \bot$ and such that $i_0, \ldots, i_n$ are all those j for which C_j is a calling configuration, of an API method $c.m$. Then, E has the shape

$$C_0 \cdots C_{i_0 - 1} C_{i_0} C_{i_0+1} C_{i_0+2} \cdots C_{i_n - 1} C_{i_n} \cdots C_m$$

such that each transition $C_{i_j - 1} \rightarrow_{\text{JVM}} C_{i_j}$ is the execution of a ghost instruction corresponding to a *BEFORE* block in figure 3, and each transition $C_{i_j+1} \rightarrow_{\text{JVM}} C_{i_j+2}$ is the execution of the corresponding *AFTER*/ *EXCEPTIONAL* block. Note that it may be that $m = i_n$, and if $m > i_n + 1$

then again $C_{i_n+1} \to_{\mathrm{JVM}} C_{i_n+2}$ is the execution of the corresponding *AFTER/EXCEPTIONAL* block.

Lemma 2 *There is a derivation of $\mathcal{A}$ of the form*

$$\begin{aligned} q_0 \quad = \quad & \mathbf{ms}^g(C_{i_0-1}) \overset{sra(C_{i_0})}{\longrightarrow} \mathbf{ms}^g(C_{i_0}) \overset{sra(C_{i_0}C_{i_0+1})}{\longrightarrow} \mathbf{ms}^g(C_{i_0+2}) = \\ & \mathbf{ms}^g(C_{i_1-1}) \overset{sra(C_{i_1})}{\longrightarrow} \cdots \overset{sra(C_{i_n})}{\longrightarrow} \mathbf{ms}^g(C_{i_n}) \overset{sra(C_{i_n}C_{i_n+1})}{\longrightarrow} \mathbf{ms}^g(C_{i_n+2}) \end{aligned}$$

where, by convention, $\overset{\varepsilon}{\longrightarrow}$ is the identity relation, q_0 is the initial state of $\mathcal{A}$, and, if $m \leq i_n + 1$ the last transition is not included.

Proof (Sketch) We ignore the details of the *BEFORE*, *AFTER*, and *EXCEPTIONAL* ghost state updates (which is really the meat of the argument). By induction on n. The base case is an easy special case of the induction step. So, consider an execution of the shape $EC_{i_n-1}C_{i_n}C_{i_n+1}C_{i_n+2}$ where C_{i_n} is calling. By the induction hypothesis there is a derivation as prescribed by the lemma ending in an automaton state $\mathbf{ms}^g(C_{i_{n-1}+2})$, and, since all calling configurations are included among the i_j, $\mathbf{ms}^g(C_{i_{n-1}+2}) = \mathbf{ms}^g(C_{i_n-1})$. By the assumed correctness of the *BEFORE* block we obtain that $\mathbf{ms}^g(C_{i_n-1}) \overset{sra(C_{i_n})}{\longrightarrow} \mathbf{ms}^g(C_{i_n})$, and similarly, by the correctness of the *AFTER* or *EXCEPTIONAL* clause, if $\mathbf{ms}^g(C_{i_n+2}) \neq \bot$, $\mathbf{ms}^g(C_{i_n+1}) \overset{sra(C_{i_n}C_{i_n+1})}{\longrightarrow} \mathbf{ms}^g(C_{i_n+2})$. This is sufficient to derive the result. $\square$

Corollary 1 *Let $E = C_0 \cdots C_{i_0} \cdots C_{i_n} \cdots C_m$ be an execution of $GI(P, \mathcal{P}_\mathcal{A})$ such that $i_0, \ldots, i_n$ are all j for which C_j is a calling configuration, of some API method c.m. If $\mathbf{ms}^g(C_{i_n}) \neq \bot$ then $srt(E) \in \mathcal{P}_\mathcal{A}$.*

Proof Use lemma 2. $\square$

10 Contract Adherence Proofs

Recall that the embedded state should be in sync with the ghost state when control of execution passes to another method (through a method call, a return statement or an exception). This can be guaranteed by letting each method ensure, require and exsure the equality of the ghost monitor state and the embedded monitor state, $\mathbf{ms}^g = \mathbf{ms}$. This assertion is from now on denoted Ψ and referred to as the *synchronization assertion*.

Accordingly, we call an extended method definition (I, H, A, req, ens, exs) *synchronizing*, if it is locally valid and $req = ens = exs = \psi$. We call a program P *synchronizing*, and say that P has *synchronizing annotations*, if all method definitions in P are synchronizing.

If an extended method definition $M = (I', H, A, \mathit{req}, \mathit{ens}, \mathit{exs})$ is synchronizing and $I' = GI(I, \mathcal{P}_\mathcal{A})$ then the method (I, H) adheres to the policy $\mathcal{P}_\mathcal{A}$. This allows us to derive verification conditions for policy adherence as follows:

Definition 5 (Verification Conditions) *Given an array of assertions A, a method $M = (I, H)$ and a policy $\mathcal{P}_\mathcal{A}$ the set of verification conditions $VC(A, M, \mathcal{P}_\mathcal{A})$ contains the following assertions:*

- $\Psi \Rightarrow A_0$
- $A_L \Rightarrow wp_{M'}(A)_L$ *for each label* $L \in I' = GI(M, \mathcal{P}_\mathcal{A})$ *where* $M' = (I', H, A, \Psi, \Psi, \Psi)$

Corollary 2 *$VC(A, M, \mathcal{P}_\mathcal{A})$ is valid if, and only if, the extended method $(GI(M, \mathcal{P}_\mathcal{A}), H, A, \psi, \psi, \psi)$ is synchronizing.*

Theorem 3 (Validity of VC Implies Contract Adherence) *Suppose there is an A such that $\bigwedge VC(A, M, \mathcal{P}_\mathcal{A})$ is valid for each method M in P. Then P adheres to the security policy $\mathcal{P}_\mathcal{A}$.*

Proof (Sketch) We assume that $\bigwedge VC(A, M, \mathcal{P}_\mathcal{A})$ is valid for each method M in P. Consider any execution $E = C_0 \cdots C_{i_0} \cdots C_{i_n} \cdots C_m$ such that $i_0 \ldots, i_n$ are all those j for which C_j is calling some API method (as above). We claim that then Ψ holds at all program points C_{i_j}. The proof of this is similar to the proof of theorem 1 and left out. By construction, $\mathbf{ms}(C_{i_j}) \neq \perp$. (The embedded inliner never assigns $\perp$ to $\mathbf{ms}$). It follows that $\mathbf{ms}^g(C_{i_j}) \neq \perp$, and the result then follows by corollary 1. $\Box$

10.1 Proof Generation

Reflecting def. 5 we view a policy adherence proof as an assignment of assertions to program points in a program with all methods ghost inlined, such that all verification conditions are valid.

Definition 6 (Adherence Proof) *An adherence proof is defined locally for each method and for the program as a whole.*

1. *A* method adherence proof *for a method definition $M_\mathcal{A} = (I, H)$ with contract $\mathcal{P}_\mathcal{A}$ is a mapping A of labels in $GI(I, \mathcal{P}_\mathcal{A})$ to assertions such that $\bigwedge VC(A, M_\mathcal{A}, \mathcal{P}_\mathcal{A})$ is valid.*
2. *An* adherence proof *for a program P assigns a method adherence proof to each method in each class of P.*

The reader may justifiably object to a concept of proof which is not in general efficiently recognizable. In this paper, however, we only generate proofs for programs that have already been inlined. This ensures that proofs are sufficiently simple that they can be both efficiently generated and efficiently recognized. Regardless:

Corollary 3 (Adherence Proof Soundness) *If there is an adherence proof for P with respect to policy $\mathcal{P}_{\mathcal{A}}$ then P adheres to $\mathcal{P}_{\mathcal{A}}$.*

Proof By theorem 3.

The annotation for a given program point is generated differently, according to whether the instruction at that program point can appear as part of an inlined block or not. Instructions inside the inlined block affect the processing of the embedded and the ghost state, method call arguments etc. For this reason they need detailed analysis using *wp*. Instructions outside the inlined blocks, on the other hand, allow a more robust treatment, as they only are required to preserve the synchronization assertion $\mathbf{ms} = \mathbf{ms}^g$ which they do, due to assumption 1.

For this to add up, an important prerequisite is that the synchronization assertion is preserved by each block of inlined and ghost inlined code. That is, if an inlined block is entered in a state satisfying the synchronization assertion then the synhronization assertion also holds on exit of that block. To make this statement precise let an inlined block at the interval $[L, L']$ be the consecutive sequence of instructions, including those generated by the ghost inliner, appearing in the code fragment on fig. 2.

Lemma 3 (Preservation of the Synchronization Assertion) *Given an inlined and ghost inlined method definition $M = (I, H)$ and an inlined block at $[L, L']$ in I, there exists an assertion array A with $|A| = |I|$ such that $\Psi \Rightarrow A_L$, $A_L \Rightarrow wp_{M'}(L)$, $A_{L+1} \Rightarrow wp_{M'}(L+1)$, ..., $A'_{L'} \Rightarrow wp_{M'}(L')$ and $A'_{L'} \Rightarrow \Psi$, where M' is the extended method definition $(I, H, A, \Psi, \Psi, \Psi)$.*

Proof The construction is shown in appendix C. □

Lemma 3 relies crucially on the property of inlined blocks that control transfers out of the inlined block is possible only when the *wp*-generated assertion is sufficient to guarantee the synchronization assertion, and vice versa for transfers inside the inlined block.

Generation of the annotations is done by the *anno* function defined as follows.

Definition 7 (Annotation Function) *Given a method definition $M = (I, H)$ and a set of labels IL of the inlined instructions in I, $anno(M, IL)$ returns an array A of assertions such that*

$$A_L = \begin{cases} narr(wp_{M'}(L)) & \textit{if } L \in IL \textit{ or } I_L \textit{ is an invoke instruction} \\ \Psi & \textit{otherwise} \end{cases}$$

where $M' = (I, H, A, \Psi, \Psi, \Psi)$ and $narr(\phi)$ returns $\phi' = \Psi \wedge a_0^g = a_0 = s_0 \wedge \ldots \wedge a_n^g = a_n = s_n$ if $\phi' \Rightarrow \phi$, otherwise ϕ.

Note that *anno* is well-defined as inlined blocks are loop-free.

Theorem 4 (Locally Valid Annotation) *Given an inlined method $M' = I(M, \mathcal{A}) = (I, H)$ and a ghost inlined method body $I' = GI(I, \mathcal{P}_\mathcal{A})$ with IL the set of labels of inlined or ghost inlined instructions in I', the method $(I', H, anno((I', H), IL), \Psi, \Psi, \Psi)$ is locally valid.*

Proof We show that the two properties in definition 3 hold. Let IL be the set of labels of inlined instructions in $M_\mathcal{C}$.

1. $\Psi \Rightarrow A_0$

   This holds since $A_0 = \Psi$ by definition of *anno* if $0 \notin IL$ and by lemma 3 otherwise.

2. $A_L \Rightarrow wp_M(L)$ for all labels $L \in B$

   If $L \in IL$ or B_L is an invoke instruction we have, by definition of *anno*, $A_L \Rightarrow wp_M(L)$.

   Otherwise we have, by definition of *anno*, $A_L = \Psi$. We also know that $wp_M(L) = \Psi$ (by assumption 1 and 3), thus we conclude that $A_L \Rightarrow wp_M(L)$ for this case as well.

Theorem 4 allows us to prove a partial converse to theorem 3, namely that verification conditions generated from properly inlined methods are valid.

Corollary 4 *Suppose $M = I(M', \mathcal{P}_\mathcal{A})$. Then there is an A such that $\bigwedge VC(A, M, \mathcal{P}_\mathcal{A})$ is valid.* □

Putting it all together:

Theorem 5 (Proof Generation) *For each program P and policy $\mathcal{P}_\mathcal{A}$ there is an algorithm, linear in $|P| + |\mathcal{P}_\mathcal{A}|$, which produces an adherence proof of $I(P, \mathcal{P}_\mathcal{A})$ with contract $\mathcal{P}_\mathcal{A}$.*

Proof Given theorem 4 the only property left to show is the complexity bound which is verified by inspection. □

10.2 Proof Recognition

To verify that a given mapping of labels to assertions indeed is a proof, the receiver has to verify that the resulting verification conditions hold. This task is in general undecidable, however in our settings the verification conditions are simple and predictable enough to be efficiently provable.

For the proof recognition statement to follow we consider TM recognizers of "adherence proof candidates", i.e. TM's which take type and arity correct assignments of assertions to program points for each method in a given target program, as input, and produce a binary accept/reject result.

Theorem 6 (Efficient Recognition) *The class of linear-time recognizable adherence proofs includes all adherence proofs generated from inlined programs using the algorithm of theorem 5.*

Proof It suffices to show that each member of a set

$$VC(anno(GI(I(M,\mathcal{P}_{\mathcal{A}}),\mathcal{P}_{\mathcal{A}})),I(M,\mathcal{P}_{\mathcal{A}}),\mathcal{P}_{\mathcal{A}})$$

can be reduced to `TRUE` in time linear in $|M|$.

- $\Psi \Rightarrow A_0$

  This is trivial since $A_0 = \Psi$. (See proof of theorem 4, item 1.)

- $A_L \Rightarrow wp_M(L)$ for each label L in B

  If $L \in IL$ or I_L is an invoke instruction we have $A_L = narr(wp_M(L))$ as defined in definition 7. A_L then either equals $wp_M(L)$ (in which case it is trivial) or it is on the form $\Psi \wedge a_0^g = a_0 = s_0 \wedge \ldots \wedge a_n^g = a_n = s_n$. The later form is chosen at points where the weakest precondition is

$$\begin{aligned}
&\text{SELECT}((t : c^n \wedge t^g : c^n, \ldots, t : c^1 \wedge t^g : c^1),\\
&\qquad (\text{SELECT}((c^n.m_{G_1} \wedge c^n.m^g_{G_1}, \ldots, c^n.m_{G_i} \wedge c^n.m^g_{G_i}),\\
&\qquad\qquad (c^n.m_{f_1}(\mathbf{ms},\mathbf{a}) = c^n.m^g_{f_1}(\mathbf{ms}^g,\mathbf{a}^g), \ldots,\\
&\qquad\qquad c^n.m_{f_i}(\mathbf{ms},\mathbf{a}) = c^n.m^g_{f_i}(\mathbf{ms}^g,\mathbf{a}^g)), \texttt{TRUE}),\\
&\qquad\quad \vdots \qquad\qquad \vdots\\
&\qquad \text{SELECT}((c^1.m_{G_1} \wedge c^1.m^g_{G_1}, \ldots, c^1.m_{G_j} \wedge c^1.m^g_{G_j}),\\
&\qquad\qquad c^1.m_{f_1}(\mathbf{ms},\mathbf{a}) = c^1.m^g_{f_1}(\mathbf{ms}^g,\mathbf{a}^g), \ldots,\\
&\qquad\qquad c^1.m_{f_j}(\mathbf{ms},\mathbf{a}) = c^1.m^g_{f_j}(\mathbf{ms}^g,\mathbf{a}^g)), \texttt{TRUE})),\\
&\qquad \mathbf{ms} = \mathbf{ms}^g)
\end{aligned}$$

  (See proof of lemma 3.) The verification condition can then be rewritten and simplified by iterated applications of the rule $x = y \Rightarrow \phi \longrightarrow \phi[z/x][z/y]$ where x and y are instantiated with real variables and ghost counterparts respectively and where z does not occur in ϕ. These rewrites takes time proportional to the product of the number of variables and the length of the formula and does not change the size of the expression since x, y and z are atomic. The result can then be rewritten to `TRUE` using the rules $(\psi \Rightarrow \phi) \wedge (\neg\psi \Rightarrow \phi) \longrightarrow \phi$ and $\phi = \phi \longrightarrow$ `TRUE` in time linearly proportional to the formula.

Tyhe upshot is that we can use a linear-time algorithm to recognize adherence proofs, and in so doing we are guaranteed to recognize at least those proofs generated by ourselves from correctly inlined input.

11 System Architecture

Our architecture resembles the one by Colby, Lee and Necula [6] except that we focus on verifying that a proper IRM is present. An overview of our system is show in figure 4.

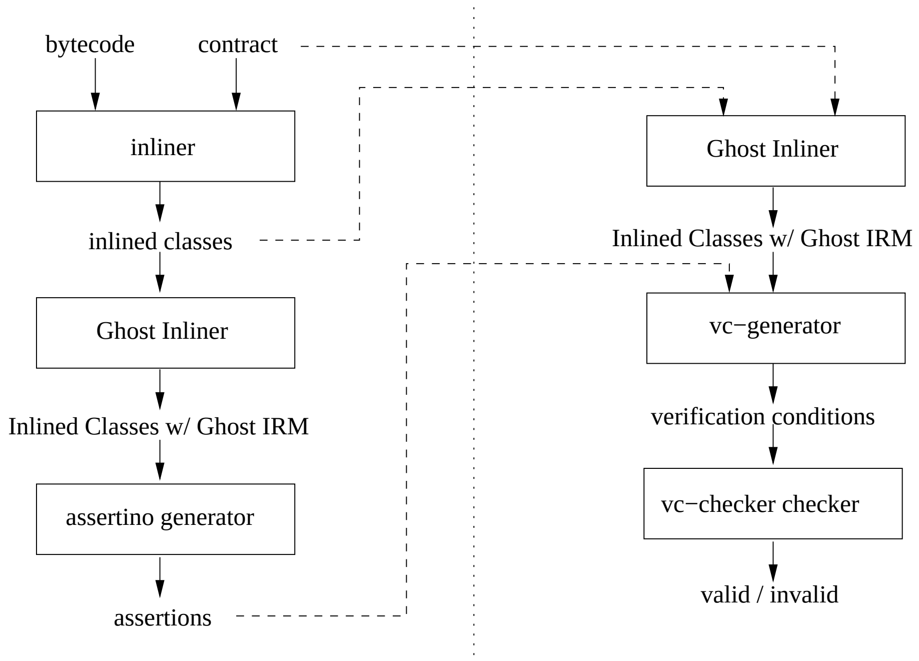


Figure 4: The architecture of our PCC implementation.

The developer side of the system runs in Java SE and the user side in Java ME. Both parts utilizes a parser generated by CUP/JFlex [17, 18] and the ASM library [7] for handling class files.

Inliner Our inliner is similar to the JVML SASI inliner by Erlingsson and Schneider [12] and their successor implementation in PoET [10]. As input it takes a Java bytecode program and a CosSpec policy. It rewrites the program so that it complies with the given policy. Depending on the original behavior of the program the inserted code may or may not be redundant. In either case it simplifies the proof generation process significantly.

To keep track of the security state during execution the inliner creates a set of public static fields in an final auxiliary class called `SecState` which has no methods. The types and identifiers of the fields correspond to those of the variables declared in the `SECURITY STATE` section in the policy. Code that initializes these fields is compiled and added as a static initializer of the `SecState` class.

The inliner proceeds by locating and replacing all invoke instructions with code that comprises checks and updates for all relevant event clauses as described in section 8. If an invoke instruction calls $c.m$ an event clause concerning method $c'.m'$ is relevant iff $m = m'$ and $c = c'$ or $c <: c'$, $c' <: c$.

Finally, for each method an `IRMOffsetAttribute` is embedded which specifies at what offsets the inlined instructions can be found. This information is needed for the proof generation.

Ghost Inliner The ghost inliner is an implementation of the *GI* function described in section 9.2. It parses the class files and builds an object model corresponding to the structure of the method. Ghost instructions are then generated from the policy and inserted into this model.

Assertion Generator The assertion generator is an implementation of the *anno* function. The list of offsets for the inlined instructions is recovered from the `IRMOffsetAttribute`. The generator then initializes all assertions of the non-inlined instructions to Ψ and proceeds by annotates the remaining instructions by computing their weakest preconditions bottom up.

The `IRMOffsetAttribute` is discarded and an `AssertionsAttribute` is embedded, containing a serialization of the resulting array of assertions.

11.1 Complexity

The execution time and memory foot print is linear in the size of the input. The inliner and proof generator is implemented with the visitor pattern and could, if preferred, be executed chained together in a single pass over the bytecode.

12 System Demonstration

To demonstrate the system we have written a security specification and chosen a program which potentially violates this specification. We inline the program with the security specification and then show how a proof is generated and verified.

12.1 A Privacy Policy and a Game

The policy prohibits executions which sends data on the network after accessing the memory of the device (see figure 5). The program we have chosen is a “snake” game featuring loading of a highscore list from phone memory and submitting current score to a server (see figure 6). These two functions makes it possible for the user to violate the policy.

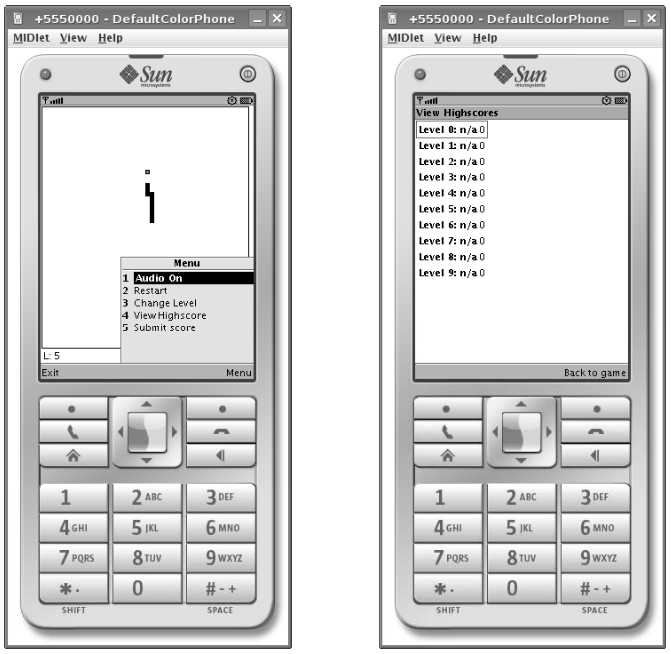
```
SCOPE Session

SECURITY STATE boolean haveRead = false;

BEFORE javax.microedition.rms.RecordStore.openRecordStore(
         string name, boolean createIfNecessary)
    PERFORM
         true -> { haveRead = true;  }

BEFORE javax.microedition.io.Connector
         .openDataOutputStream(string url)
    PERFORM
         haveRead == false -> { }
```

Figure 5: A ConSpec specification which disallows the program from sending data over the network after accessing phone memory.



```
Level 0: n/a 0
Level 1: n/a 0
Level 2: n/a 0
Level 3: n/a 0
Level 4: n/a 0
Level 5: n/a 0
Level 6: n/a 0
Level 7: n/a 0
Level 8: n/a 0
Level 9: n/a 0
```

Figure 6: Screenshot of the Snake game.

12.2 Developers perspective

From a developers perspective, the only difference lies in the deployment process. Between compilation and preverification the inliner is executed and between preverification and jar-packaging the proof generater is executed. Figure 7 illustrates the PCC deployment process.

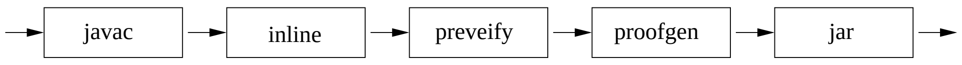


Figure 7: PCC deploy process.

The inliner and proof generator is executed from the CLI in the following way:

```
> java -jar Inliner.jar  -basic             \
                         -lib midpapi20.jar  \
                         -lib cldcapi11.jar  \
                         -policy policy.txt  \
                         -target *.class

> java -jar ProofGen policy.txt inlined/*.class
```

Both components could easily be integrated in the compile chain in any modern IDE such as Eclipse.

It turns out that the inliner modifies the target program in two places. The modification and the generated assertions for `openRecordStore` and `openDataOutputStream` are illustrated in figure 8 and 9 respectively.

12.3 Users Perspective

The system should, to an as large extent as possible, be transparent to the user and thus only intervene in case the proof checker failes to verify a given proof. See figure 10.

13 Conclusions

We have demonstrated the feasibility of a proof-carrying approach to certified monitor inlining at the level of practical Java bytecode, including exceptions and inheritance. This answers a question raised in [16]. We have proved correctness of our approach in the sense of soundness: Policy adherence proofs are sufficient to ensure compliance, and we also obtain partial completeness results, namely that proofs for inlined programs can always be generated, and such proofs are guaranteed to be recognized at program loading time. Other properties are also interesting such as transparency: Roughly, that all adherent behaviour is allowed by the inliner. This type of property is, however, more relevant for the specific inliner, and not so much for the certification mechanism, and consequently not addressed here (but see e.g. [22, 9] for results in this direction). The approach is efficient:

```
                  ⋮
          2:   {Ψ}
               ldc "HighScores"
          4:   {Ψ}
               iconst_1
        ⎧ 5:   {TRUE = 1}
        ⎪      istore_3
        ⎪ 6:   {TRUE = 1}
        ⎪      astore_2
        ⎪ 7:   {TRUE = 1}
        ⎪      iconst_1
inlined ⎨ 8:   {TRUE = s0}
        ⎪      putstatic SecState.haveRead
        ⎪11:   {TRUE = SecState.haveRead}
        ⎪      aload_2
        ⎪12:   {TRUE = SecState.haveRead}
        ⎪      iload_3
        ⎪13:   {TRUE = SecState.haveRead}
        ⎩      ⟨haveRead^g := TRUE → TRUE⟩
               {Ψ}
               invokestatic RecordStore.openRecordStore
         16:   {Ψ}
               astore_0
                  ⋮
```

Figure 8: Generated assertions for inlining of `RecordStore.openRecord-Store`

Proofs are small and recognised easily, by a simple proof checker (which is the only required extension to the trusted computing base). An interesting feature of our approach is that detailed modelling of bytecode instructions is needed only for instructions appearing in the inlined code snippets. For other instructions a simple conditional invariance property on static fields of final objects suffices. This means, in particular, that our approach adapts to new versions of the Java virtual machine very easily, needing only a check that the static field invariance is maintained.

Two issues in particular are left for further investigation:

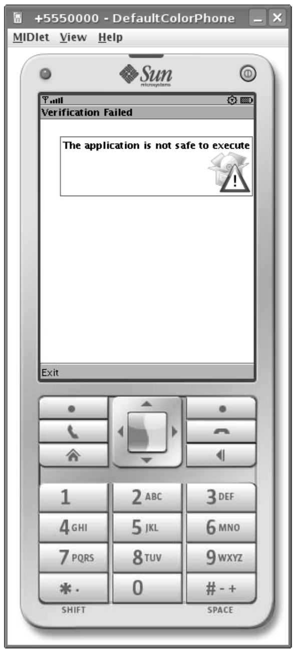
- The policy language lacks the ability to store object references. This, we believe, is crucial for the long-term practical utility of the approach, as it is needed in general, for instance to correlate object use events with object creation events. To keep the complexity manageable we have, however, left this issue for future work.

- Threads are not considered (but see [16] for a type-based approach). Threads add interesting complications at all levels of the present work (policy, inlining, proof generation, proof recognition). An investigation

```
                ⋮
          40:   {$\Psi$}
                aload_1
        ⎧ 41:   {IF($0 \neq$ SecState.haveRead, TRUE,
        ⎪        IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead))}
        ⎪       astore_3
        ⎪ 42:   {IF($0 \neq$ SecState.haveRead, TRUE,
        ⎪        IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead))}
        ⎪       getstatic SecState.haveRead
        ⎪ 45:   {IF($0 \neq s_0$, TRUE,
        ⎪        IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead))}
        ⎪       iconst_0
        ⎪ 46:   {IF($s_0 \neq s_1$, TRUE,
        ⎪        IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead))}
        ⎪       if_icmpne 52
        ⎪ 49:   {IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead)}
        ⎪       goto 70
        ⎪ 52:   {TRUE}
        ⎪       getstatic System.err
inlined ⎨ 55:   {TRUE}
        ⎪       dup
        ⎪ 56:   {TRUE}
        ⎪       ldc "Program terminated!"
        ⎪ 58:   {TRUE}
        ⎪       invokevirtual PrintStream.println
        ⎪ 61:   {TRUE}
        ⎪       ldc "BEFORE openDataOutputStream violated."
        ⎪ 63:   {TRUE}
        ⎪       invokevirtual PrintStream.println
        ⎪ 66:   {TRUE}
        ⎪       iconst_m1
        ⎪ 67:   {TRUE}
        ⎪       invokestatic System.exit
        ⎪ 70:   {IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead)}
        ⎪       aload_3
        ⎪       {IF($haveRead^g =$ FALSE, $\Psi$, $\bot =$ SecState.haveRead)}
        ⎩       $\langle haveRead^g := haveRead^g =$ FALSE $\rightarrow haveRead^g \rangle$
          71:   {$\Psi$}
                invokestatic Connector.openDataOutputStream
          74:   {$\Psi$}
                astore_2
                ⋮
```

Figure 9: Generated assertions for inlining of `Connector.openDataOutput-Stream`

to address this is currently going on.



The application is not safe to execute

Figure 10: The proof could not be verified.

References

[1] S^{3}MS web page. http://www.s3ms.org.

[2] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. http://www.csc.kth.se/∼irem/S3MS/TechRep07.pdf, November 2007.

[3] I. Aktug, M. Dam, and D. Gurov. Provably correct runtime monitoring. In *To appear in proc. of 15th Int. Symposium on Formal Methods (FM '08)*, May 2008.

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

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

[6] Christopher Colby, Peter Lee, and George C. Necula. A proof-carrying code architecture for java. In *Computer Aided Verification*, pages 557–560, 2000.

[7] ObjectWeb Consortium. Asm - home page. Accessed 2008-03-05, February 2008.

[8] Robert DeLine and Manuel Fähndrich. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 59–69, New York, NY, USA, 2001. ACM.

[9] Frank Piessens Dries Vanoverberghe. Security enforcement aware software development. *To be published in the Special issue on Software Engineering for Secure Systems, journal on Information and Software Technology*, 2008.

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

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

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

[13] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. *SIGPLAN Not.*, 37(5):1–12, 2002.

[14] S. N. Freund and J. C. Mitchell. A type system for object initialization in the Java bytecode language. *ACM Trans. Program. Lang. Syst.*, 21(6):1196–1250, 1999.

[15] Yoonsik Cheon Gary T. Leavens. Design by contract with jml. http://www.eecs.ucf.edu/ leavens/JML/jmldbc.pdf, September 2006.

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

[17] Scott Hudson. Cup. Accessed 2008-03-05, October 2003.

[18] Gerwin Klein. Jflex. Accessed 2008-03-05, October 2007.

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

[20] X. Leroy. Java bytecode verification: algorithms and formalizations, 2003.

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

[22] J. A. Ligatti. *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University, June 2006.

[23] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 333–344, New York, NY, USA, 1998. ACM.

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

[25] J. Saltzer and M. Schroeder. The protection of information in computer systems. *IEEE, Vol. 9(63)*, 1975.

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

[27] Christian Skalka and Scott Smith. History effects and verification. In *Asian Programming Languages Symposium*, November 2004.

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

[29] Frank Yellin. Low level security in java. http://java.sun.com/sfaq/-verifier.html, 1996.

Appendix A. JVM Instruction Semantics

In this appendix we present a transition-style semantics of JVM instructions used by the inliner. The semantics applies only to configurations that are type safe and have passed the Java bytecode verifier, cf. [20, 29].

Notation

The transition rules use the following auxiallary operations:

- $\nu l.-$ is a new location binder: it binds l to a location which is new, i.e. not already allocated, in the argument.
- $\mathit{upd}(r, n, l)$ is the update of the local variables obtained by replacing the n'th register by l.
- (pc, l, h) *matches* (b, e, pc', c) if, and only if, $pc \in [b, e)$ and $h \vdash l : c$.
- $\mathit{clr}(s)$ clear the operand stack.

Transition Rules

$$\frac{M[pc] = \texttt{ALOAD}\ n}{((M, pc, s, r) :: R, h) \to ((M, pc+1, r[n] :: s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{ASTORE}\ n}{((M, pc, l :: s, r) :: R, h) \to ((M, pc+1, s, \mathit{upd}(r, n, l)) :: R, h)}$$

$$\frac{M[pc] = \texttt{ATHROW}}{((M, pc, l :: s, r) :: R, h) \to ((l) :: (M, pc, l :: s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{DUP}}{((M, pc, v :: s, r) :: R, h) \to ((M, pc+1, v :: v :: s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{GETSTATIC}\ c.f}{((M, pc, s, r) :: R, h) \to ((M, pc+1, h(c.f) :: s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{GETFIELD}\ f}{((M, pc, o :: s, r) :: R, h) \to ((M, pc+1, h(o).f :: s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{GOTO}\ L}{((M, pc, s, r) :: R, h) \to ((M, pc+L, s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{ICONST\_}n}{((M, pc, s, r) :: R, h) \to ((M, pc+1, n :: s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{IFEQ}\ L \quad n = 0}{((M, pc, n :: s) :: R, h) \to ((M, pc+L, s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{IFEQ}\ L \quad n \neq 0}{((M, pc, n :: s, r) :: R, h) \to ((M, L, s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{IF\_ICMPNEQ}\ L \quad n = m}{((M, pc, n :: m :: s, r) :: R, h) \rightarrow ((M, L, s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{IFICMPEQ}\ L \quad n \neq m}{((M, pc, n :: m :: s, r) :: R, h) \rightarrow ((M, pc + 1, s, r) :: R, h)}$$

$$\frac{M[pc] = \texttt{PUTSTATIC}\ c.f}{((M, pc, v :: s, r) :: R, h) \rightarrow ((M, pc + 1, s, r) :: R, h[(c.f) \mapsto v])}$$

Exceptions:

$$\frac{\begin{array}{c} M = (P, H) \quad (pc, l, h)\ \mathit{matches}\ H(i) \quad H(i) = (b, e, pc', c) \\ \forall j > i : \neg((pc, l, h)\ \mathit{matches}\ H(j)) \end{array}}{((l) :: (M, pc, s, r) :: R, h) \rightarrow ((M, pc', l :: \mathit{clr}(s), r) :: R, h)}$$

$$\frac{M = (P, H) \quad \forall j > 0 : \neg((pc, l, h)\ \mathit{matches}\ H(j)}{((l) :: (M, pc, s, r) :: R, h) \rightarrow ((l) :: R, h}$$

Appendix B. Definition of Inliner

Here follows a general security specification concerning a method m : `int` $\rightarrow$ `int` defined in class c and overridden in a subclass d. The policy defines event clauses for BEFORE, AFTER and EXCEPTIONAL cases for each definition of m. Each event clause has two guards and two statement lists.

The security specification and the following section of inlined code can easily be generalized to handle an arbitrary number of event clauses, arbitrary number of guards and methods of arbitrary arity.

```
SCOPE Session
SECURITY STATE DECLARATION

BEFORE         c.m(int a) PERFORM cb$_{g_1}$ -> {cb$_{s_1}$} | cb$_{g_2}$ -> {cb$_{s_2}$}
AFTER      r = c.m(int a) PERFORM ca$_{g_1}$ -> {ca$_{s_1}$} | ca$_{g_2}$ -> {ca$_{s_2}$}
EXCEPTIONAL c.m(int a) PERFORM ce$_{g_1}$ -> {ce$_{s_1}$} | ce$_{g_2}$ -> {ce$_{s_2}$}

BEFORE         d.m(int a) PERFORM db$_{g_1}$ -> {db$_{s_1}$} | db$_{g_2}$ -> {db$_{s_2}$}
AFTER      r = d.m(int a) PERFORM da$_{g_1}$ -> {da$_{s_1}$} | da$_{g_2}$ -> {da$_{s_2}$}
EXCEPTIONAL d.m(int a) PERFORM de$_{g_1}$ -> {de$_{s_1}$} | de$_{g_2}$ -> {de$_{s_2}$}
```

```
    storeArgs: ASTORE $r_a$
               ASTORE $r_t$
               ALOAD $r_t$
               ALOAD $r_a$

      dbCheck: ALOAD $r_t$
               INSTANCEOF d
               IFEQ cbCheck

     dbGuard1: [EVALUATE $db_{g_1}$]
               IFEQ dbGuard2
               [EXECUTE $db_{s_1}$]
               GOTO BEnd

     dbGuard2: [EVALUATE $db_{g_2}$]
               IFEQ dBFail
               [EXECUTE $db_{s_2}$]
               GOTO BEnd

       dBFail: ICONST_1
               INVOKESTATIC System.exit

      cbCheck: ALOAD $r_t$
               INSTANCEOF c
               IFEQ BEnd

     cbGuard1: [EVALUATE $cb_{g_1}$]
               IFEQ cbGuard2
               [EXECUTE $cb_{s_1}$]
               GOTO BEnd

     cbGuard2: [EVALUATE $cb_{g_2}$]
               IFEQ cbFail
               [EXECUTE $cb_{s_2}$]
               GOTO BEnd

       cbFail: ICONST_1
               INVOKESTATIC System.exit

         BEnd: INVOKEVIRTUAL c.m

               GOTO handlerEnd

 handlerStart: ALOAD $r_t$
               INSTANCEOF d
               IFEQ ceCheck

     deGuard1: [EVALUATE $de_{g_1}$]
               IFEQ deGuard2
               [EXECUTE $de_{s_1}$]
               GOTO EEnd

     deGuard2: [EVALUATE $de_{g_2}$]
               IFEQ deFail
               [EXECUTE $de_{s_2}$]
               GOTO EEnd

       deFail: ICONST_1
               INVOKESTATIC System.exit

      ceCheck: ALOAD $r_t$
               INSTANCEOF c
               IFEQ EEnd

     ceGuard1: [EVALUATE $ce_{g_1}$]
               IFEQ ceGuard2
               [EXECUTE $ce_{s_1}$]
               GOTO EEnd

     ceGuard2: [EVALUATE $ce_{g_2}$]
               IFEQ ceFail
               [EXECUTE $ce_{s_2}$]
               GOTO EEnd

       ceFail: ICONST_1
               INVOKESTATIC System.exit

         EEnd: ATHROW

   handlerEnd: ALOAD $r_t$
               INSTANCEOF d
               IFEQ caCheck

     daGuard1: [EVALUATE $da_{g_1}$]
               IFEQ daGuard2
               [EXECUTE $da_{s_1}$]
               GOTO AEnd

     daGuard2: [EVALUATE $da_{g_2}$]
               IFEQ daFail
               [EXECUTE $da_{s_2}$]
               GOTO AEnd

       daFail: ICONST_1
               INVOKESTATIC System.exit

      caCheck: ALOAD $r_t$
               INSTANCEOF c
               IFEQ AEnd

     caGuard1: [EVALUATE $ca_{g_1}$]
               IFEQ caGuard2
               [EXECUTE $ca_{s_1}$]
               GOTO AEnd

     caGuard2: [EVALUATE $ca_{g_2}$]
               IFEQ caFail
               [EXECUTE $ca_{s_2}$]
               GOTO AEnd

       caFail: ICONST_1
               INVOKESTATIC System.exit

         AEnd:
```

Each ***[EVALUATE g]*** section transforms the a configuration $((M, pc, s, r) :: R, h)$ to $((M, pc', v :: s, r) :: R, h)$ where v is 0 or 1 if the guard g is false or true respectively. An ***[EXECUTE stmts]*** transforms the a configuration $((M, pc, s, r) :: R, h)$ to $((M, pc', s, r) :: R, h[\mathit{collapse}(\mathit{stmts}, \mathbf{ms})/\mathbf{ms}])$.

Appendix C. Proof of lemma 3

This proof shows that lemma 3 holds for an example that can easily be generalized. Just as in appendix B we concider a general policy for a method $m : \texttt{int} \rightarrow \texttt{int}$. In this example this method is defined by one class c and $c.m$ has a before, after and exceptional event clause with one guard each. The policy is shown below.

```
SCOPE Session
SECURITY STATE DECLARATION

BEFORE       c.m(int a) PERFORM cb_g -> {cb_s}
AFTER   r = c.m(int a) PERFORM ca_g -> {ca_s}
EXCEPTIONAL c.m(int a) PERFORM ce_g -> {ce_s}
```

We let $c.m_f^{bf}(\mathbf{ms})$ denote $collapse(cb_s, \mathbf{ms})$ and treat the after and exceptional cases similarly. Furthermore we assume that the first entry in the exception table is $(30, 32, 34, any)$.

```
     $\mathbf{ms} = \mathbf{ms}^g$
     NON-INLINED INSTRUCTION

     // INLINED CODE START

     $\mathbf{ms} = \mathbf{ms}^g$
     ASTORE a
     ASTORE t
     ALOAD t
     ALOAD a

     // BEFORE

26:  IF$(t : c, A_{28}, A_{30})$
     ALOAD t
     INSTANCEOF c
     IFEQ 30

28:  IF$(c.m_G^{bf\,g},$ IF$(s_1 : c,$ IF$(c.m_G^{bf} \wedge c.m_G^{bf\,g}, c.m_f^{bf}(\mathbf{ms}) = c.m_f^{bf\,g}(\mathbf{ms}^g, s_0), c.m_f^{bf}(\mathbf{ms}) = \perp),$
         $\mathbf{ms} = \mathbf{ms}^g) \wedge a = s_0 \wedge t = s_1,$ TRUE$)$
     [EVALUATE c BEFORE GUARD]
     IFEQ 29
     [PERFORM c BEFORE ACTION]
     GOTO 30

29:  TRUE
     ICONST_1
     INVOKESTATIC System.exit

30:  IF$(s_1 : c,$ IF$(c.m_G^{bf\,g}, \mathbf{ms} = c.m_f^{bf\,g}(\mathbf{ms}^g, s_0), \mathbf{ms} = \perp), \mathbf{ms} = \mathbf{ms}^g) \wedge a = s_0 \wedge t = s_1$
     $\langle (t^g, a^g) := (s_1, s_0) \rangle$
     $\langle \mathbf{ms}^g := t^g : c \to (c.m_G^{bf\,g} \to c.m_f^{bf\,g}(\mathbf{ms}^g, a^g)) \mid$ TRUE $\to \mathbf{ms}^g \rangle$

     $\mathbf{ms} = \mathbf{ms}^g \wedge a = a^g \wedge t = t^g$
     INVOKEVIRTUAL c.m(int) :  int

32:  $\mathbf{ms} = \mathbf{ms}^g \wedge a = a^g \wedge t = t^g$
     $\langle r^g := s_0 \rangle$
     $\langle \mathbf{ms}^g := t^g : c \to (c.m_G^{af} \to c.m_f^{af}(\mathbf{ms}^g, r^g, a^g)) \mid$ TRUE $\to \mathbf{ms}^g \rangle$

     IF$(t : c,$ IF$(c.m_G^{af}, c.m_f^{af}(\mathbf{ms}, s_0, a) = \mathbf{ms}^g,$ TRUE$), \mathbf{ms} = \mathbf{ms}^g)$
     ASTORE r
     ALOAD r

     $A_{43}$
     GOTO 43

     // EXCEPTIONAL

34:  $\mathbf{ms} = \mathbf{ms}^g \wedge a = a^g \wedge t = t^g$
     $\langle \mathbf{ms}^g := t^g : c \to (c.m_G^{ex\,g} \to c.m_f^{ex\,g}(\mathbf{ms}^g, a^g)) \mid$ TRUE $\to \mathbf{ms}^g \rangle$

38:  IF$(t : c, A_{40}, A_{42})$
     ALOAD t
     INSTANCEOF c
     IFEQ 42

40:  IF$(c.m_G^{ex}, c.m_f^{ex}(\mathbf{ms}, a) = \mathbf{ms}^g,$ TRUE$)$
     [EVALUATE c EXCEPTIONAL GUARD]
     IFEQ 41
     [PERFORM c EXCEPTIONAL ACTION]
     GOTO 42

41:  TRUE
     ICONST_1
     INVOKESTATIC System.exit

42:  $\mathbf{ms} = \mathbf{ms}^g$
     ATHROW

     // AFTER

43:  IF$(t : c, A_{44}, A_{46})$
     ALOAD t
     INSTANCEOF c
     IFEQ 46

44:  IF$(c.m_G^{af}, c.m_f^{af}(\mathbf{ms}, r, a) = \mathbf{ms}^g,$ TRUE$)$
     [EVALUATE c AFTER GUARD]
     IFEQ 45
     [PERFORM c AFTER ACTION]
     GOTO 46

45:  TRUE
     ICONST_1
     INVOKESTATIC System.exit

     // INLINING END

46:  $\mathbf{ms} = \mathbf{ms}^g$
     NON-INLINED INSTRUCTION
```