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

Mads Dam, Andreas Lundblad
Royal Institute of Technology, KTH

December 15, 2010

Abstract

We propose a light-weight approach for certification of monitor inlining for sequential Java bytecode using proof-carrying code. The goal is to enable the use of monitoring for quality assurance at development time, while minimizing the need for post-shipping code rewrites as well as changes to the end-host TCB. Standard automaton-based security policies express constraints on allowed API call/return sequences. Proofs are represented as JML-style program annotations. This is adequate in our case as all proofs generated in our framework are recognized in time polynomial in the size of the program. Policy adherence is proved by comparing the transitions of an inlined monitor with those of a trusted "ghost" monitor represented using JML-style annotations. At time of receiving a program with proof annotations, it is sufficient for the receiver to plug in its own trusted ghost monitor and check the resulting verification conditions, to verify that inlining has been performed correctly, of the correct policy. We have proved correctness of the approach at the Java bytecode level and formalized the proof of soundness in Coq. An implementation, including an application loader running on a mobile device, is available, and we conclude by giving benchmarks for two sample applications.

1 Introduction

Program monitoring [23, 20, 8] is a well-established technique for software quality assurance, used for a wide range of purposes such as performance monitoring, protocol compliance checking, access control, and general security policy enforcement. The conceptual model is simple: Monitorable events by a client program are intercepted and routed to a decision point where the appropriate action can be taken, depending on policy state such as access control lists, or on application history. This basic setup can be implemented in a huge variety of ways. In this paper our focus is monitor inlining [15]. In this approach, monitor functionality is weaved into client code in AOP style, with three main benefits:

- Extensions to the TCB needed for managing execution of the client, intercepting and routing events, and policy decision and enforcement are to a large extent eliminated.
- Overhead for marshalling and demarshalling policy information between the various decision and enforcement points in the system is eliminated.
- Moreover, there is no need to modify and maintain a custom API or Virtual Machine.

This, however, presupposes that the user can trust that inlining has been performed correctly. This is not a problem if the inliner is known to be correct, and if inlining is performed within the users jurisdiction. But it could of interest to make inlining available as a quality assurance tool to third parties (such as developers or operators) as well. In this paper we examine if proof-carrying code can be used to this effect in the context of Java and mobile applications, to enable richer, history-dependent access

control than what is allowed by the current, static sandboxing regime.

Our approach is as follows: We assume that J2ME applications are equipped with *contracts* that express the provider commitments on allowed sequences of API calls performed by the application. Contracts are given as security automata in the style of Schneider [30] in a simple contract specification language ConSpec [2]. The contract is compiled into bytecode and inlined into the application code as in PoET/PSLang [14], and a proof is generated asserting that the inlined program adheres to the contract, producing in the end a self-certifying code "bundle" consisting of the application code, the contract, and an embedded proof object.

Upon reception the remote device first determines whether the received bundle should be accepted for execution, by comparing the received contract with the device policy. This test uses a simulation or language containment test, and is explored in detail by K. Naliuka et al. [7].

The contribution of this paper is the efficient representation, generation, and checking of proof objects. The key idea is to compare the effects of the inlined, untrusted, monitor with a "ghost" monitor which implements the intended contract. A ghost monitor is a virtual monitor which is never actually executed, and which is represented using program annotations. Such a ghost monitor is readily available by simply interpreting the statements of the ConSpec contract as monitor updates performed before and after security relevant method calls. No JVM compilation is required at this point, since these updates are present solely for proof verification purposes.

The states of the two monitors are compared statically through a *monitor invariant*, expressing that the state of the embedded monitor is in synchrony with that of the ghost monitor. This monitor invariant is then inserted as an assertion at each security relevant method call. The assertions for the remaining program points could then in principle be computed using a weakest pre-condition (WP) calculus. Unfortunately, there is no guarantee that such an approach would be feasible. However, it turns out that it is sufficient to perform the WP computations 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 in and out of these contiguous code snippets are allowed only when the monitor invariant is guaranteed to hold.
- 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.

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 instructions that do not appear in the inlined snippets, and do not include `putstatic` instructions to the security state field, may be annotated with the monitor invariant to obtain a fully annotated program. This means, in particular, that a simple syntactic check is sufficient to eliminate costly WP checks in almost all cases and allows a very open-ended treatment of the JVM instruction set.

The resulting annotations are locally valid in the sense that method pre- and post-conditions match, and that each program point annotation follows from successor point annotations by elementary reasoning. This allows to robustly and efficiently generate and check assertions using a standard verification condition (VC) approach, as indicated in Figure 1.

Our approach is general enough to handle a wide range of inliners. The developer (who has a better insight in the application in question) is free to tweak the inlining process for his specific application and could for instance optimize for speed in certain security relevant call sites, and for code size elsewhere.

1.1 Related Work

Our approach adopts the Security-by-Contract (SxC) paradigm (cf. [7, 25, 12, 20, 8]) which has been explored and developed mainly within the S^{3}MS project [28].

Monitor inlining has been considered by a number of authors, cf. [15, 14, 13, 1, 34].

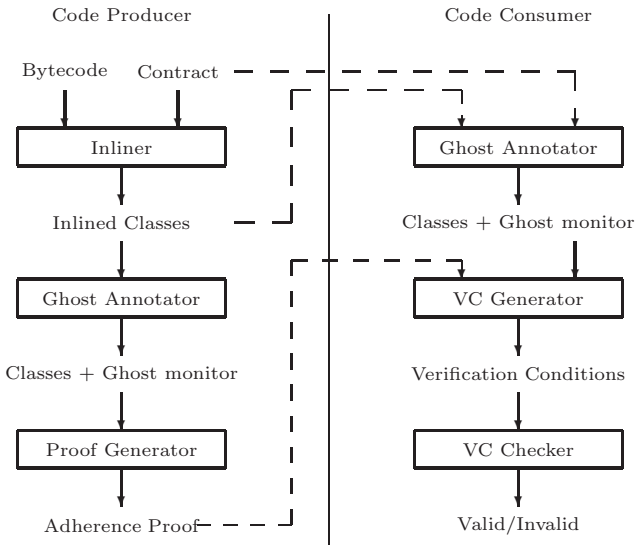

Figure 1: The architecture of our PCC implementation.

Erlingsson and Schneider [14] represents security automata directly as Java code snippets, making the resulting code difficult to reason about. The ConSpec contract specification language used here is for tractability restricted to API calls and (normal or exceptional) returns, and uses an independent expression syntax. This corresponds roughly to the call/return fragment of PSLang which includes all policies expressible using Java stack inspection [15].

Edit automata [22, 23] 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. This approach has been fully implemented for Java by J. Ligatti et al. in the Polymer tool [5] which is closely related to Naccio [16] and PoET/PSLang [14].

Certified reference monitors has been explored by a number of authors, mainly through type systems, e.g. in [31, 6, 35, 18, 11], but more recently also through model checking and abstract interpretation [33, 32]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al. [18]. 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. 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, as mentioned in the previous section, is less sensitive to this.

In [33, 32] Sridhar et al. explores the idea of certifying inlined reference monitors for ActionScript using model-checking and abstract interpretations. The approach is not tied to a specific inlining strategy and is general enough to handle different inlining techniques including non-trivial optimizations of inlined code. Although the certification process is efficient, the analysis however, has to be carried out by the consumer.

For background on proof-carrying code we refer to [26]. Our approach is based on simple Floyd-like program point annotations in the style of Bannwarth and Müller [3], and method specifications extended by pre- and post-conditions in the style of JML [17]. Recent work related to proof-carrying code for the JVM include [4], all of which has been developed in the scope of the Mobius project.

An alternative to inline reference monitoring and proof-carrying code, is to produce binaries that are structurally simple enough for the consumer to analyze himself. This is currently explored by B. Chen et al. in the Native Client project [36] which handles untrusted x86 native code. This is done through a customized compile chain that targets a subset of the x86 instruction set, which in effect puts the application in a sandbox. When applicable it has a few advantages in terms of runtime overhead, as it eliminates the monitoring altogether, but is constrained in terms of application and policy complexity.

1.2 Overview of the Paper

The JVM machine model is presented in Section 2. In Section 3 the state assertion language is introduced, and in Section 4 we address method and program annotations and give the conditions for (local and global) validity used in the paper. We briefly describe the ConSpec language and (our version of) security automaton in Section 5. The example inlining algorithm is described briefly is Section 6. Section 7 introduces the ghost monitor, and Section 8, then, presents the main results of the paper, namely the algorithms for proof generation and proof recognition, including soundness proofs. Finally, Section 9 reports briefly on our prototype implementation, and we conclude by discussing some open issues and directions for future work.

2 Program Model

We assume that the reader is familiar with Java bytecode syntax and the Java Virtual Machine (JVM). Here, we only present components of the JVM, that are essential for the definitions in the rest of the text. Much of this is standard and can be skipped in a first reading. 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. We use c for (fully qualified) class names, m for method names, and f for field names. Types are either primitive or object types, i.e. classes, or arrays. Class declarations induce a class hierarchy, denoted by $<:$. 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.

We let v range over the set of all values of all types. Values of object type are (typed) locations, mapped to objects, or arrays, by a heap h. The typing assertion $h \vdash v : c$ asserts that v is some location ℓ, and that in the typed heap h, ℓ 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. 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.

A program is a set of classes, and for our purposes each class denotes a mapping from method identifiers to definitions (I, H) consisting of an instruction array I and an exception handler array H.

We write $c.m[L] = \iota$ to indicate that $c(m) = (I, H)$ and that I_L is defined and equal to the instruction ι. The exception handler array H is a list of of exception handlers. An exception handler (b, e, L, c) catches exceptions of type c and its subtypes raised by instructions in the range $[b, e)$ and transfers control to address L, if it is the first handler in the handler array that catches the exception for the given type and instruction.

A *configuration* of the JVM is a pair $C = (h, R)$ of a heap h and a stack R of activation records. For normal execution, the activation record at the top of the execution stack has the shape (M, pc, s, l), where M is the currently executing method, pc is the program counter, $s \in Val^*$ is the operand stack, and l is the local variable store. Except for API calls (see below) the transition relation $\rightarrow_{\text{JVM}}$ on JVM configurations is standard. A configuration $(h, (M, pc, s, l) :: R)$ 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 (ℓ) where ℓ is the location of an exceptional object, i.e. of class Throwable. Such a configuration is called *exceptional*. We say that C is *returning exceptionally* if C is exceptional, and if $C \rightarrow_{\text{JVM}} C'$ implies that C' is 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.

An *execution* E of a program P is a (possibly infinite) sequence of JVM configurations $C_0C_1\ldots$ 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, $pc = 0$ and for each $i \geq 0$, $C_i \rightarrow_{\text{JVM}} C_{i+1}$.

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.

The only non-standard aspect of $\rightarrow_{\text{JVM}}$ is the treatment of API methods. We assume a fixed API for which we have access only to the signature (types), but not the implementation, of its methods. We therefore treat API method calls as atomic instructions with a non-deterministic semantics. In this sense, we do not practice *complete mediation* [29]. When an API method is called either the *pc* is incremented and arguments popped from the operand 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 API calls. This property is destroyed by the *reflect* API, which is consequently not considered.

3 Assertions

Annotations are given in a language similar to the one described by F. Y. Bannwart and P. Müller in [3]. The syntax of assertions a and (partial) expressions e are given in the following BNF grammar:

$$e ::= v \mid e.f \mid c.f \mid \mathrm{s}_i \mid \mathrm{l}_i \mid e \circ e \mid a \rightarrow e|e \mid (e,e) \mid \bot$$
$$a ::= tt \mid ff \mid e\ r\ e \mid a \wedge a \mid \neg a \mid e : c$$

where $i \in \omega$. The semantics, as mappings $\|e\|C$ and $\|a\|C$ is given in Figure 2. The operations $\circ$ and r are generic binary operators/relation symbols, respectively, with Kleene equality. The expression s_i refers to the i'th element of the operand stack, and l_i refers to the i'th local variable; the expression $a \rightarrow e_1 \mid e_2$ is a conditional, (e_1, e_2) is pairing and tt and ff represent true and false respectively; a *heap assertion* is an assertion that does not reference the stack, or any of the local variables. Disjunction ($\vee$) and implication ($\Rightarrow$) are defined as usual. We let $\textsc{if}(a_0, a_1, a_2)$ denote the conditional expression $(a_0 \Rightarrow a_1) \wedge (\neg a_0 \Rightarrow a_2)$ and $\textsc{select}(\mathbf{a_1}, \mathbf{a_2}, a_{else})$ the generalized conditional expression $\textsc{if}(a_{1,0}, a_{2,0}, \textsc{if}(a_{1,1}, a_{2,1}, \ldots, \textsc{if}(a_{1,n}, a_{2,n}, a_{else})\ldots))$.

$$\begin{aligned}
\|e.f\|(h,R) &= h(\|e\|(h,R)).f \\
\|c.f\|(h,R) &= h(c.f) \\
\|\mathrm{s}_i\|(h,(M,pc,s,r)::R) &= s_i \\
\|\mathrm{l}_i\|(h,(M,pc,s,r)::R) &= l_i \\
\|e_1 \circ e_2\|C &= \|e_1\|C \circ \|e_2\|C \\
\|e_1 \rightarrow e_2 \mid e_3\|C &= \begin{cases} \|e_2\|C\ , & \|e_1\|C = tt \\ \|e_3\|C, & \text{otherwise} \end{cases} \\
\|(e_1,e_2)\|C &= (\|e_1\|C, \|e_2\|C) \\
\|\bot\|C &= \bot \\
\|tt\| &= tt \\
\|ff\| &= ff \\
\|e_1\ r\ e_2\|C &= \|e_1\|C\ r\ \|e_2\|C \\
\|a_1 \wedge a_2\|C &= \|a_1\|C \wedge \|a_2\|C \\
\|\neg a\|C &= \overline{\|a\|C} \\
\|e : c\|(h,R) &= \begin{cases} tt & \text{if } h \vdash \|e\|(h,R) : c \\ ff & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 2: Semantics of expressions and assertions

4 Extended Method Definitions

In this section we extend the method definitions by an array of program point assertions and by invariants at method entry and (normal or exceptional) return.

Definition 1 (Extended Method Definition)**.** *An* extended method definition *is a tuple* $(I, H, A, pre, post)$ *in which* (I, H) *is a method definition,* A *is an array of assertions such that* $|I| = |A|$ *and pre and post are heap assertions. An* extended program *is a program with extended methods.*

For extended programs, the notions of transition and execution are not affected by the presence of assertions. An extended program is valid, if all anno-

I_L	$wp_{(I,H,A,pre,post)}(L)$
`instanceof` c	$A_{L+1}[\mathrm{s}_0 : c/\mathrm{s}_0]$
`aload` n	$\mathit{unshift}(A_{L+1}[\mathrm{l}_n/\mathrm{s}_0])$
`astore` n	$(\mathit{shift}(A_{L+1})) \wedge \mathrm{s}_0 = \mathrm{l}_n$
`athrow`	$\textsc{select}((\mathrm{s}_0 : c \wedge b \leq L < e)_{(b,e,L',c)\in H}$ $(A_{L'})_{(b,e,L',c)\in H}, post)$
`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'	$\textsc{if}(\mathrm{s}_0 = \mathrm{s}_1, \mathit{shift}^2(A_{L'}), \mathit{shift}^2(A_{L+1}))$
`ifeq` L'	$\textsc{if}(\mathrm{s}_0 = 0, \mathit{shift}(A_{L'}), \mathit{shift}(A_{L+1}))$
`invoke` $c.m$	$\bigwedge_{c'\in defs(c.m)} pre_{c'.m}$
`putstatic` $c.f$	$\mathit{shift}(A_{L+1})[\mathrm{s}_0/c.f]$
`return`	$post$
`ldc` v	$\mathit{unshift}(A_{L+1}[v/\mathrm{s}_0])$
`invokestatic` `System.exit` tt	

Table 1: Specification of the wp_M function

tations are validated by their corresponding configurations in any execution starting in a configuration satisfying the initial pre-condition. In other words:

Definition 2 (Extended Program Validity)**.** *An extended program P is* valid *if for each maximal execution $E = C_0C_1 \cdots C_k$ of P*

1. *$\|pre_{\texttt{main}}\|C_0$ holds,*
2. *$\|post_{\texttt{main}}\|C_k$ holds, and*
3. *for each i such that $0 \leq i \leq k$, if C_i has the shape $((M, pc, s, r) :: R, h)$ and $P(M) = (I, H, A, pre, post)$ then $\|A_{pc}\|C_i$ holds*

The WP-calculus used in the proof generation / recognition is given in Table 1. The definition uses the auxillary functions *shift* and *unshift* which increments, resp. decrements, each stack index by one and *defs*$(c.m)$ which denotes the set of all classes c' such that $c <: c'$ and c' defines m. The account of dynamic call resolution in Table 1 is crude, but the details are unimportant since, in this paper, pre- and post-conditions are always identical and common to all methods.

A locally valid method is one for which each assertion can be validated by reference to "neighbouring" assertions only.

Definition 3 (Local Validity)**.** *An extended method $M = (I, H, A, pre, post)$ is locally valid, if the verification conditions*

1. *$pre \Rightarrow A_0$, and*
2. *$A_L \Rightarrow wp_M(L)$ for all $0 \leq L < |I|$*

are valid. An extended program is locally valid if all its methods are locally valid and the pre-condition of the main method holds in an initial configuration.

We note that local validity implies validity, as expected.

Theorem 1 (Local Validity Implies Validity)**.** *For any extended program P, if P is locally valid then P is valid.*

Proof. Follows by induction on the length of the execution. For details we refer to the Coq formalization [24]. □

5 Security Specifications

We consider security specifications written in a policy specification language ConSpec [2], similar to PSlang [14], but more constrained, to be amenable to analysis. An example specification is given in Figure 3. The syntax is intended to be largely self-explanatory: The specification in Figure 3 states that the program can only send files using the Bluetooth Obex protocol upon direct request by the user. No exception may arise during evaluation of 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 *security relevant*. In the example there are two security relevant methods, `GUI.fileSendQuery` and `Bluetooth.obexSend`. The specification expresses constraints in terms BEFORE, AFTER and EXCEPTIONAL clauses. Each clause is a guarded command where the guards are side-effect

```
SECURITY STATE String lastApproved = "";

AFTER file = GUI.fileSendQuery()
    PERFORM true -> { lastApproved = file; }

EXCEPTIONAL GUI.fileSendQuery()
    PERFORM

BEFORE Bluetooth.obexSend(String file)
    PERFORM file = lastApproved -> { }
```

Figure 3: A security specification example written in ConSpec.

free and terminating boolean expressions, and the assignment updates the security state. Guards may involve constants, method call parameters, object fields, and values returned by accessor or test methods that are guaranteed to be side-effect free and terminating. Guards are evaluated top to bottom in order to obtain a deterministic semantics. If no clause guard holds, the policy is violated. In return clauses the guards must be exhaustive.

5.1 Security Automata

A ConSpec contract determine a security automaton (Q, Σ, δ, q_0) where Q is a countable (not necessarily finite) set of states, Σ is the alphabet of security relevant actions, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Sigma \to Q$ is the transition function. We assume a special error state $\perp \in Q$ and view all states in Q except $\perp$ as accepting. We require that security automata are strict in the sense that $\delta(\perp, \alpha) = \perp$.

A security automata is generated by a ConSpec contract in a straightforward manner (cf. [1]). The alphabet Σ is partitioned into pre-actions (for calls) and (normal or exceptional) post-actions (for normal or exceptional returns). Pre-actions have the form $(c.m, \mathbf{v})^{\uparrow}$, normal post-actions have the form $(c.m, \mathbf{v}, r)^{\downarrow}$ and exceptional post-actions have the form $(c.m, \mathbf{v})^{\Downarrow}$, where $c.m$ is the relevant API-method, $\mathbf{v}$ is the arguments used when calling the method, and r is the returned value.

Executions produce security relevant actions in the expected manner. A calling configuration generates a pre-action determined by the called method and the current arguments (top n operand stack values for an n-ary method). A returning configuration then gives rise to a normal post-action determined by the identifier of the returning method and the return value (top operand stack value). For sake of simplicitly we assume that all API methods return a value. An exceptionally returning configuration generates an exceptional post-action determined by the method identifier of the returning method. The security relevant actions (the security relevant trace) of an execution E is denoted by $SRT(E)$ and formally defined below.

Definition 4 (Security Relevant Trace)**.** *The security relevant trace, $SRT(E)$, of an execution E is defined as*

$$
\begin{aligned}
SRT(E) &= SRT(E, \epsilon) \\
SRT(\epsilon, \epsilon) &= \epsilon \\
SRT(CE, \gamma) &= \begin{cases}
(c'.m', \mathbf{v})^{\uparrow} SRT(E, \mathbf{v} :: \gamma) \\
\quad \textit{if } C = (h, (c.m, pc, \mathbf{v} :: s, l) :: R) \\
\quad \textit{is calling } c'.m' \\
(c.m, \mathbf{v}, r)^{\downarrow} SRT(E, \gamma') \\
\quad \textit{if } C = (h, (c.m, pc, r :: s, l) :: R) \\
\quad \textit{is returning and } \gamma = \mathbf{v} :: \gamma' \\
(c.m, \mathbf{v})^{\Downarrow} SRT(E, \gamma') \\
\quad \textit{if } C = (h, (o) :: R) \textit{ is returning} \\
\quad \textit{exceptionally and } \gamma = \mathbf{v} :: \gamma' \\
SRT(E, \gamma) \qquad \textit{otherwise}
\end{cases}
\end{aligned}
$$

We generally identify a ConSpec contract with its set of security relevant traces, i.e. the language recognized by its corresponding security automaton. A program is said to *adhere* to a contract if all its security relevant traces are accepted by the contract.

Definition 5 (Contract Adherence)**.** *The program P adheres to contract $\mathcal{C}$ if for all executions E of P, $SRT(E) \in \mathcal{C}$.*

6 Example Inliner

In this section we give an algorithm for monitor inlining (from now on referred to as an inlining algorithm, or simply an inliner) in the style of Erlingsson [15]. As previously mentioned, the developer is free to decide what inlining strategy to use, so the algorithm presented here serves merely as an example and does for instance not include any optimizations. For the implementation details and an example, we refer to Appendix A.

The inliner traverses the instructions and replaces each invoke instruction with a block of monitoring code. This block of code first stores the method arguments in local variables for use in post-actions. Then the class hierarchy is traversed bottom up for virtual call resolution, and when a match is found the relevant clauses, guards, and updates are enacted. For post-actions the main difference is in exception handling; exceptions are rerouted for clause evaluation, and then rethrown.

We refer to the method resulting from inlining a method M (program P) with a contract $\mathcal{C}$ as $\mathcal{I}(M, \mathcal{C})$ ($\mathcal{I}(P, \mathcal{C})$). The main correctness property we are after for inlined code is contract compliance:

Theorem 2 (Inliner Correctness)**.** *The inlined program $\mathcal{I}(P, \mathcal{C})$ adheres to $\mathcal{C}$.*

Proof. This follows from the fact that we are always able to generate a valid adherence proof (theorem 4) and that the existence of such adherence proof ensures contact adherence (theorem 3). (Both statements are proved in later sections.) □

7 The Ghost Monitor

The purpose of the ghost monitor is to keep track of what the embedded monitor state should be at key points during method execution. This provides a useful reference for verification. Moreover, since the ghost monitor assigns only to special ghost variables that are invisible to the client program, and since it is incapable of blocking, it does not in fact have any observable effect on the client program.

The ghost monitor uses special assignments which we refer to as ghost updates: Guarded multi-assignment commands used for updating the state of the ghost monitor and for storing method call arguments and dynamic class identities in temporary variables. A ghost update has the shape $\langle \mathbf{x}^g := e \rangle$ where $\mathbf{x}^g$ is a tuple of ghost variables, special variables used only by the ghost monitor, and e is an expression of matching type. Typically, e is a conditional of similar shape as the policy expressions, and e may mention security state ghost variables as well as other ghost variables holding security relevant call parameters. Given the post-condition A_{L+1}, the weakest pre-condition for the ghost instruction $\langle \mathbf{x}^g := e \rangle$ at label L is $wp_M(L) = A_{L+1}[e/\mathbf{x}^g]$.

The ghost updates are embedded right before and after each security relevant invoke instruction as well as in an exception handler catching any exception (`Throwable`) thrown by the invoke instruction and nothing else. Note that the existence of such an exception handler is easily checked, and that the code delivered by our inliner always has exception handlers of this form. The details are presented in Figure 4. A method M with ghost updates embedded, corresponding to the security automaton of a contract $\mathcal{C}$ is denoted by $\mathcal{I}^g(M, \mathcal{C})$.

Let $\mathcal{I}^g(M, \mathcal{C})$ be the result of embedding a ghost monitor corresponding to contract $\mathcal{C}$ into M. The key property of the ghost monitor is that the trace of ghost monitor states in an execution E, is the same as the states visited by the security automaton, given $SRT(E)$ as input. This is easily be shown by an induction over the length of E.

Lemma 1. *Let $E = C_0 \dots C_k$ be an execution of $\mathcal{I}^g(P, \mathcal{C})$ and $\mathbf{ms}_i^g$ denote the ghost monitor state in configuration C_i. If for all $0 \leq i \leq k$, $\mathbf{ms}_i^g \neq \bot$, then $SRT(E) \in \mathcal{C}$.*

Proof. Follows by induction on the length of the execution. For details we refer to the Coq formalization [24]. □

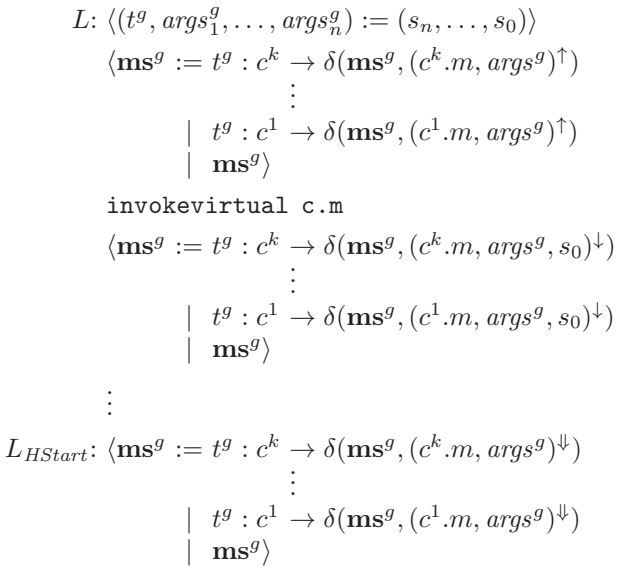

Figure 4: Ghost updates induced by security automaton (Q, Σ, δ, q_0) for an invokation of $c.m$, where t^g is the target object, $args^g$ represents the arguments and $c^1 <: \ldots <: c^k$ denote all API-classes defining or overriding m.

8 Contract Adherence Proofs

The key idea of a contract adherence proof is to show that the embedded monitor state **ms** of the program $\mathcal{I}^g(P, \mathcal{C})$ and the ghost monitor state $\mathbf{ms}^g$ are in agreement at certain program points. These points certainly need to include all potentially security relevant call and return sites. But, since we aim for a procedural analysis, and to cater for virtual call resolution actually all call and return sites are included.

In fact, this is all that is needed, and hence:

Definition 6 (Adherence Proof)**.** *An adherence proof for program P and contract $\mathcal{C}$ assigns to each method $M = (I, H)$ in $\mathcal{I}^g(P, \mathcal{C})$ an assertion array A such that the extended method $(I, H, A, \mathbf{ms} = \mathbf{ms}^g, \mathbf{ms} = \mathbf{ms}^g)$ is locally valid.*

Such an account has two main benefits which are heavily exploited below:

- It leaves the choice of a particular proof generation strategy open.
- It opens for a lightweight approach to on-device proof checking, by performing the local validity check on a program with a locally produced ghost monitor.

Theorem 3 (Adherence Proof Soundness)**.** *If an adherence proof exists for a program P and contract $\mathcal{C}$, then P adheres to $\mathcal{C}$.*

Proof. Assume Π is an adherence proof for a program P and a contract $\mathcal{C}$. By theorem 1 we know that the corresponding extended program for $\mathcal{I}^g(P, \mathcal{C})$ is globally valid. This implies that $\mathbf{ms} = \mathbf{ms}^g$ at each configuration that is calling (or returning from) a security relevant configuration. Furthermore, since the $\perp$ value is an artifical "error" value of the security automaton with no Java counterpart, we know that if $\mathbf{ms} = \mathbf{ms}^g$, then $\mathbf{ms}^g \neq \perp$. Thus, by lemma 1, $SRT(E) \in \mathcal{C}$ and therefore P adheres to $\mathcal{C}$. □

8.1 Example Proof Generation

The process of generating contract adherence proofs is closely related to the process of embedding the ref-

erence monitor, thus the inlining and proof generation is preferrably done by the same agent. This section describes how proofs may be generated for code produced by the example inliner presented in Section 6.

The monitor invariant, $\mathbf{ms} = \mathbf{ms}^g$ is set as each methods pre- and post-condition. The assertion for each specific instruction is generated differently, according to whether the instruction appears as part of an inlined block or not. Instructions inside the inlined block affect the processing of the embedded state, method call arguments etc. For this reason these instructions need detailed analysis using the *wp* function. Instructions outside the inlined blocks, on the other hand, allow a more robust treatment, as they are only required to preserve the monitor invariant which they do (see fact 1 in Appendix A). The critical property of the annotation function is the following:

Lemma 2. *Given a method $M = (I, H)$ of $\mathcal{I}^g(P, \mathcal{C})$ and a set IL labelling the inlined instructions in I, an array A of assertions can be computed such that the extended method $(I, H, A, \mathbf{ms} = \mathbf{ms}^g, \mathbf{ms} = \mathbf{ms}^g)$ is locally valid.*

Proof. A general construction is illustrated in Appendix B. □

The array is constructed by annotating the return instructions with the post-condition, and then in a breadth first manner, annotate the preceeding instructions using the *wp* function in case of inlined instructions and by using the monitor invariant in other cases.

Theorem 4 (Proof Generation)**.** *For each program P and contract $\mathcal{C}$ there is an algorithm, polynomial in $|P| + |\mathcal{C}|$, which produces an adherence proof of $\mathcal{I}(P, \mathcal{C})$.*

Proof. The algorithm described above treats each method in isolation. The breadth first traversal of the instructions takes time linearly proportional to the size of the instruction array plus the number of ghost updates. The resulting adherence proof is correct by construction. □

As an example Figure 6 illustrates a generated proof for a part of a program which has been inlined to comply with the policy in Figure 5.

```
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.

8.2 Proof Recognition

Checking the validity of contract adherence proofs involves verifying local validity, which in general is undecidable. However, the problem is much simplified in our setup, since proofs apply to programs that have already been inlined. Validity checking may still be hard or impossible, however, due to the use of primitive data types with difficult equational theories. For this reason the theorem below is restricted to contracts over freely generated theories.

Theorem 5 (Efficient Recognition)**.** *The class of adherence proofs generated from programs inlined with contracts over a freely generated theory is recognizable in polynomial time.*

Proof. To verify the validity of a given adherence proof we look at the requirements of definition 6. Verifying that the pre- and post-conditions equal the monitor invariant is a simple syntactic check and can be done in time linearly proportional to the number of methods in P.

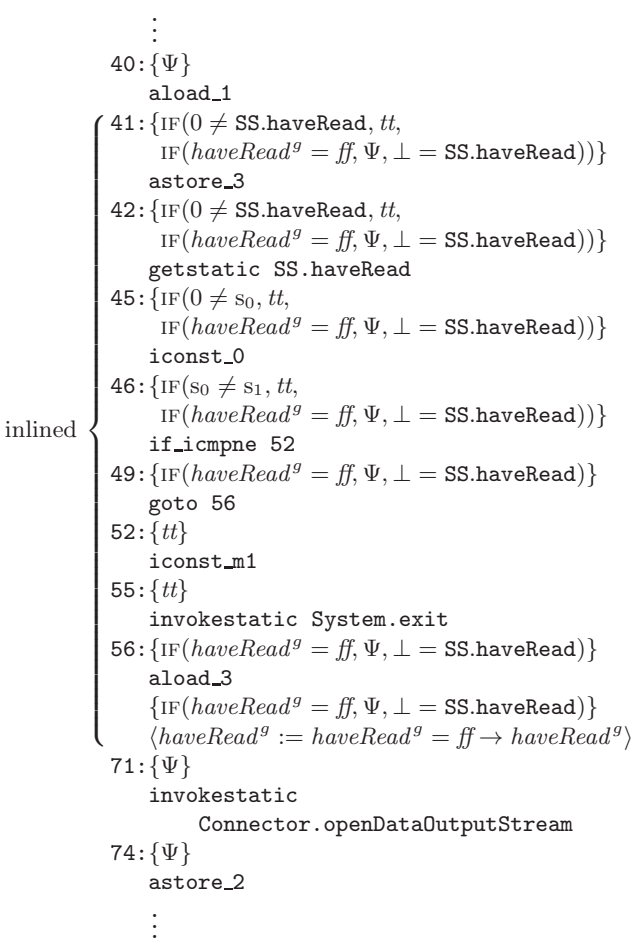

Figure 6: Generated assertions for inlining of `Connector.openDataOutputStream` where Ψ denotes the monitor invariant.

For the requirement of local validity, it is sufficient to check that the verification conditions 1 and 2 from definition 3 can be rewritten to tt in time polynomial in the size of the instruction array. The interesting verification conditions are those of the form $A_L \Rightarrow wp_M(L)$ where L is the label of the first instruction in an inlined block. A_L is, in this case, of the form $\mathbf{ms} = \mathbf{ms}^g \wedge a_0^g = a_0 = s_0 \wedge \ldots \wedge a_m^g = a_m = s_m$ and $wp_M(L)$ is of the form

$$
\begin{array}{l}
(\textsc{select}((t : c^n \wedge t^g : c^n, \ldots, t : c^1 \wedge t^g : c^1),\\
\quad (\textsc{select}((c^n.m_{G_1} \wedge c^n.m^g_{G_1}, \ldots, c^n.m_{G_i} \wedge c^n.m^g_{G_i}),\\
\qquad (c^n.m_{f_1}(\mathbf{ms}, \mathbf{a}) = c^n.m^g_{f_1}(\mathbf{ms}^g, \mathbf{a}^g), \ldots,\\
\qquad\quad c^n.m_{f_i}(\mathbf{ms}, \mathbf{a}) = c^n.m^g_{f_i}(\mathbf{ms}^g, \mathbf{a}^g)), tt),\\
\qquad \vdots \quad \vdots\\
\quad \textsc{select}((c^1.m_{G_1} \wedge c^1.m^g_{G_1}, \ldots, c^1.m_{G_j} \wedge c^1.m^g_{G_j}),\\
\qquad c^1.m_{f_1}(\mathbf{ms}, \mathbf{a}) = c^1.m^g_{f_1}(\mathbf{ms}^g, \mathbf{a}^g), \ldots,\\
\qquad c^1.m_{f_j}(\mathbf{ms}, \mathbf{a}) = c^1.m^g_{f_j}(\mathbf{ms}^g, \mathbf{a}^g)), tt)),\\
\mathbf{ms} = \mathbf{ms}^g)
\end{array}
$$

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 can be performed in time proportional to the length of the formula and does not increase the size of the expression since x, y and z are atomic. The result can then be rewritten to tt using the rules $(\psi \Rightarrow \phi) \wedge (\neg\psi \Rightarrow \phi) \longrightarrow \phi$ and $\phi = \phi \longrightarrow tt$ in time polynomial in the size of the formula.

All other verification conditions ($pre_M \Rightarrow A_0$, $A_L \Rightarrow wp_M(L)$ for all labels L except those of the first instructions in an inlined block are trivial as their antecedents and succeedents are identical. □

9 Implementation and Evaluation

A full implementation of the framework, including a Java SE proof generator, a Java ME client, instructions and examples is available at `www.csc.kth.se/~landreas/irm_pcc`. Both the on- and the off-device software utilize a parser generated by CUP / JFlex [19, 21] and the ASM library [27] for handling class files. Table 2 summarizes overhead for inlining, proof generation and load-time proof recognition on two example applications and policies:

- *MobileJam*: A GPS based traffic jam reporter which utilizes the Yahoo! Maps API.
  Policy: Only connect to `http://local.yahooapis.com`.
- *Snake*: A classic game of snake in which the player may submit current score to a server.

	MobileJam	Snake
Security Relevant Invokes	4	2
Original Size	428.0 kb	43.7 kb
Size increase for IRM	4.8 kb	1.1 kb
Size increase for Proofs	20.6 kb	2.6 kb
Inlining	10.1 s	8.6 s
Proof Generation	4.7 s	0.8 s
Proof Recognition	98 ms	117 ms

Table 2: Benchmarks for the two case studies.

> Policy: Do not send data over network after reading from phone memory.

Inlining and proof generation was performed on an Intel Core 2 CPU at 1.83 GHz with 2 Gb memory and proof recognition was performed on a Sony-Ericsson W810i. The implementation is to be considered a prototype, and very few optimizations in terms of e.g. proof size have been implemented.

10 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 partially a question raised by K. W. Hamlen et al. [18].

We have proved correctness of our approach in the sense of soundness: Contract adherence proofs are sufficient to ensure compliance. This soundness proof has been formalized [24] in Coq. 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, at least when contracts do not use equational tests that are too difficult. Other properties are also interesting such as transparency [29], roughly, that all adherent behaviour is preserved 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. [23, 34, 10, 9] for results in this direction).

The approach is efficient: Proofs are small and recognised easily, by a simple proof checker. 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. Worth pointing out also is that 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.

It is possible to extend our framework to multi-threading by protecting security relevant updates with locks, either locking the entire inlined block or releasing the lock during the security relevant call itself for increased parallellism. For proof generation the main upshot is that assertions must be stable under interference by other threads. Briefly, this requires the ability to protect fields, such as those in the security state class, with locks by only allowing updates of these fields when the lock has been acquired. The validity of an assertion may then only depend on fields protected by locks that has been acquired at that point in the code. This work is currently in progress.

References

[1] Irem Aktug, Mads Dam, and Dilian Gurov. Provably correct runtime monitoring. In *FM '08: Proceedings of the 15th international symposium on Formal Methods*, pages 262–277. Springer-Verlag, 2008.

[2] Irem Aktug and Katsiaryna Naliuka. ConSpec – a formal language for policy specification. *Electronic Notes in Theoretical Computer Science*, 197(1):45–58, 2008.

[3] F. Y. Bannwart and P. Müller. A logic for bytecode. In *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*,

volume 141-1 of *Electronic Notes in Theoretical Computer Science*, pages 255–273. Elsevier, 2005.

[4] Gilles Barthe, Pierre Crégut, Benjamin Grégoire, Thomas P. Jensen, and David Pichardie. The MOBIUS Proof Carrying Code Infrastructure. In *FMCO*, pages 1–24, 2007.

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

[6] Lujo Bauer, Jarred Ligatti, and David Walker. Types and effects for non-interfering program monitors. In M. Okada, B. Pierce, A. Scedrov, H. Tokuda, and A. Yonezawa, editors, *Software Security—Theories and Systems. Mext-NSF-JSPS International Symposium*, volume 2609 of *Lecture Notes in Computer Science*, pages 154–171. Springer, 2003.

[7] N. Bielova, N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Matching in security-by-contract for mobile code. *Journal of Logic and Algebraic Programming*, 78(5):340 – 358, 2009.

[8] Feng Chen. Java-MOP: A monitoring oriented programming environment for Java. In *In Proceedings of the Eleventh International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS*, pages 546–550. Springer, 2005.

[9] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Security monitor inlining for multithreaded Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings,*, pages 546–569. Springer-Verlag, 2009.

[10] Mads Dam, Bart Jacobs, Andreas Lundblad, and Frank Piessens. Provably correct inline monitoring for multithreaded Java-like programs. *Journal of Computer Security*, 18:37 – 59, 2010.

[11] 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. ACM, 2001.

[12] Lieven Desmet, Wouter Joosen, Fabio Massacci, Pieter Philippaerts, Frank Piessens, Ida Siahaan, and Dries Vanoverberghe. Security-by-Contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32, 2008.

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

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

[15] 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. ACM Press, 2000.

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

[17] Yoonsik Cheon Gary T. Leavens. Design by Contract with JML. `http://www.eecs.ucf.edu/~leavens/JML/jmldbc.pdf`, 2006.

[18] 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.

[19] Scott Hudson. Cup. `http://www2.cs.tum.edu/projects/cup/`, March 2003.

[20] M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Java-MaC: A run-time assurance tool for Java programs. *Electronic Notes in*

Theoretical Computer Science, 55(2):218 – 235, 2001. RV'2001, Runtime Verification (in connection with CAV '01).

[21] Gerwin Klein. JFlex. `http://jflex.de/`, October 2007.

[22] 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, 2005.

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

[24] Andreas Lundblad. Coq-formalization of the security theorems of the IRM / PCC approach. `http://www.csc.kth.se/~landreas/irm_pcc/coq`, 2010.

[25] K. Naliuka N. Dragoni, F. Massacci and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In *Proc. 4th European PKI Workshop*, volume 4582 of *Lecture Notes in Computer Science*, pages 297–312. Springer, 2007.

[26] G. C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119. ACM Press, 1997.

[27] ObjectWeb. ASM web page. `http://asm.objectweb.org/`, February 2008.

[28] Project web page. `http://www.s3ms.org`, 2008.

[29] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

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

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

[32] Meera Sridhar and Kevin W. Hamlen. Actionscript in-lined reference monitoring in prolog. In *PADL*, pages 149–151, 2010.

[33] Meera Sridhar and Kevin W. Hamlen. Model checking in-lined reference monitors. In *Verification, Model Checking, and Abstract Interpretation*, pages 312–327, 2010.

[34] Dries Vanoverberghe and Frank Piessens. Security enforcement aware software development. *Inf. Softw. Technol.*, 51(7):1172–1185, 2009.

[35] 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. ACM, 2000.

[36] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. *Security and Privacy, IEEE Symposium on*, 0:79–93, 2009.

A Implementation of the Example Inliner

Our inliner lets the state of the embedded security monitor be represented by 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 execution and allows for our open-ended treatment of large parts of the instruction set.

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

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 $c.f$. In particular, the assumption ensures that instructions originating from the target program are unable to affect the embedded monitor state.

For simplicity we assume (without loss of generality) that ConSpec policies initialize the security state variables to the default Java values.

Each `invokevirtual` $c.m$ instruction is replaced by a block of inlined code that evaluates which concrete method is being invoked, then checks and updates the security state accordingly. We assume for simplicity that no instructions in a block of inlined code other than `athrow` will raise exceptions. The code is easily adapted at the cost of some additional complexity to take runtime exceptions violating this assumption into account.

Figure 7 shows a schematic policy for a method m : `int` $\rightarrow$ `int` defined in class a c and overridden in a subclass d. The policy has event clauses for BEFORE, AFTER and EXCEPTIONAL cases for each definition of m, each with two guards and two statement lists.

Figure 8 gives the inlining details for the policy schema in Figure 7. In the figure, each *`[EVALUATE g]`* section transforms a configuration $(h, (M, pc, s, r) :: R)$ to $(h, (M, pc', v :: s, r) :: R)$ where v is 0 or 1 if the guard g is false or true respectively. An *`[EXECUTE stmts]`* transforms the configuration $(h, (M, pc, s, r) :: R)$ to $(h[[\![stmts]\!](\mathbf{ms})/\mathbf{ms}], (M,\ pc', s, r) :: R)$.

The remaining invoke instructions (`invokestatic`, `invokeinterface` and `invokespecial`) can be handled similarly.

```
SCOPE Session
SECURITY STATE int ms = 0;

BEFORE      c.m(int a) PERFORM cb_g1 -> {cb_s1} | cb_g2 -> {cb_s2}
AFTER   r = c.m(int a) PERFORM ca_g1 -> {ca_s1} | ca_g2 -> {ca_s2}
EXCEPTIONAL c.m(int a) PERFORM ce_g1 -> {ce_s1} | ce_g2 -> {ce_s2}

BEFORE      d.m(int a) PERFORM db_g1 -> {db_s1} | db_g2 -> {db_s2}
AFTER   r = d.m(int a) PERFORM da_g1 -> {da_s1} | da_g2 -> {da_s2}
EXCEPTIONAL d.m(int a) PERFORM de_g1 -> {de_s1} | de_g2 -> {de_s2}
```

Figure 7: Schematic ConSpec policy

```
  tArgs: astore r_a
         astore r_t
         aload r_t
         aload r_a
  dbChk: aload r_t
         instanceof d
         ifeq cbChk
 dbGrd1: [EVALUATE db_g1]
         ifeq dbGrd2
         [EXECUTE db_s1]
         goto BEnd
 dbGrd2: [EVALUATE db_g2]
         ifeq dBFail
         [EXECUTE db_s2]
         goto BEnd
 dBFail: iconst_1
         inv_static Sys.exit
  cbChk: aload r_t
         instanceof c
         ifeq BEnd
 cbGrd1: [EVALUATE cb_g1]
         ifeq cbGrd2
         [EXECUTE cb_s1]
         goto BEnd
 cbGrd2: [EVALUATE cb_g2]
         ifeq cbFail
         [EXECUTE cb_s2]
         goto BEnd
 cbFail: iconst_1
         inv_static Sys.exit
   BEnd: invokevirtual c.m
         goto hdlEnd
hdlStrt: aload r_t
         instanceof d
         ifeq ceChk
 deGrd1: [EVALUATE de_g1]
         ifeq deGrd2
         [EXECUTE de_s1]
         goto EEnd
 deGrd2: [EVALUATE de_g2]
         ifeq deFail
         [EXECUTE de_s2]
         goto EEnd
deFail: iconst_1
         inv_static Sys.exit
 ceChk: aload r_t
         instanceof c
         ifeq EEnd
ceGrd1: [EVALUATE ce_g1]
         ifeq ceGrd2
         [EXECUTE ce_s1]
         goto EEnd
ceGrd2: [EVALUATE ce_g2]
         ifeq ceFail
         [EXECUTE ce_s2]
         goto EEnd
ceFail: iconst_1
         inv_static Sys.exit
  EEnd: athrow
hdlEnd: aload r_t
         instanceof d
         ifeq caChk
daGrd1: [EVALUATE da_g1]
         ifeq daGrd2
         [EXECUTE da_s1]
         goto AEnd
daGrd2: [EVALUATE da_g2]
         ifeq daFail
         [EXECUTE da_s2]
         goto AEnd
daFail: iconst_1
         inv_static Sys.exit
 caChk: aload r_t
         instanceof c
         ifeq AEnd
caGrd1: [EVALUATE ca_g1]
         ifeq caGrd2
         [EXECUTE ca_s1]
         goto AEnd
caGrd2: [EVALUATE ca_g2]
         ifeq caFail
         [EXECUTE ca_s2]
         goto AEnd
caFail: iconst_1
         inv_static Sys.exit
  AEnd:
```

Figure 8: Schematic inlining of policy in Figure 7

B Proof of Lemma 2

Figure 10 shows the construction for a call of a method m : `int` $\rightarrow$ `int` in class c, under the schematic contract shown in Figure 9. We assume that an exception thrown by the invoked method is matched by an exception handler table entry on the form $(30, 32, 34, any)$. For brevity we let σ_{bef}, σ_{aft} and σ_{exc} denote the appropriate substitution for the effect of updating **ms** according to the before, after and exceptional clause of $c.m$ respectively. For instance, if $\texttt{bef}_s$ denotes `ms = ms * x; ms = ms - 5`, then σ_{bef} is $[(\mathbf{ms} \cdot x) - 5/\mathbf{ms}]$.

```
SCOPE Session
SECURITY STATE DECLARATION

BEFORE      c.m(int a) PERFORM bef_g -> {bef_s}
AFTER   r = c.m(int a) PERFORM aft_g -> {aft_s}
EXCEPTIONAL c.m(int a) PERFORM exc_g -> {exc_s}
```

Figure 9: Schema contract for the proof of Lemma 2.

$\mathbf{ms} = \mathbf{ms}^g$
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$(\texttt{bef}_g, \text{IF}(s_1 : c, \text{IF}(\texttt{bef}_g, \mathbf{ms}\sigma_{bef}(a) = \mathbf{ms}^g\sigma_{bef}(s_0), \mathbf{ms}\sigma_{bef} = \bot),$
$\mathbf{ms} = \mathbf{ms}^g) \wedge a = s_0 \wedge t = s_1, tt)$
[EVALUATE $\texttt{bef}_g$]
IFEQ 29
[PERFORM $\texttt{bef}_s$]
GOTO 30
29: tt
ICONST_1
INVOKESTATIC System.exit
30: IF$(s_1 : c, \text{IF}(\texttt{bef}_g, \mathbf{ms} = \mathbf{ms}^g\sigma_{bef}(s_0), \mathbf{ms} = \bot), \mathbf{ms} = \mathbf{ms}^g) \wedge$
$a = s_0 \wedge t = s_1$
$\langle (t^g, a^g) := (s_1, s_0) \rangle$
$\langle \mathbf{ms}^g := t^g : c \rightarrow \delta(\mathbf{ms}^g, (c.m, a^g)^{\uparrow}) \mid \mathbf{ms}^g \rangle$
$\mathbf{ms} = \mathbf{ms}^g \wedge a = a^g \wedge t = t^g$
INVOKEVIRTUAL c.m(int) : int
32: $\mathbf{ms} = \mathbf{ms}^g \wedge a = a^g \wedge t = t^g$
$\langle r^g := s_0 \rangle$
$\langle \mathbf{ms}^g := t^g : c \rightarrow \delta(\mathbf{ms}^g, (c.m, a^g, r^g)^{\downarrow}) \mid \mathbf{ms}^g \rangle$
$A_{43}[r/\text{s}_0]$
ASTORE r
ALOAD r
A_{43}
GOTO 43
// EXCEPTIONAL
34: $\mathbf{ms} = \mathbf{ms}^g \wedge a = a^g \wedge t = t^g$
$\langle \mathbf{ms}^g := t^g : c \rightarrow \delta(\mathbf{ms}^g, (c.m, a^g)^{\Downarrow}) \mid \mathbf{ms}^g \rangle$
38: IF$(t : c, A_{40}, A_{42})$
ALOAD t
INSTANCEOF c
IFEQ 42
40: IF$(\texttt{exc}_g, \mathbf{ms}\sigma_{exc}(a) = \mathbf{ms}^g, A_{41})$
[EVALUATE $\texttt{exc}_g$]
IFEQ 41
[PERFORM $\texttt{exc}_s$]
GOTO 42
41: tt
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$(\texttt{aft}_g, \mathbf{ms}\sigma_{aft}(r, a) = \mathbf{ms}^g, tt)$
[EVALUATE $\texttt{aft}_g$]
IFEQ 45
[PERFORM $\texttt{aft}_s$]
GOTO 46
45: tt
ICONST_1
INVOKESTATIC System.exit
// INLINING END
46: $\mathbf{ms} = \mathbf{ms}^g$
NON-INLINED INSTRUCTION

Figure 10: Schematic annotation for contract displayed Figure 9