Security Monitor Inlining and Certification for Multithreaded Java

MADS DAM[1], BART JACOBS[2],
ANDREAS LUNDBLAD[1] and FRANK PIESSENS[2]
[1] Royal Institute of Technology (KTH), Sweden,
[2] Katholieke Universiteit Leuven, Belgium

Security monitor inlining is a technique for security policy enforcement whereby monitor functionality is injected into application code in the style of aspect-oriented programming. The intention is that the injected code enforces compliance with the policy (security), and otherwise interferes with the application as little as possible (conservativity and transparency). Such inliners are said to be correct. For sequential Java-like languages, inlining is well understood, and several provably correct inliners have been proposed. For multithreaded Java one difficulty is the need to maintain a shared monitor state. We show that this problem introduces fundamental limitations in the type of security policies that can be correctly enforced by inlining. A class of race-free policies is identified that precisely characterizes the inlineable policies by showing that inlining of a policy outside this class is either not secure or not transparent, and by exhibiting a concrete inliner for policies inside the class which is secure, conservative, and transparent. The inliner is implemented for Java and applied to a number of practical application security policies. Finally, we discuss how certification in the style of Proof-Carrying Code could be supported for inlined programs by using annotations to reduce a potentially complex verification problem for multithreaded Java bytecode to sequential verification of just the inlined code snippets.

1. Introduction

Security monitoring, cf. (Schneider, 2000; Ligatti, 2006), is a technique for security policy enforcement, widely used for access control, authorization, and general security policy enforcement in computers and networked systems. The conceptual model is simple: Security relevant events by an application program such as requests to read a certain file, or opening a connection to a given host, 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 history or other contextual information. This basic setup can be implemented in many different ways, at different levels of granularity. Two approaches of fundamental interest are known, respectively, as execution monitoring (EM) and inlined reference monitoring (IRM) (cf. (Hamlen et al., 2006b)). In EM (Schneider, 2000; Viswanathan, 2000), monitors perform the event interception and control explicitly, typically by an

agent external to the program being executed. Using IRM, cf. (Erlingsson and Schneider, 2000b), the enforcement agent modifies the application program prior to execution in order to guarantee policy compliance, for instance by weaving monitor functionality into the application code in an aspect oriented style. Upon encountering a program event which may be relevant to the security policy currently being enforced – such as an API call – the inlined code will typically retrieve both the application program state and the security state to determine if the program event should be allowed to go ahead, and if not, terminate execution.

Under the assumption that the external monitor is only given capabilities available to an IRM, execution monitoring and inlining enforce the same policies (Hamlen et al., 2006b).[†] But if the external monitor has stronger capabilities – for instance the capability to perform type-unsafe operations, external execution monitoring can be more powerful. Our first contribution is to show that such an effect arises in a multithreaded setting. The fact that an inlined monitor can only influence the scheduler indirectly – by means of the synchronization primitives offered by the programming language – has the consequence that certain policies cannot be enforced securely and transparently by an inlined reference monitor. In support of this statement we give a simple example of a policy which an inliner is either unable to enforce securely, or else the inliner will need to affect scheduling by locking in a way that can result in loss of transparency, performance degradation and, possibly, deadlocks. On the other hand, the policy is easily enforced by an execution monitor which at each computation step can inspect the global execution state.

In spite of this, inlining remains an attractive implementation strategy in many applications. We identify a class of *race-free policies*, and show that this class characterizes the policies which can be enforced correctly by inlining in multithreaded Java. We argue that the set of race-free policies is in fact the largest class that is meaningful in a multithreaded setting. Even if many inliners for multithreaded Java-like languages exist for non-race-free policies (Erlingsson, 2004; Bauer et al., 2005; Hamlen et al., 2006a), these inliners must necessarily sacrifice either security or transparency, and anyhow these policies are, in a multithreaded setting, likely to *not* express what the policy writer intended.

The characterization result is proved in two steps: First we show that no inliner exists which can enforce a non-race-free policy both securely and transparently without taking implementation specific details of the API, scheduler or JVM into account. Then, we exhibit a concrete inliner and prove that it correctly enforces all race-free policies.

A potential weakness of inlining is that there is a priori no way for a consumer of an inlined piece of code to tell that inlining has been performed correctly. This makes it hard to use IRM as a general software quality improvement tool. Also, it generally forces inlining and execution to take place under the same jurisdiction. To address this problem we turn to certification. For sequential code, certification can be done using Proof-Carrying Code (PCC) (Necula, 1997). In this case a code producer essentially ships along with the code a correctness proof, which can be efficiently validated at the

---

[†] In this paper security policies are viewed as sets of traces of observable, security relevant events. If we consider broader classes of policies for e.g. information flow, program rewriting can enforce strictly more policies (Hamlen et al., 2006b).

time the code is invoked by the code consumer. For multithreaded programs, however, the construction of general purpose program logics and verification condition generators is a significant research challenge. We bypass this problem by restricting attention to multithreaded Java bytecode produced using the IRM presented earlier. This allows us to produce security certificates for race-free ConSpec policies by combining existing program verification techniques for sequential Java with a small number of syntactic checks on the received code. Certificates are presented as bytecode augmented with a reference ("ghost") monitor. This allows the code consumer to validate certificates against a local, trusted policy by checking the certificate with the monitor suitably replaced. The main result is a soundness result, that if a certificate exists for a program with a given policy, then the program is secure, i.e. the policy is guaranteed not to be violated.

1.1. *Related Work*

Our approach adopts the Security-by-Contract (SxC) paradigm (cf. (Bielova et al., 2009; N. Dragoni and Siahaan, 2007; Desmet et al., 2008; Kim et al., 2001; Chen, 2005)) which has been explored and developed mainly within the S^3MS project (S^3MS, 2008).

Monitor inlining has been considered by a large number of authors, for a wide range of languages, mainly sequential ones, cf. (Deutsch and Grant, 1971; Erlingsson and Schneider, 2000b; Erlingsson and Schneider, 2000a; Erlingsson, 2004; Aktug et al., 2009; Vanoverberghe and Piessens, 2009; Hamlen et al., 2006b; Hamlen and Jones, 2008; Sridhar and Hamlen, 2010a). Several authors (Hamlen and Jones, 2008; Chen, 2005; Bauer et al., 2005) have exploited the similarities between inlining and AOP style aspect weaving. Erlingsson and Schneider (Erlingsson and Schneider, 2000a) represents security automata directly as Java code snippets. This makes the resulting code difficult to reason about. The ConSpec policy specification language used here (Aktug and Naliuka, 2008) 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 (Erlingsson and Schneider, 2000b).

Aktug et al. (Aktug et al., 2009) formalized the analysis of inlined reference monitors and showed how to systematically generate correctness proofs for the ConSpec language, but restricted to sequential Java. Chudnov and Naumann (Chudnov and Naumann, 2010) propose a provably correct inliner for an information flow monitor. They prove security and transparency, but again restricted to a sequential programming language.

Edit automata (Ligatti et al., 2005; Ligatti, 2006) 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 Bauer and Ligatti in the Polymer tool (Bauer et al., 2005) which is closely related to Naccio (Evans and Twyman, 1999) and PoET/PSLang (Erlingsson and Schneider, 2000a).

Certified reference monitors has been explored by a number of authors, mainly through type systems, e.g. in (Skalka and Smith, 2004; Bauer et al., 2003; Walker, 2000; Hamlen

et al., 2006a; DeLine and Fähndrich, 2001), but more recently also through model checking and abstract interpretation (Sridhar and Hamlen, 2010c; Sridhar and Hamlen, 2010a).

The type-based Mobile system (Hamlen et al., 2006a) uses a simple bytecode extension to help managing updates to the security state. The use of linear types allows security-relevant actions to be localized 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 possible that Mobile could be adapted, at least to some forms of per session policies. As we show in the present paper, however, the synchronization needed to maintain a shared security state will have non-trivial effects. In particular the locking regime suggested in (Hamlen et al., 2006a) forces mutually exclusive access to security-relevant calls (it is *blocking*, in the terminology used below), potentially resulting in deadlocks.

In (Sridhar and Hamlen, 2010c; Sridhar and Hamlen, 2010a) Sridhar and Hamlen explore the idea of certifying inlined reference monitors for ActionScript using model-checking and abstract interpretations. The approach can handle a limited range of inlining strategies including non-trivial optimizations of inlined code. It is, however, restricted to sequential code and to non-recursive programs. Although the certification process is efficient, the analysis has to be carried out by the consumer.

The impact of multithreading has so far had limited systematic attention in the literature. There are essentially two different strategies, depending on whether or not the inliner is meant to block access to the shared security state during security relevant events such as API method calls. In the present paper we focus attention on the non-blocking strategy, which is the most relevant case in practice. In an earlier paper (Dam et al., 2010) we have examined the blocking strategy. In that case transparency is generally lost, as the inliner may introduce synchronization constraints that rule out correct executions that would otherwise have been possible. However, the blocking inlining strategy is not acceptable in practice as it may cause uncontrollable performance degradation and deadlock which motivates our attention to the non-blocking case in this paper.

The present paper is an extended and completely rewritten version of (Dam et al., 2009). In that paper the main results concerning inlineability and race-free policies were presented. This version contains a more thorough and self-contained presentation of the policy framework, rewritten and restructured proofs, and a completely rewritten presentation of the inliner. New material is the sections on case studies and evaluation, and on certification.

1.2. *Overview of the Paper*

The rest of this paper is structured as follows: We start by describing the JVM model that we adopt (Section 2) and the syntax and semantics of the security policies we consider in the paper (Section 3). We then define the notion of correct (secure, transparent and conservative) reference monitor inlining (Section 4) and show that these correctness criteria cannot be met for the programs and policies previously presented (Section 5). An

alternative, weaker correctness criterion, is presented (Section 6) together with an inlining algorithm that satisfies this criterion (Section 7). We then report on our experience with our implementation in five case studies (Section 8). Finally we present an approach for certifying an inlined reference monitor (Section 9) and present our conclusions and future work (Section 10).

2. Program Model

Our study is set in the context of multithreaded Java bytecode. We assume that the reader is familiar with Java bytecode syntax and the JVM. In this section we give an overview of our program model and discuss the semantics of the monitorable API calls.

Table 1 provides an overview of the structure of bytecode programs and JVM configurations. Details and transition semantics for the relation, $\rightarrow$, for key instructions and configuration types are given in the appendix.

Java Bytecode Programs

$Prg : c \rightarrow Class$	(programs)
$c \in String$	(class identifiers)
$Class ::= (m \rightarrow M, f^*)$	(class definitions)
$m \in String$	(method identifiers)
$M ::= (\iota^+, H^*)$	(method definitions)
$\iota \in Insn$	(instructions)
$f \in String$	(field identifiers)
$H ::= (\ell_b, \ell_e, \ell_t, c)$	(exception handler)
$\ell \in \mathbb{N}$	(program labels)

JVM Configurations

$C ::= (h, \Lambda, \Theta)$	(configurations)
$h : ((o \times f) \cup (c \times f)) \rightarrow Val$	(heap)
$o \in \mathbb{N} \cup \{null\}$	(references)
$Val ::= o \mid v$	(values)
$v \in byte \cup short \cup int \cup long \cup$	(primitive values)
$\quad float \cup double \cup boolean \cup char$	
$\Lambda : o \rightarrow tid$	(lock map)
$tid \in \mathbb{N}$	(thread identifiers)
$\Theta : tid \rightarrow \theta$	(thread config. map)
$\theta \in R^*$	(thread configuration)
$R ::= (c.m, \mathrm{pc}, s, l) \mid (o)$	(activation record)
$\mathrm{pc} \in \mathbb{N}$	(program counter)
$s \in Val^*$	(operand stack)
$l : \mathbb{N} \rightarrow Val$	(local variable store)

Table 1. *JVM Programs and configurations.*

2.1. *API Method Calls*

We are interested in security policies as constraints on the usage of external (API) methods. To this end we assume a fixed API, as a set of classes disjoint from that of the client program, for which we have access only to the signature, but not the implementation, of its methods. We therefore represent API method activation records specially. When an API method is called in some thread a special API method stack frame is pushed onto the call stack, as detailed in the appendix. The thread can then proceed by returning or throwing an exception. When the call returns, an arbitrary return value of appropriate type is pushed onto the caller's operand stack; alternatively, when it throws an exception, an arbitrary, but correctly typed exceptional activation record is placed on the call stack. Since this model makes no assumptions about the behavior of API methods, our results hold for all (correctly typed) API implementations. This semantics does not make any provisions for call-backs. How to extend inlining to call-backs is discussed in the conclusion.

It is essential that we perform API calls in two steps, to correctly model the fact that API calls are non-atomic in a multithreaded setting.

To support thread creation there is a distinguished API method that has, besides the standard effect of an API call discussed above, an additional side effect of creating a new thread in the configuration.

To refer to API calls and returns we use labelled transitions. Transition labels, or *actions*, α come in four variants to reflect the act of invoking an external method (referred to as a *pre-action*), returning from an external method normally or exceptionally (referred to as a *normal* or *exceptional post-action*), or performing an internal, not directly observable computation step. Actions have one of the following shapes:

— $(tid, c.m, o, v)^\uparrow$ represents the invocation of API method $c.m$ on object o with arguments v by thread tid.

— $(tid, c.m, o, v, r)^\downarrow$ similarly represents the normal return of $c.m$ with return value r.

— $(tid, c.m, o, v, t)^\Downarrow$ represents the exceptional return of $c.m$ with exception object (of class `Throwable`) t.

— τ represents an internal computation step.

We write $C \xrightarrow{\alpha} C'$ if either $\alpha = \tau$ and $C \to C'$, or $\alpha \neq \tau$ and C' results from C by the action α according to the above non-deterministic semantics. Refer to the appendix for details.

2.2. *Executions, Traces*

An *execution* of a program *Prg* is a finite or infinite sequence of configurations $E = C_0 C_1 \ldots$ where C_0 is an initial configuration, and for each pair of consecutive configurations we have $C_i \xrightarrow{\alpha_i} C_{i+1}$, such that E is compatible with the happens-before relation as defined by JLS3 (Gosling et al., 2005). The initial configuration consists of a single thread with a single, normal activation record with an empty stack, no values for local variables, with the main method of *Prg* as its current method and with $pc = 1$.

Since we are interested in inliners that are independent of implementation details

concerning e.g. scheduling, memory management and error handling we do not make any distinctions between executions that are allowed by the JLS3 memory model and executions that are possible for an actual implementation. The *trace* of E, $\omega(E)$, is the sequence $\alpha_0\alpha_1\ldots$ with τ actions removed, and $\mathcal{T}(Prg) = \{\omega(E) \mid E$ is an execution of $Prg\}$. In this paper we restrict attention to traces T that are realizable, in the sense that $T = \omega(E)$ for some execution E.

3. Security Policies

We study security policies in terms of allowed sequences of API method invocations and returns, as in a number of previous works, cf. (Erlingsson and Schneider, 2000a; Bauer et al., 2005; Aktug and Naliuka, 2008; Vanoverberghe and Piessens, 2009; Aktug et al., 2009; Dam et al., 2010). Our work is based on a slight extension of the ConSpec policy specification language (Aktug and Naliuka, 2008). We briefly present our dialect of ConSpec here for completeness.

ConSpec is similar to Erlingsson's PSlang (Erlingsson and Schneider, 2000a), but for tractability it describes conditionals and state updates in a small purpose-built expression language instead of the object language (Java, for PSLang) itself. ConSpec policies represent security automata by providing a representation of a security state together with a set of clauses describing how the security state is affected by the occurrence of a control transfer action between the client code and the API. A control transfer can be either an API method invocation, or a return action, either normal or exceptional. ConSpec proper allows for both per-object, per-session, and per-multisession policies. In this paper we work exclusively with per-session policies which is the case most interesting in practice.

3.1. *ConSpec Policy Syntax*

A ConSpec policy $\mathcal{P}$ consists of a security state declaration of the shape

$$\texttt{SECURITY STATE } Type_1 \ s_1, \ldots, Type_n \ s_n; \tag{1}$$

together with a list of rules. For simplicity, we require that the initial values for the security state variables are the default initial values for their corresponding Java types.

A *rule* defines how the security automaton reacts to an API method call of a given signature. Rules have the following general shape:

$$\begin{aligned} modifier \ [Type \ y \ \texttt{=}] \ c.m(Type_1 \ x_1, \ldots, Type_n \ x_n) \ [\texttt{ON } z] \\ \texttt{PERFORM } G_1 \ \texttt{-> \{ } F_1 \ \texttt{\}} \ldots G_m \ \texttt{-> \{ } F_m \ \texttt{\}} \ [\texttt{ELSE \{ } F \ \texttt{\}}] \end{aligned} \tag{2}$$

where *modifier* is either `BEFORE`, `AFTER` or `EXCEPTIONAL`, $Type, Type_1, \ldots, Type_n$ are the return and argument types of $c.m$ and G_i and F_i are guards and update statements respectively. `BEFORE` rules refer to pre-actions, and `AFTER` and `EXCEPTIONAL` rules to normal and exceptional post-actions respectively. The method signature following the event modifier specifies the method that the rule applies to. If the policy has a rule defined for a method (of a given signature, of a given modifier type), the method is said

```
SECURITY STATE  String requestorURL,  String requestedFile;

BEFORE BluetoothToolkit.sendFile(String destURL, String file)
    PERFORM
        requestorURL.equals(destURL) &&
        requestedFile.equals(file) -> { }

AFTER int reply = JOptionPane.showConfirmDialog(String query)
    PERFORM
        reply != 0 && goodFileQuery(query) -> {
            requestedFile = queryFile(query);
            requestorURL = queryRequestor(query)
        } ELSE { }
```

Fig. 1. A security specification example written in ConSpec.

to be *security relevant* and we refer to invocations and returns of this method as *security relevant actions*. For instance, if a `BEFORE` rule for method $c.m$ of a given signature is present then invocations of $c.m$ of that signature are security relevant, but if no `AFTER` rule is present, normal returns are not regarded as security relevant. There is at most one rule per method defined for each of the three event modifiers. The return value specification is absent for `BEFORE` rules. Each *clause* of the shape G_i -> { F_i }, or the clause `ELSE` { F } expresses a (conditional) update of the security state in the obvious way. The `ELSE` clause is syntactic sugar for a clause with a constantly true guard. The callee qualifier `ON` z and the `ELSE` clause are both optional except for `AFTER` and `EXCEPTIONAL` rules for which the `ELSE` clause is required. Hence a policy can never forbid a return from an API method.

The syntax of the guards G_i and update expressions, F_j and F are only described by example in this paper. Additional examples are given in Section 5. The syntax details are not critical. The only requirements are that expressions are side-effect free and that the expressions allow verification conditions to be efficiently generated. Currently this is an unchecked obligation of the policy-writer but can of course be enforced by restricting the use of methods to an allowed subset of API methods. Guards and update expressions may refer to the state variables, argument and return value variables and the callee variable. Guards are evaluated top to bottom, in order to obtain a deterministic semantics. For the first guard that evaluates to true, the corresponding update expression is executed. If no guard evaluates to true (and no `ELSE` clause is present) the rule is not allowed to fire. This indicates a security violation and program execution must be terminated.

Example 1. The policy in Figure 1 states that the program has to ask the user for permission each time it intends to send a file over Bluetooth. The specification has two security relevant methods, `JOptionPane.showConfirmDialog` and `BluetoothToolkit.send-File`. The specification uses the following three helper functions which we leave undefined:

— `goodFileQuery(query)` returns true iff `query` is a well formulated file send query, for instance because it matches a predefined pattern.

```
SECURITY STATE Set<Thread> initialized = new HashSet<Thread>();

BEFORE C.initialize() PERFORM
    !initialized.contains(Thread.currentThread()) -> {
        initialized.add(Thread.currentThread());
    }
```

Fig. 2. Accessing the current thread identifier in ConSpec.

— `queryRequestor(query)` and `queryFile(query)` returns the requestor and file substrings of `query` respectively.

Example 2. The policy in Figure 2 expresses that `C.initialize` can only be invoked once for each thread.

3.2. *ConSpec Semantics*

A ConSpec policy $\mathcal{P}$ specifies a deterministic automaton (Q, Σ, δ, q_0), explained below, which observes an execution of some client program and changes state, and potentially aborts, according to the policy specification. The details are straightforward. Assume an execution $E = C_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{n-1}} C_n$. The initial state q_0 is obtained by initializing the security state of $\mathcal{P}$ to its default, using, if necessary, a local heap. The alphabet Σ is the set of observable actions. The state space Q is the set of all type safe assignments to the security state variables. Having reached the i'th configuration of E with automaton state q_i, if $\alpha_i = \tau$ or if the action is not security relevant (of the given modifier type) the $i + 1$:th state is q_i as well. Otherwise the relevant rule is extracted, variables are bound as indicated above, a matching guard clause is identified, and the first matching update is enacted to compute q_{i+1}, and if no matching guard is found, $\omega(E)$ is rejected. If $\omega(E)$ is not rejected it is accepted, and if the traces of all executions of a program *Prg* are accepted by (the automaton determined by) $\mathcal{P}$, *Prg* is said to *adhere to* $\mathcal{P}$.

4. Reference Monitor Inlining

A reference monitor inliner (for short just inliner) is a function $\mathcal{I}$ that for each policy $\mathcal{P}$ and program *Prg* produces a program $\mathcal{I}(\mathcal{P}, Prg)$ with embedded policy checking functionality.

Our program model makes a clear distinction between the (untrusted) program, and the (trusted) API that it interacts with, and inliners are limited to rewriting the program. This may seem to limit the applicability of our model, as some existing inliners do rewrite the Java Platform API implementation. However, the reader should keep in mind that what we call the API in our model does not necessarily have to map on the Java Platform API. Any inliner has to make a choice as to what part of the system can be rewritten and what remains unchanged. In our model, this is what defines the boundary between program and API. Existing inliners make different choices as to where they draw this boundary: some can rewrite all Java bytecode (including Java Platform API methods that

are themselves implemented in Java). For such inliners the API of our model covers only the natively implemented methods. Other inliners will only rewrite application classes and leave the entire Java Platform API untouched. For such inliners the API of our model covers the entire Java Platform API. If an inliner were also to rewrite the native method implementations, then our model is not directly applicable, since we only model Java bytecode. But a similar model where the program consists of assembly code and the API consists of system calls could be built and would reveal the same limitations as the one we discuss in this paper: the limitations are fundamental.

One assumption that does limit the applicability of the model is the fact that we assume that API method invocations and returns are good abstractions of the security relevant actions that policies want to talk about. In other words, the limitations on enforceable policies that we identify in this paper are only applicable to policies that talk about API method invocations and returns, where the API is defined as above: the boundary of the part of the system that can be rewritten. The implementation of an API method is trusted to achieve exactly the effect that the policy writer wants to talk about. Hence we do not consider calls from within the implementation of an API method to other API methods.

Another consequence of the model is that an inliner can never prevent an API method from returning: inlined code can only be executed after the call has returned. This is why post-actions are required to always be enabled in ConSpec.

4.1. *Inlining Correctness Properties*

There are three correctness properties of fundamental interest (cf. (Ligatti, 2006; Hamlen et al., 2006b)), namely security, conservativity and transparency.

Security, arguably the most important property of an inliner, states that all possible traces of the inlined program should be compliant with the policy provided to the inliner.

Definition 1 (Security). An inliner $\mathcal{I}$ is *secure* if, for every program *Prg* and policy $\mathcal{P}$, every trace of the inlined program $\mathcal{I}(\mathcal{P}, Prg)$ adheres to $\mathcal{P}$, i.e.

$$\mathcal{T}(\mathcal{I}(\mathcal{P}, Prg)) \subseteq \mathcal{P}.$$

Transparency states that the policy adherent behavior of the client program should be preserved by the inliner.

Definition 2 (Transparency). An inliner $\mathcal{I}$ is *transparent*, if for every policy $\mathcal{P}$ and program *Prg*, each trace of *Prg* that adheres to $\mathcal{P}$ is also a trace of the inlined program, i.e.

$$\mathcal{T}(Prg) \cap \mathcal{P} \subseteq \mathcal{T}(\mathcal{I}(\mathcal{P}, Prg)).$$

Conservativity states that no behavior should be added to the original program.

Definition 3 (Conservativity). An inliner $\mathcal{I}$ is *conservative* if, for every program *Prg* and policy $\mathcal{P}$, every trace of the inlined program $\mathcal{I}(\mathcal{P}, Prg)$ is a trace of *Prg*, i.e.

$$\mathcal{T}(\mathcal{I}(\mathcal{P}, Prg)) \subseteq \mathcal{T}(Prg).$$

Other correctness properties have been proposed, such as the concept of *strong conservativity*, which was used in (Dam et al., 2010). This correctness criteria refines the notion of conservativity and forbids arbitrary truncation of the traces. Since this is mostly useful for the case of a blocking inliner to account for the necessary loss of transparency, cf. (Dam et al., 2010), we do not discuss it further in this paper.

5. Limitations of Inlining in a Multithreaded Setting

In this section, we show that the traditional correctness criteria for inlined monitors are too strong in a multithreaded setting. While it is possible to securely and transparently enforce any policy specified as explained in Section 3 by an *external* monitor implemented as part of the JVM, it is impossible to do this with an inlined monitor without taking specificities of the API implementation and/or virtual machine into account.

One of the key differences between an external monitor and a monitor inlined in the client program is the ability to affect the behavior of a thread executing within an API-method. As opposed to an external reference monitor, an inlined reference monitor cannot in general control the scheduling of such a thread, and this affects the enforceability of certain policies.

Consider the policy in Figure 3. This policy states that $c.n$ may only be called when ok has been set to true, that is, after $c.m$ has been called (but not necessarily returned). So the trace $T_1 = (tid, c.m, o, v)^\uparrow, (tid', c.n, o', v')^\uparrow$ is allowed by the policy, but the trace $T_2 = (tid', c.n, o', v')^\uparrow, (tid, c.m, o, v)^\uparrow$ is not. Now consider a program whose traces include both T_1 and T_2, for instance the one shown in Figure 5. For an inliner to exclude trace T_2 from this program (but keep the trace T_1), it could either exploit some implementation-dependent knowledge of the virtual machine, or else it would have to introduce a happens-before relation between $(tid, c.m, o, v)^\uparrow$ and $(tid', c.n, o', v')^\uparrow$. In the latter case we note that there is no way such a happens-before relation can be enforced by the inliner since, by convention, after the call has been made, the control lies within the API method which is not to be altered. In terms of the formal semantics of API calls given in Section 2.1 the former case is also ruled out. To lift this to practical virtual machines let us say that a correctness property is *uniform* if it holds for all API implementations, including the fully nondeterministic one of Section 2.1. Using the API semantics of Section 2.1, the inlined program will either have both traces T_1 and T_2 (in which case the inliner is not secure) or it will have neither of the two traces (in which case the inliner is not transparent). We have thus shown:

Theorem 1. No inliner can be both uniformly transparent and uniformly secure for the policy $\mathcal{P}$ in Figure 3. □

Evidently, an inliner could "over-approximate" and guard the entire call to `c.m` by a lock and let the monitor release the lock after `c.m` has returned, but in that case the monitor would be enforcing the stronger policy shown in Figure 4 and prevent some traces that are allowable by the policy in Figure 3.

```
SECURITY STATE
    boolean ok = false;

BEFORE c.m() PERFORM
    true -> { ok = true; }

BEFORE c.n() PERFORM
    ok == true -> {}
```

Fig. 3. Example of policy which is not enforceable by inlining.

```
SECURITY STATE
    boolean ok = false;

AFTER c.m() PERFORM
    true -> { ok = true; }

BEFORE c.n() PERFORM
    ok == true -> {}
```

Fig. 4. Example of policy enforceable by inlining.

```
class SomeClass {
    public static void main(String[] args) {
        new Thread() {
            public void run() { c.m(); }
        }.start();
        c.n();
    }
}
```

Fig. 5. A program invoking *c.m* and *c.n* in a non-deterministic order.

6. Race-free Policies

Generalizing from the example in Figure 3, the key issue is that no client program (not even after inlining) can arbitrarily constrain the set of observable traces. Given a certain trace of observable actions, in general there will be permutations of that trace that are also possible traces of the client program no matter what synchronization efforts the client performs. These permutations that are always possible are captured by the notion of *client-order preserving* permutations.

Definition 4 (Client-order Preserving Permutation). A permutation $\pi(T)$ of a trace T of observable actions is *client-order preserving* if, for all i and j such that $i < j$ and (a) T_i and T_j take place on the same thread, or (b) T_i and T_j correspond to a post- resp. pre-action, then $\pi(i) < \pi(j)$.

The intuition is the following: the client can control pre-actions, and can only observe post-actions. If a pre-action takes place somewhere after a post-action, the client *could* have synchronized to ensure this ordering. The client cannot perform such synchronization for concurrent pre-actions or concurrent post-actions.

If a policy accepts a given trace, but rejects a client-order preserving permutation of the trace, then that policy is not securely and transparently enforceable by inlining a monitor in the client code. This is captured by the following definition:

Definition 5. A policy is *race-free* iff, for any trace T and any client-order preserving permutation T' of T, if T is allowed, then T' is allowed.

As an example, the policy in Figure 1 is race-free. As a broader class of examples consider the class of policies where the security state is a set of permissions, pre-actions require a permission to be present in this set and cause the permission to be removed, and post-actions restore the permission. Such policies are race-free. This can be checked for instance by using Proposition 2 below.

We show further that the class of race-free policies is a lower bound on the class of policies enforceable by inlining by constructing an inliner that is secure, transparent and conservative for this class of policies.

The following theorem shows that the bound is tight.

Theorem 2. No inliner can be uniformly secure and uniformly transparent for a non-race-free policy.

Proof. Let $\mathcal{P}$ be a non-race-free policy. It suffices to show that $\mathcal{P}$ is not enforceable for the fully non-deterministic semantics of Section 2.1. By definition there is a trace T of some program *Prg* which $\mathcal{P}$ accepts and a client-order preserving permutation T' of T which $\mathcal{P}$ rejects. Now for an inliner, $\mathcal{I}$, to be transparent, $\mathcal{I}(\mathcal{P}, Prg)$ has to admit the trace T. But, since a client-order preserving permutation respects the happens-before relations stipulated by any program, $\mathcal{I}(\mathcal{P}, Prg)$ must also admit the trace T', which means that $\mathcal{I}$ is not secure. □

A policy for which there exists a (uniformly) secure, transparent and conservative inliner is said to be (uniformly) *inlineable*. The corollary below follows immediately.

Corollary 1. The set of uniformly inlineable policies is a subset of the set of race-free policies.

Proof. Let $\mathcal{P}$ be an arbitrary uniformly inlineable policy. By definition there exists a uniformly secure and transparent inliner for $\mathcal{P}$, thus by Theorem 2, $\mathcal{P}$ must be race-free. □

An interesting question is how to decide if a policy is race-free. Using Lipton's moverness terminology (Lipton, 1975) we obtain the following:

Proposition 1. It is a necessary and sufficient condition for race-freedom that all pre- and post-actions occurring in different threads are right- resp. left-movers, in the set of allowed observable traces. (I.e., if a trace T is allowed, then swapping a pair of consecutive

actions α_1, α_2 in different threads where α_1 is a pre-action or α_2 is a post-action yields an allowed trace.)

Proof. Such swappings generate the client-order preserving permutations. $\square$

In particular, if such swappings always have the same effect on the policy state, we know the policy is race-free:

Proposition 2. The following is a sufficient condition for race-freedom. For any state q_1 of the security automaton corresponding to a given policy, and for any pre-action α_1 and post-action α_2 with different thread identifiers, if $\delta(\delta(q_1, \alpha_1), \alpha_2) = q_2$ then $\delta(\delta(q_1, \alpha_2), \alpha_1) = q_2$.

Proof. These conditions imply the conditions from Proposition 1. $\square$

Sufficient syntactical criteria for the conditions of Proposition 2 are easily identified. For example, for the common case where the security state is a set of permissions, a sufficient requirement is that pre-actions only consume permissions from the set, and post-actions only add permissions.

6.1. *Discussion*

Are there interesting or practically relevant policies that are not race-free? A policy that is not race-free imposes constraints not only on the client program, but also on the API implementation and/or the scheduler. Hence, we argue that such policies do not make sense. Even if an enforcement mechanism (such as an external execution monitor) could enforce the policy, the result of the enforcement is most likely not in line with what the policy writer intended to express. Policies impose constraints on API method invocations because of the effects (such as writing a file, reading from the network, activating a device, . . .) that these API implementations have. A policy such as the one in Figure 3 intends to specify that initiation of one effect should come after the initiation of another effect. But without further information about the API implementations and the operation of the scheduler, there is no guarantee that enforcing this ordering on the API invocations will also enforce this ordering on the actual effects.

In other words, the race in the policy that makes it impossible for an inliner to enforce the policy, also makes it impossible to interpret method invocations soundly as initiations of effects.

Hence, a policy that is not race-free either indicates a bug in the policy (for instance, the policy writer intended to specify the policy in Figure 4 instead of the policy in Figure 3 – an easy mistake to make as in the single-threaded setting both policies are equivalent), or it is an indication of a misunderstanding of the policy writer (for instance the policy writer considers the start of the API method invocation as a synonym of the start of the effect the API method implements). Jones and Hamlen (Jones and Hamlen, 2010) make a similar observation for a different class of policies that is hard to enforce with inlining.

As a consequence, the practicality of inlining as an enforcement mechanism is not at stake, and detection of races in policies is useful as a technique to detect bugs in policies.

7. Race-free Policies Are Inlineable

In this section we show that race-free policies can be enforced by IRM, by giving an inlining scheme that is secure, conservative and transparent for race-free policies. (From this point onward we restrict attention to the API semantics of Section 2.1 in order to eliminate from consideration pathological virtual machines that may introduce implementation-dependent errors or, e.g., manipulate the scheduler in non-standard ways). For sequential Java a correct inlining scheme is already known to exist. In this section we show that the race-free policies is the maximal set of policies for which correct inlining is possible.

The state of the IRM might possibly be updated by several threads concurrently. The updates to this state must therefore be protected by a global lock. A key design choice is whether to keep holding this lock during the API call, or to temporarily release the lock during the call and reacquire it after the call has returned. In the former case we say that the inliner is *blocking*, and in the latter we say it is *non-blocking*.

The first choice (locking across calls) is easier to prove secure, as there is a strong guarantee that the updates to the security state happen in the correct order. The implications of this design choice was examined in (Dam et al., 2010). The problem is that a blocking inliner can introduce deadlocks in the inlined program and it is thus not transparent. Consider for instance an API with a barrier method B that allows two threads to synchronize as follows: When one thread calls B, the thread blocks until the other thread calls B as well. Suppose this method is considered to be security-relevant, and the inliner, to protect its state, acquires a global lock while performing each security-relevant call. For a client program that consists of two threads, each calling B and then terminating, the inliner will introduce a deadlock, as one thread blocks in B while the other thread blocks on the global lock introduced by the inliner.

Even if it does not lead to deadlock, acquiring a global lock across a potentially blocking method call can cause serious performance penalties. For this reason, our algorithm releases the lock before calling an API method. In fact, our algorithm ensures that the global lock is only held for very short periods of time.

It is worth emphasizing that the novelty in this section is not the inlining algorithm itself: The algorithm is similar to existing algorithms developed in the sequential setting and the locking strategy is relatively straightforward. The contribution, rather, is the proof that the notion of race-free policies gives an exact characterization of the class of policies enforceable on multithreaded Java-like programs by a non-blocking inlining scheme.

7.1. *Inlining Algorithm*

In order to enforce a policy through inlining, it is convenient to be able to statically decide whether a given policy clause applies to a given call instruction. Therefore we

impose the restriction on programs that they should have *simple call matching*, namely that for all security-relevant methods $c.m$, an `invokevirtual` $d.m$ call is bound at run time to method $c.m$ if and only if $d = c$. Essentially, this means that we ignore all issues concerning inheritance and dynamic binding. These concerns are orthogonal to the results of this paper, and it has been described elsewhere how to deal with them (Vanoverberghe and Piessens, 2009; Aktug et al., 2009).

The inliner, $\mathcal{I}_{Ex}$, takes a policy with security state definition and event rules of the shapes (1) and (2) (see Section 3) and applies it to a Java bytecode program. The inliner uses static fields s_i of type $Type_i$ of an auxiliary class `SecState` to store the shared security state, as in the ConSpec security state declaration (1). (In general a unique name needs to be chosen for the security class itself, to allow the inliner to be iteratively applied). We assume for simplicity that rules are present for each of the three rule types `BEFORE`, `AFTER` and `EXCEPTIONAL`, and we use $G_{i,t}$, F_t, $F_{i,t}$, $t \in \{b, a, e\}$ to indicate the corresponding guard and update blocks in (2). The compilation of guard clauses and update blocks into bytecode is well understood and we simply assume that they are compiled into basic blocks $eval(G_{i,t})$, $eval(F_t)$, $eval(F_{i,t})$ that behave as required. In particular, the callee is extracted from the top of the stack, arguments from stack elements $1, \ldots, n$, security state variables from corresponding fields of the `SecState` class, and the calling thread identifier is extracted using `Thread.currentThread`. The inliner then replaces each instruction L : `invokevirtual` $c.m$ of arity n where $c.m$ is security-relevant by bytecode implementing the pseudo-code in Figure 6. The inliner locks the security state by acquiring the lock associated with the *SecState* class, and stores callee and arguments to the method call for use in event handler code using fresh local variables. The security state lock is taken by executing first `ldc SecState` and then entering the monitor. The use of a static class for the security state makes it easy to determine statically that locks taken or released outside the inlined code snippets do not affect the security state lock. The lock is released just prior to invocation of the inlined call, and retaken after return. Each piece of event code evaluates guards by reference to the security state and the stored arguments, and updates the state according to the matching clause, or exits, if no matching clause is found. Thus, if F_b (i.e. the `ELSE`-clause) is absent the block at *beforeEnd* is replaced by a jump to *exit*.

If no `BEFORE` rule is present, evaluation of the `BEFORE` guards and update clauses is evidently not performed. Arguments and callee are still stored in local variables and restored before the method is called, as arguments and callee may be needed for evaluating an `AFTER` or `EXCEPTIONAL` rule.

The exception handler array is modified by adding the entries in Figure 7 and adding $done - L - 1$ to all offsets above L in the original handler. Exceptions emanating from the call to $c.m$ are routed to the the inlined handler at $excG_1$. After processing of `EXCEPTIONAL` events the security state is unlocked and the exception rethrown. Exceptions caused by inlined instructions are routed to *exit*.

One complication is the possibility of internal exceptions. The Java Virtual Machine Specification (Lindholm and Yellin, 1999) allows a JVM to throw an `InternalError` or `UnknownError` exception at any time whatsoever. This means that, e.g. when the JVM attempts to compile a piece of bytecode about to be executed by a thread to machine

Inlined label Instruction		Inlined label Instruction
L:	*lock* SecState	ifeq *afterElse*
	store arguments	$[eval(F_{m,a})]$
	store callee	goto *afterEnd*
$beforeG_1$:	$[eval(G_{1,b})]$	*afterElse*: $[eval(F_a)]$
	ifeq *beforeG₂*	*afterEnd*: *restore return value*
	$[eval(F_{1,b})]$	*unlock* SecState
	goto *beforeEnd*	goto *done*
	$\vdots$	
$beforeG_m$:	$[eval(G_{m,b})]$	$excG_1$: *lock* SecState
	ifeq *beforeElse*	*store exception*
	$[eval(F_{m,b})]$	$[eval(G_{1,e})]$
	goto *beforeEnd*	ifeq $excG_{2,e}$
beforeElse:	$[eval(F_b)]$	$[eval(F_{1,e})]$
		goto *excEnd*
beforeEnd:	*restore callee*	$\vdots$
	restore arguments	$excG_m$: $[eval(G_{m,e})]$
	unlock SecState	ifeq *excElse*
invoke:	invokevirtual *c.m*	$[eval(F_{m,e})]$
invokeDone:	*lock* SecState	goto *excEnd*
	store return value	*excElse*: $[eval(F_e)]$
$afterG_1$:	$[eval(G_{1,a})]$	*excEnd*: *restore exception*
	ifeq *afterG₂*	*unlock* SecState
	$[eval(F_{1,a})]$	*excReleased*: athrow
	goto *afterEnd*	*exit*: iconst -1
	$\vdots$	invokestatic *System.exit*
$afterG_m$:	$[eval(G_{m,a})]$	*done*:

Fig. 6. The inlining replacement of L: invokevirtual *c.m*.

From	To	Target	Type
invoke	*invokeDone*	$excG_1$	*any*
L	*excReleased*	*exit*	*any*
exit	*done*	*exit*	*any*

Fig. 7. Exception handler array modifications

code but it does not have enough memory to store the machine code, it can throw an internal exception instead of having to terminate the entire program. Whereas internal exceptions are useful for JVM implementers, they cause complications for the design of our inliner. Specifically, for security, we must maintain the property that whenever no block of inlined code is being executed, the current security state matches the trace of security-relevant actions performed previously during the execution. If an internal exception were to cause control to exit a block of inlined code prematurely, this property would be violated. Therefore, we catch all exceptions that occur anywhere in the inlined code and, when any exception is thrown by any instruction other than the security-relevant call, we exit the program. Notice that this is secure and conservative, since we exit at a place where the original program does not exit. But in pathological cases (such as a JVM which chooses to randomly abort execution whenever a static class `SecState` is defined) transparency may fail. For this reason we assume below that the JVM is error-free, i.e. it never throws an internal exception.

7.2. *Correctness*

We first prove security, i.e. that for each program *Prg* and race-free policy $\mathcal{P}$, $\mathcal{T}(\mathcal{I}_{Ex}(\mathcal{P}, Prg)) \subseteq \mathcal{P}$. The basic insight is that race-freedom ensures that actions and monitor updates are sufficiently synchronized so that security is not violated. To see this we need to compare the observable actions of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ with the corresponding *monitor actions*, i.e. actions of the inlined code manipulating the inlined security state. We use the notation $mon(\alpha)$ for the monitor action corresponding to the observable action α. The monitor action $mon(\alpha)$ occurs at step $i \in [0, n-1]$ of the execution $E = C_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{n-1}} C_n$, if the instruction scheduled for execution at configuration C_i is `monitorexit`, corresponding to one of the unlocking events in Figure 6 for the action α. We refer to the points in E at which the monitor actions occur, as *monitor commit points*.

Depending on which case applies we talk of the monitor action $mon(\alpha)$ as a monitor pre-, normal monitor post-, or exception monitor post-action. Then the *extended trace of E*, $\tau_e(E)$, lists all *extended actions*—that is, non-τ actions and monitor actions—of E in sequence, and the *monitor trace* of E, $\tau_m(E)$, projects from $\tau_e(E)$ the monitor actions only. Let β range over extended actions.

Pick now an execution E of an inlined program $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$, and let $\tau_e(E) = \beta_0, \ldots, \beta_{n-1}$. Say that E is *serial* if in $\tau_e(E)$ there is a bijective correspondence between actions and monitor actions, and if each pre-action α is immediately preceded by the corresponding monitor action $mon(\alpha)$, and each post-action α' is immediately succeeded by its corresponding monitor action $mon(\alpha')$.

We first observe that monitor traces are just traces of the corresponding security automaton:

Proposition 3. Let E be an execution of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$. Then $\tau_m(E) \in \mathcal{P}$.

Proof. The locking regime ensures that all monitor actions, hence automaton state updates, are happens-before related. Since each thread updates the automaton state according to the transition relation, the result follows. □

Lemma 1. Assume that $\mathcal{P}$ is race-free. For any execution E of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ there exists a serial execution E' such that $\tau(E) = \tau(E')$.

Proof. Let E of length n be given as above. Note first that, by the happens-before constraints, the bijective correspondence must be such that pre-actions are preceded by their corresponding monitor actions, and vice versa for post-actions. We construct the execution E' by induction on the length m of the longest serial prefix of $\tau_e(E)$. If $n = m$ we are done so assume $m < n$. Say that β_{m-1} is produced by thread t. Note first that β_{m-1} can be either a pre-action or a monitor post-action as E' is serial, and that β_m can be either a post-action or a monitor pre-action. For the latter point assume for a contradiction that β_m is a pre-action. Then β_m must be produced by a thread $t' \neq t$, by the control structure of the inlining algorithm, Figure 6. The last action in $\tau_e(E')$ by thread t' must be a monitor pre-action $\beta_l = \text{mon}(\beta_m)$ for $0 \le l < m - 1$ and, as each action records the tid, $\beta_k \neq \beta_m$ for any $l < k < m - 1$. But then the extended trace $\beta_0, \ldots, \beta_{m-1}$ is not serial, a contradiction. The case where β_m is a monitor post-action is similar.

Now, if β_m is a post-action, say, then thread t is at one of the control points *invokeDone* or *excG$_1$*. Either $\text{mon}(\beta_m) = \beta_{m'}$ for some $m' > m$ or else thread t does not produce any extended actions in $\tau_e(E')$ after m. In the latter case it is possible to schedule $\text{mon}(\beta_m)$ directly, as the guards for post-actions are exhaustive. In the former case we need to also argue that all extended actions β_k for $m \le k$ and $k \neq m'$ remain schedulable, even after scheduling $\text{mon}(\beta_m)$ right after β_m. But this follows from the left-moverness of monitor post-actions with respect to both monitor actions, Proposition 1, and non-monitor actions on different threads.

If on the other hand β_m is a monitor pre-action $\text{mon}(\alpha)$. If $\beta_{m+1} = \alpha$ we are done. Otherwise β_{m+1} is a monitor action or non-monitor action of another thread, and regardless which, by rescheduling, β_m can be moved right until it is left adjacent to α. But this case can only apply a finite number of times at the end of which E' can be extended. This completes the proof. □

Inliner security is now an easy consequence.

Theorem 3 (Inliner Security). If $\mathcal{P}$ is race-free then $\mathcal{I}_{Ex}$ is secure, i.e. $\mathcal{T}(\mathcal{I}_{Ex}(\mathcal{P}, Prg)) \subseteq \mathcal{P}$.

Proof. Pick any execution E of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$. Use Lemma 1 to convert E to an execution E' with the property that $\tau(E) = \tau(E') = \tau_m(E') \in \mathcal{P}$ by Proposition 3 and since E' is serial. □

For conservativity, our proof is based on the observation that there is a strong correspondence between executions of an inlined program, and executions of the underlying program before inlining. From an execution of the inlined program, one can *erase* all the inlined instructions and the security state, and arrive at an execution of the underlying program. This is so since control entering one of the inlined blocks in Figure 6 at one of the labels L, *invokeDone*, or *excG$_1$* can only exit that block either through the corresponding labels *invoke*, *done*, or by rethrowing the original exception, or else by invoking

`System.exit`. Moreover, up to variables accessible only to the inlined code fragments, and provided `System.exit` is not invoked, the machine state at entry and at exit of each inlined block is the same. In this manner we can from an execution E of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ obtain an execution $erase(E)$ of Prg such that $\tau(E)$ is a prefix of $\tau(erase(E))$, and hence $\tau(E) \in \mathcal{T}(Prg)$. We refrain from elaborating the details and merely state:

Theorem 4. The inliner $\mathcal{I}_{Ex}$ is conservative. □

Transparency is slightly delicate as the JVM standard (Gosling et al., 2005) does not predicate the exact conditions under which a JVM is allowed to abort. Hence we need to assume that all executions allowed by JVM standard are indeed possible, and that no constraints are imposed on heap size etc., as in the abstract semantics of Section 2, which might otherwise affect execution in a way that could interfere with transparency. With this proviso, however, transparency is easily seen, by—so to speak—putting the argument for conservativity in reverse.

Theorem 5. The inliner $\mathcal{I}_{Ex}$ is transparent.

Proof. Consider an execution E of Prg such that $\tau(E) \in \mathcal{P}$. From E construct another execution E' of $\mathcal{I}_{Ex}(\mathcal{P}, Prg)$ by inserting inlined block executions similar to the way such block executions are erased in the proof of Theorem 4. This is possible for the same reasons erasure of these block executions is possible in the proof of Theorem 4, and since $\tau(E) \in \mathcal{P}$. Trivially, $\tau(E') = \tau(E)$ which suffices to conclude. □

Corollary 2. The race-free policies is the maximal set of inlineable policies.

Proof. Since $\mathcal{I}_{Ex}$ is secure, transparent and conservative for all race-free policies, we know that any race-free policies is by definition inlineable. The result then follows from Corollary 1. □

8. Case Studies

We have implemented an inliner that parses policies written in ConSpec and performs inlining according to the algorithm described in Section 7.1. This inliner has been evaluated in five case studies of varying characteristics. Case study descriptions and results are provided below. For detailed descriptions and case study applications and policies, we refer to the web page (Lundblad, 2010).

8.1. *Case Study 1: Session Management*

It is common for web applications to allow users to login from one network and then access the web page using the same session ID but with a different IP address from another network. Provided that the session ID is kept secret this poses no security problems. However, the session can be *hijacked* due to for instance predictable session IDs, session sniffing or cross-site scripting attacks (OWASP, 2010).

In this case study we examine a simple online banking application implemented using the Winstone Servlet Container and the HyperSQL DBMS. Users may login though an

HTML form, transfer money and logout. The session management is handled by the classes provided by the standard Servlet API (Apache Software Foundation, 2002).

To eliminate one source of session hijacking attacks the policy in this case study forbids a session ID from being used from multiple IP addresses. It does this by a) associating every fresh session ID with the IP address performing the request, and b) rejecting requests referring a known session ID performed from IP addresses not equal to the associated one.

The policy is implemented using a `HashMap` for storing the IP to session ID association, and monitors (and restricts) all invocations of the `HttpServlet.service` method.

8.2. *Case Study 2: HTTP Authentication*

In this case study we look at the HTTP authentication mechanism (Franks et al., 1999). This allows a user to provide credentials as part of an HTTP request. On top of this the Servlet API provides a security framework based on user roles. The access control of this setup is on the level of HTTP-commands, such as GET and POST. This is however too coarse-grained for some applications.

The application in this case study is the same as in case study 1, but here we focus on the administrative part of the web application. This part is protected by HTTP authentication and supports two roles: *Secretaries* and *administrators*. The intention is that secretaries should be allowed to *query* the database whereas administrators are allowed to also *update* the database.

The policy enforces this by making sure the application calls `HttpServletRequest.is-UserInRole` and that only users in the secretary role may invoke `java.sql.Statement.executeQuery` and only users in the administrator role may invoke `java.sql.Statement.executeUpdate`. Since these rules only apply for the administrative part of the web application the policy is implemented to check requests only if `request.getRequestURI().startsWith("/admin")` returns true. Furthermore, to prevent interference of multiple simultaneous requests, the policy state is stored in `ThreadLocal` variables.

8.3. *Case Study 3: Browser Redirection*

Following the example of Sridhar and Hamlen (Sridhar and Hamlen, 2010b) we examined an ad applet that, when being clicked on, redirects the browser to a new URL. The policy in this case states that the applet is only allowed to redirect the browser to URLs within the same domain as which the applet was loaded from.

The policy enforces this by asserting that URLs passed to `AppletContext.showDocument` have the same host as the host returned by `Applet.getDocumentBase()`.

8.4. *Case Study 4: Cash Desk System*

In this case study we monitor the behavior of a concurrent model of a cash desk system. The application stems from an ABS model that was developed for the HATS project (HATS, 2010). The policy keeps track of the number of sales in progress (by monitoring

invocations of `newSaleStarted()` and `saleFinished()`) and asserts that the number of ongoing sales is positive.

8.5. *Case Study 5: Swing API Usage*

The classes in the Java Swing API are not thread safe and once the user interface has been realized (`Window.show()`, `Window.pack()` or `Window.setVisible(true)` has been called) the classes may be accessed only through the event dispatch thread (EDT). This constraint is sometimes tricky to adhere to as it is hard to foresee all flows of a program and whether or not some code will be executed on the EDT or not.

In this case study we monitor the usage of the Swing API in a large (68 kloc), off-the-shelf, drawing program called JPicEdt (version 1.4.1_03) (Reynal, 2010). The inlined monitor has two states: realized and not realized and the policy states that once realized, a Swing method may only be called if `EventQueue.isDispatchThread()` return true.

This case study demonstrates how the inliner can be useful, not only in a security critical setting, but also during testing. The inlined reference monitor revealed three violations of the policy and by letting the monitor print the stack trace upon a violation we managed to locate and patch the errors.

8.6. *Results*

A summary of the case studies is given in Table 2. Benchmarks were performed on a computer with a 1.8 GHz dual core CPU and 2 GB memory. The runtime overhead due to inlining was measured for the web application case studies (CS1 and CS2) and for the Swing case study (CS 5). The runtime overhead for the web application was based on a roughly one minute long stress test and for the Swing application we measured the startup time (the time required to construct the user interface).

Case Study	ConSpec clauses	Size before inlining (kB)	Size after inlining (kB)	Size increase (%)	Security relevant calls	Inlining time (s)	Runtime Overhead (%)
CS1 (Sessions)	1	532.7	533.1	0.08	1	2.47	0.44
CS2 (HTTP Auth.)	4	532.7	535.6	0.54	12	2.66	0.87
CS3 (Redirection)	2	27.5	28.2	2.41	1	0.18	n/a
CS4 (Cash Desk)	2	652.9	654.0	0.17	2	2.52	n/a
CS5 (Swing)	249	1888.6	2140.7	13.35	1038	26.68	11.27

Table 2. *Quantative results of the case studies.*

9. Certification

Monitoring is essentially a tool for quality assurance: By monitoring program execution we are able to observe actions taken by a program and intervene if a state of affairs is discovered which we for some reason are unhappy with. By inlining we can make this tool available for developers as well, for instance to enforce richer, history-dependent access control than what is allowed in the current, static sandboxing regime.

However, the code consumer may not necessarily trust the developer (code producer) to enforce the consumer's security policy. Moreover, different consumers may want to enforce different security policies. In this section we turn to the issue of *certification*, that is, we ask for an algorithm, a *checker*, by which the recipient of a piece of code can convince herself that the application is secure. To support efficient verification, the code producer can ship additional metadata with the code, for instance (elements of) a proof, following the idea of Proof-Carrying Code (PCC) (Necula, 1997). This metadata will be called a *certificate*, not to be confused with the concept with the same name used in public-key cryptography.

The scenario we want to support is the following (a classic PCC scenario):

1. A code producer develops an application, and ensures that it complies with the *producer policy* by inlining a corresponding monitor. This producer policy is developed with the intention that it will cover all the security concerns of potential consumers of the application, but of course these consumers do not necessarily trust the producer for this.

2. Various code consumers want to run the application. Before doing so, each consumer will check that the code complies with his or her *consumer policy*. (Each consumer may have a different policy.)

3. In order to help a consumer with this check, the producer ships a *certificate* together with the code. The certificate will contain a proof of the fact that the code complies with the consumer policy.

4. The code consumer uses a *checking* algorithm which checks if the application complies with his consumer policy. This checking algorithm takes as (untrusted) input the application code and the certificate.

We outline an approach for building a checker that can verify the security property of IRMs inlined using techniques similar to the algorithm we discussed in this paper. The contribution of this section is that we show that, for this inlining approach, a checker for multithreaded Java programs can be built using established program verification techniques based on sequential Java.

9.1. *Assumptions about the inlined code*

The checking algorithm in this section is designed for a class of inliners that (1) are non-blocking, i.e. they do not lock the security state across security relevant API calls, and (2) use one global lock to protect the inlined security state.

More concretely, let us assume that the security state is kept in static fields of a

designated `SecState` class, and that the `SecState` class object is used to lock the security state. The actual inlined code then operates in phases:

1   A *neutral* phase (N), where the `SecState` lock is not held. If all threads are in this N state, then the inlined security state is in sync with the history of security relevant actions encountered so far.

2   A *locked before* phase (LB), where the inliner is updating its state in anticipation of an upcoming security relevant call.

3   An *unlocked before* phase (UB), where things might be happening between the inlined check and the actual call. The inlined security state has been updated already, but the actual security relevant action has not yet happened.

4   A *calling* phase (C) where the actual security relevant call is executing.

5   An *unlocked after* phase (UA), where things might be happening between the (normal) return of the call, and the inlined security state update.

6   A *locked after* phase (LA), where the inliner is updating its state in response to a successfully returned security relevant call.

7   Similar *unlocked exceptional* and *locked exceptional* phases, to deal with exceptional returns of the security relevant method invocation. These are similar to the UA and LA phases, and we do not discuss them further in this section. Extending the results in this section to deal with exceptional returns of security relevant calls is straight-forward.

Notice that, with the inliner of Figure 6, it appears that no instructions are actually executed during the UB and UA phases. This is, however, not entirely accurate: When the inliner is applied iteratively, say twice in succession, the instructions executed in the locked phases of the second inlining will appear as instructions in the unlocked phases for the first inlining. In fact, we can allow arbitrary code to be present in the unlocked phases, as long as it does not interfere with the inlined state. This allows a wider class of inliners to be supported than the one introduced above. One such example is briefly discussed in the conclusions.

A key part of the checking algorithm is to recognize these phases. Once the phases are recognized, an approach similar to the one taken in (Aktug et al., 2009) for sequential Java can be enacted.

To assist the checker in identifying the phases, the certificate contains the following information: For each bytecode instruction in the program that performs a security relevant method invocation, the code producer should include in the certificate a tuple $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$, where $c'.m'$ is the name of the method containing the call, and the other elements of the tuple are labels in the method body of $c'.m'$:

— L_{lb} indicates where the LB phase starts,

— L_{ub} indicates where the LB phase ends and the UB phase starts,

— L_{call} indicates where the calling phase C starts and ends. Recall that in our semantics, API calls happen in two steps. The first step initiates the calling phase, and the second step ends it, and starts the UA phase.

— L_{la} indicates where the UA phase ends and the LA phase starts.

— Finally, L_n indicates where the LA phase ends and the inliner returns to the neutral phase.

A first part of the checking algorithm verifies, based on the above information, whether the code complies with the assumptions we make about the inlining process. The example inliner $\mathcal{I}_{Ex}$ that we proposed in Section 7 will pass this check.

Check 1. For each tuple, $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$, in the certificate, perform the following checks:

— The L_{lb} and L_{la} labels point to a `ldc SecState` instruction, followed by a `monitor-enter`.
— The L_{ub} and L_n labels point to a `monitorexit` instruction preceded by a `ldc SecState`.
— The labels L_{lb}, L_{ub}, L_{call}, L_{la}, L_n occur in this order in the method body of $c'.m'$.
— Construct the control-flow-graph (CFG) for the method body of $c'.m'$, and check that:

  – The only way to enter the block between L_{lb} and L_n is by entering through L_{lb}. (No jumps over blocks of inlined code or into the middle of inlined code)

  – Each path in the CFG that passes through L_{lb} also passes through L_{ub}, L_{call}, L_{la}, and L_n, or leads to `System.exit()`.

In addition, to make sure that the global security state (stored in static fields of the `SecState` class) is only accessed under the `SecState` lock, perform the following checks:

— No other `ldc SecState` instructions occur anywhere in the program. This makes sure the SecState class object is only used for acquiring or releasing a lock, and no other aliases to the object are created.
— `putstatic` and `getstatic` for fields of the SecState class only occur between L_{lb} and L_{ub}, and between L_{la} and L_n labels.

These checks allow us to reason about the actual inlined security state sequentially (because all accesses to that state happen under a single lock). Moreover, any invariant on the security state that is true in the initial state and maintained by each block of code that holds the SecState lock will be true at each program point where the SecState lock is not held.

These two observations will be crucial in designing the second step of the checker. For this second step, the checker will inline a reference automaton used for verification purposes, henceforth referred to as a "ghost reference monitor", or ghost IRM for short. We first describe this ghost IRM and how it is inlined by the checker.

9.2. *The Ghost Reference Monitor*

The ghost IRM is implemented by inserting special purpose assignments called *ghost instructions* into the program. The ghost instructions are essentially ConSpec rules, lightly compiled to evaluate guards and updates using the JVM stack and heap, together with a set of auxiliary *ghost variables* used to represent the state of the ghost IRM, and to store intermediate values, e.g. across method calls. Programs containing ghost instructions are called *augmented programs*.

A ghost instruction has the shape

$$\langle x^g := a_1 \to e_1 \mid \ldots \mid a_n \to e_n \rangle$$

where x^g is a vector of *ghost variables*, a_i are guard assertions and e_i are expression vectors of the same type and dimension as x^g. The instruction assigns the first expression whose guard holds, to the left hand side variable, similar to the way ConSpec rules are evaluated. If no guards hold, the instruction *fails* and the execution is said to be *incorrect*. The guards a_i and expressions e_i may refer to ghost variables, actual variables, the stack, and they may extract callee and thread id as described above.

Example 3. The ghost instruction below could be used to express that an execution is incorrect if the invoke instruction is executed with true as argument more than 10 times.

> . . .
> $\langle x^g := \mathrm{s}_0 \wedge x^g < 10 \to x^g + 1 \mid \neg\mathrm{s}_0 \to x^g \rangle$
> invoke $c.m$
> . . .

Ghost variables can be global or local. This scope will be notationally clarified by the superscripts x^g and x^{gl}, respectively.

An execution of an augmented program is a sequence of augmented configurations which in turn are regular configurations augmented with a ghost variable valuation. An augmented program is said to be *correct* if all of its executions are correct.

9.3. *Ghost Inlining*

The ghost inliner augments clients with ghost instructions to maintain various types of state information. This includes the ghost IRM state, intermediate data used only by the ghost IRM, and information to assist the checker in relating the ghost IRM state and the actual IRM state.

The code consumer will perform the ghost inlining algorithm, using the following inputs:

— The consumer policy, from which the ghost IRM state, and the implementation of the ghost IRM state transitions can be computed.

— The code and the certificate.

The ghost inliner introduces the variables listed in Table 3, and it implements the ghost IRM by inserting blocks of ghost instructions according to the following scheme. For each $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$ tuple in the certificate for a call to security relevant method $c.m$, do the following:

Identifier	Purpose
ms^g	A global vector representing the ghost security state, i.e. a type correct assignment to the security state variables as in Section 3.
$status^{gl}$	A local variable ranging over ready, meaning that the action trace is in sync with the ghost IRM, or before_$c.m$, return_$c.m$, indicating that the ghost IRM is one pre- or post-action out of sync.
arg^{gl}, tid^{gl}, o^{gl}, r^{gl}	Local variables to hold the arguments of security relevant calls during the call (they may be referenced in an after-clause), resp. calling thread, callee, and return value.

Table 3. *Variables introduced by ghost inliner.*

1   Insert in $c'.m'$ before label $L_{ub} - 1$:

$$\langle tid^{gl} := \texttt{Thread.currentThread}() \rangle$$
$$\langle o^{gl} := \mathrm{s}_0 \rangle$$
$$\langle arg^{gl} := (\mathrm{s}_1, \ldots, \mathrm{s}_n) \rangle$$
$$\langle ms^g := status^g = \mathsf{ready} \to \delta((tid^{gl}, c.m, o^{gl}, arg^{gl})^{\uparrow}) \rangle$$
$$\langle status^{gl} := \mathsf{before\_}c.m \rangle$$

If $c.m$ is security relevant but not BEFORE security relevant the ghost security state ms^g is not updated, but the other assignments are still performed.

2   Insert in $c'.m'$ before label L_{call}:

$$\langle status^{gl} := status^{gl} = \mathsf{before\_}c.m$$
$$\land\ o^{gl} = \mathrm{s}_0\ \land arg^{gl} = (\mathrm{s}_1, \ldots, \mathrm{s}_n) \to \mathsf{ready} \rangle$$

3   Insert in $c'.m'$ after label L_{call}:

$$\langle r^{gl} := \mathrm{s}_0 \rangle$$
$$\langle status^{gl} := status^{gl} = \mathsf{ready} \to \mathtt{return\_}c.m \rangle$$

4   Insert in $c'.m'$ before label $L_n - 1$:

$$\langle ms^g := status^{gl} = \mathtt{return\_}c.m \to \delta((tid^{gl}, c.m, o^{gl}, arg^{gl}, r^{gl})^{\downarrow}) \rangle$$
$$\langle status^{gl} := \mathsf{ready} \rangle$$

We refer to ghost instruction blocks inserted according to condition i above as a *block of type i*.

A schematic summary of the treatment of a security relevant invoke is illustrated in Figure 8. Correctness is proved by an extension of the inliner security argument of Section 7. In analogy with Proposition 3 we first show that the ghost inliner is sound in the sense that traces of the ghost monitor are allowed by the policy, and we then show security through a serialization property similar to Lemma 1.

Let $\mathcal{I}^g(\mathcal{P}, Prg)$ be the result of ghost inlining Prg with respect to policy $\mathcal{P}$ and Prg's
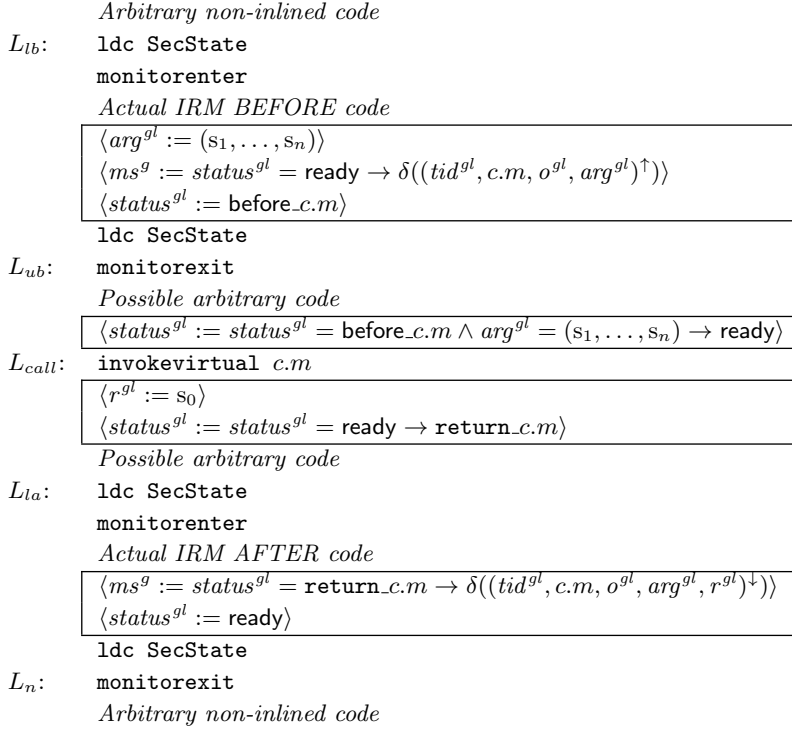
Arbitrary non-inlined code

L_{lb}:   `ldc SecState`

       `monitorenter`

       Actual IRM BEFORE code

$$\langle arg^{gl} := (\mathsf{s}_1, \ldots, \mathsf{s}_n) \rangle$$
$$\langle ms^g := status^{gl} = \mathsf{ready} \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl})^{\uparrow})\rangle$$
$$\langle status^{gl} := \mathsf{before\_c.m}\rangle$$

       `ldc SecState`

L_{ub}:   `monitorexit`

       Possible arbitrary code

$$\langle status^{gl} := status^{gl} = \mathsf{before\_c.m} \wedge arg^{gl} = (\mathsf{s}_1, \ldots, \mathsf{s}_n) \rightarrow \mathsf{ready}\rangle$$

L_{call}:   `invokevirtual` $c.m$

$$\langle r^{gl} := \mathsf{s}_0 \rangle$$
$$\langle status^{gl} := status^{gl} = \mathsf{ready} \rightarrow \mathbf{return\_}c.m\rangle$$

       Possible arbitrary code

L_{la}:   `ldc SecState`

       `monitorenter`

       Actual IRM AFTER code

$$\langle ms^g := status^{gl} = \mathbf{return\_}c.m \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl}, r^{gl})^{\downarrow})\rangle$$
$$\langle status^{gl} := \mathsf{ready}\rangle$$

       `ldc SecState`

L_n: `monitorexit`

       Arbitrary non-inlined code

Fig. 8. Schematic summary of ghost inlining for `invokevirtual` $c.m$. Current thread *tid* and callee s_0 has been omitted for brevity.

certificate. Similar to Section 7 we compare the observable actions of *Prg* with *ghost actions* α^g of $\mathcal{I}^g(\mathcal{P}, Prg)$. The *ghost extended trace* of an execution E, $\tau_{ge}(E)$ is the sequence of observable actions and ghost actions of E, and the ghost trace of E, $\tau_g(E)$, projects from $\tau_{ge}(E)$ the ghost actions only.

Proposition 4. Let E be a legal execution of $\mathcal{I}^g(\mathcal{P}, Prg)$. Then $\tau_g(E) \in \mathcal{P}$.

Proof. Let $\tau_g(E) = \alpha_0{}^g \cdots \alpha_n{}^g$ be the ghost trace of E. In the context of E, say that a block of type 1 *justifies* a block of type 2 or 4, if the values assigned to ghost variables o^{gl}, arg^{gl} in the type 1 block are the values used in the block of type 2 or 4. For the case of a type 2 block the value of $status^{gl}$ also needs to match the value assigned in the type 1 block. Similarly say that a block of type 4 *confirms* a block of type 3, if the values assigned to r^{gl}, $status^{gl}$ in the type 3 block are those used in the type 4 block.

If α_n is a pre-action then a block of type 1 justifying $\alpha_n{}^g$ happens before $\alpha_n{}^g$ and after $\alpha_{n-1}{}^g$. Since the prefix of $\tau_g(E)$ not including $\alpha_n{}^g$ is in $\mathcal{P}$, so is $\tau_g(E)$. For this argument to work out we need to observe that, if $\alpha_{n-1}{}^g$ is a block of type 3 then that block is confirmed by a block of type 4 before control is transferred to the block of type 1 justifying $\alpha_n{}^g$. The case of α_n a post-action is virtually identical and left to the reader. □

With Proposition 4 in place the security proof is essentially complete, as the proof of serialization can follow that of Lemma 1 line for line.

As a result we obtain the correlate of the Inliner Security Theorem, now transferred to the ghost inliner:

Theorem 6 (Ghost Security). If $\mathcal{P}$ is race-free, and *Prg* is a correct program, then $\mathcal{T}(\mathcal{I}^g(\mathcal{P}, Prg)) \subseteq \mathcal{P}$ $\qquad\qquad\square$

9.4. *The checker*

The checker algorithm should check that a given program (with certificate) satisfies a code consumer policy. To achieve this, the checker first performs Check 1 from Section 9.1. Then the checker augments a ghost IRM based on the consumer policy. Building on Theorem 6, the only remaining thing the checker needs to do is verify that the resulting program is *correct*, i.e. that none of the inlined ghost instructions fail.

Checking that an arbitrary program with inlined ghost instructions is correct is a hard problem, as hard as verifying full functional correctness of multithreaded Java code. However, with the assumptions we made about the actual inlining process, and given the concrete ghost inlining algorithm, checking correctness can be substantially simplified. In particular, we show in this section that verification of correctness can be done using sequential reasoning only. We assume that we are given as an oracle a proof checker for a standard sequential bytecode program logic (for instance the logic proposed by Bannwart and Müller (Bannwart and Müller, 2005)). In order to ensure that sequential verification is sound in our multithreaded setting, we rewrite the bytecode before sending it to the sequential verifier. In a multithreaded setting, reads from the heap are not necessarily stable. The only two parts of the state that we can reason about sequentially are local variables and the global security state (while the SecState lock is being held). We encode this by replacing all other reads from the heap by method calls to a method `randomValue()` of appropriate return type. This ensures that the verifier knows nothing about values read from the heap. Whenever we send blocks of bytecode (and corresponding proofs) to the verification oracle, we preprocess these blocks of bytecode to (1) remove all the locking/unlocking instructions, and (2) to replace reads from the heap (except reads of the fields of SecState in the *LB* or *LA* phase) with calls to such a `randomValue()` method of the appropriate type.

To support this second part of the checking algorithm, the code producer should include additional information in the certificate.

First, the code producer should provide an invariant $I(ms, ms^g)$ that relates the actual inlined security state ms to the ghost inlined security state ms^g. This invariant can be through of as a *simulation* relation between the states of the actual security automaton and the ghost automaton. Obviously, $I(ms, ms^g)$ is only allowed to refer to ghost security state variables and to static fields of the SecState class.

Second, the certificate provided by the code producer should contain some proofs checkable by the sequential program verification oracle, as detailed below.

Check 2. For each tuple $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$ in the certificate for a security relevant call to $c.m$, the checker performs the following verifications:

— For the locked before block B (the code between the acquiring of the SecState lock at L_{lb} and releasing of that lock at L_{ub}), check that the certificate contains a valid proof that the following code:

$$\langle ms^g := \delta((tid^{gl}, c.m, s_0, (s_1, \ldots, s_n))^\uparrow)\rangle; B$$

maintains the invariant $I(ms, ms^g)$, and does not fail when started from a state where this invariant is true.
— For the full inlined block F (the code between the acquiring of the SecState lock at L_{la} and releasing of that lock at L_n), check that the certificate contains a valid proof that F maintains the invariant $I(ms, ms^g)$, and does not fail when started from a state where this invariant is true.

Finally, check that $I(ms, ms^g)$ holds for the default initial values for all ghost and actual security state variables.

Lemma 2. If a program passes the checker, then, in any execution of the program, the invariant $I(ms, ms^g)$ holds whenever the SecState lock is not being held by any thread.

Proof. By contradiction. Assume there is an execution that violates this property. Identify the first step in the execution where the property fails. This cannot be the first step of the execution, as Check 2 checks that $I(ms, ms^g)$ holds in the initial state. Since changes to the variables mentioned in the invariant can only be done under the SecState lock (Check 1), the first step where the property fails must be a step where the SecState lock is being released. Because of Check 1, the lock can only be released by an instruction that is labeled L_{ub} or L_n. Let us consider the case L_n (the other case is similar), and let us call the thread that performs this monitorexit t. Select from the execution all steps from the thread t. Since t reaches L_n, and because of the control flow checks in Check 1, one of these execution steps must execute the instruction at L_{lb}. Consider the last step of thread t that executes the instruction at L_{lb}, and remove from the execution all steps before that one. The resulting execution is a single-threaded execution of the full inlined block F verified in Check 2 to maintain the invariant. Moreover, the execution starts in a state where the invariant holds (because we have selected the first step in the execution where the property fails). If our sequential verification oracle is sound, this can not happen. □

We can now show that the checker is secure: if all the checks succeed, the program being checked is secure.

Theorem 7. A program that passes the checker is secure.

Proof. By Theorem 6 it suffices to prove that the ghost inlined program can never fail. We prove this by contradiction. Assume there is an execution of the program that fails, i.e. that leads to one of the guards in the ghost statements evaluating to false. We show that from this execution, we can construct a failing single-threaded execution of one of the blocks of code that have been verified not to fail by the sequential verification oracle.

Let the thread identifier of the thread where the failure happens be t.

Consider all steps of thread t leading to the failure of a ghost statement. Because of the CFG check in Check 1, and since thread t reaches one of the ghost inlined instructions, thread t must have executed the instruction at label L_{lb}. Select the latest execution by thread t of that instruction, and remove all steps before that step. The remaining execution is a single threaded execution of the full inlined block verified not to fail during Check 2. Contradiction. $\qquad\square$

9.5. *Creating certificates for the example inliner*

Finally, we show that a code producer that uses the concrete inliner $\mathcal{I}_{Ex}$ that we proposed in Section 7 can easily produce a certificate that the resulting program complies with the inlined policy. Certificates contain three parts:

— For each security relevant `invokevirtual` bytecode instruction at a label L_{call} in method $c'.m'$, a certificate contains the tuple $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_n)$ marking the beginning and ending of the different phases of the inliner. Computing these for $\mathcal{I}_{Ex}$ is trivial.
— An invariant $I(ms, ms^g)$ that relates ghost security state to actual security state. To certify that an inlined program complies with the inlined policy, this invariant is just the identity.
— For each security relevant `invokevirtual` bytecode instruction, the certificate contains two sequential correctness proofs, one for the locked before block B, and one for the full inlined block F. It is an easy exercise to verify that the code blocks produced by our inliner are valid. Given an oracle for constructing proofs of valid programs in sequential Java, we can complete the certificate with this third part.

Theorem 8. A program inlined with our inliner and with a certificate constructed as above will pass the checker. $\qquad\square$

To summarize, we have shown that our inliner is able to inline a reference monitor in a way such that it is statically decidable whether or not the resulting program adheres to the given (race-free) policy. This is what Hamlen et al refers to as $\mathcal{P}$-verifiability (Sridhar and Hamlen, 2011). Thus, put another way, we have shown that the the set of race free policies are $\mathcal{P}$-verifiable.

9.6. *Discussion*

The checker developed in this section is, to the best of our knowledge, the first one that can certify compliance with security automata for multithreaded Java bytecode. The certification approaches proposed by other authors (and discussed in Section 1.1) focus on sequential programs only, or on blocking inliners for multithreaded programs. While our checker can only handle programs that have been generated by an inliner that complies with the assumptions we outlined in Section 9.1 (it will reject any other program as possibly insecure), this is a significant step forward. However, further improvements are possible.

Most importantly, one of the key motivations for Proof-Carrying Code is that it can reduce the Trusted Computing Base (TCB). Security only relies on correctness of the verifier, not on the (possibly complicated) techniques used by the code producer to construct the code and the proof. In many PCC approaches, the verifier is just a proof checker for proofs in a simple program logic. The checker we proposed in this paper is significantly more complicated than that. The main reason for this is that there is no existing program logic for multithreaded Java bytecode. Designing such program logics (and proving them sound) is an important avenue for future work.

What we did show in this section is how, for the class of inliners that we support, the issues related to multithreading can be handled separately using a relatively simple syntactic check (Check 1). Given a suitable program logic, it is likely that the insight reported in this section could be used to construct security proofs in that logic for programs that are inlined with such an inliner. Then, security could be verified using just a proof checker for a program logic.

Even though we have not yet reached that stage, our checker is still significantly simpler than the inliner: ghost inlining is done at a higher level of abstraction, and avoids many of the intricate bytecode rewriting tasks that the real inliner has to deal with, including things such as updating jumps, recomputing switch tables, updating exception handling tables, and so forth.

10. Conclusions and Future Work

Inlining is a powerful and practical technique to enforce security policies. Several inlining implementations exist, also for multithreaded programs. The study of correctness and security of inlining algorithms is important, and has received a substantial amount of attention the past few years. But, these efforts have focused on inlining in a sequential setting. This paper shows that inlining in a multithreaded setting brings a number of additional challenges. Not all policies can be enforced by inlining in a manner which is both secure and transparent. Fortunately, these non-enforceable policies do not appear very important in practice: They are policies that constrain not just the program, but also the API or the scheduler. We have identified a class of so-called race-free policies which characterizes exactly those policies that can be enforced by inlining in a secure and transparent fashion on multithreaded Java bytecode. This result is quite general: It relies mainly on the ability of policies to distinguish between entries to and exits from some set of API procedures, and very little on the specificities of the Java threading model. We have shown that the approach is useful in practice by applying it in several realistic application scenarios, and we have shown how certification of inlining in the multithreaded setting can be reduced to standard verification condition checking for sequential Java.

A number of extensions of this work merit attention. We discuss three issues: Inheritance, iterated inlining, and callbacks.

Inheritance, first, is relatively straightforward: In order to evaluate the correct event clause, runtime checks on the type of the callee object would be interleaved with the checks of the guards. This is spelled out for the sequential setting in (Vanoverberghe

and Piessens, 2009) for C#. We do not expect any issues to carry this over to the multithreaded setting.

For iterated inlining there are two options:

1. The ConSpec policies are merged before inlining. This can be done using a straight-forward, syntactic cross product construction for policies, $\mathcal{I}(\prod_i \mathcal{P}_i, Prg)$.
2. Alternatively, the monitors can be nested by inlining one policy at a time: $\mathcal{I}(\mathcal{P}_n, \ldots, \mathcal{I}(\mathcal{P}_2, \mathcal{I}(\mathcal{P}_1, Prg)) \ldots)$.

If the example inliner, $\mathcal{I}_{Ex}$, is used, the certification approach described above is general enough to easily certify the fully inlined program from certificates for each policy $\mathcal{P}_i$ by itself. If a different inliner is used however, the second approach needs a different treatment in general. One common strategy, for instance, is to create a wrapper method for each security relevant method, place the policy code in the wrapper method and replace the security relevant calls, with calls to the wrapper methods. The reason for this is that, except for the last inlining step, the inlined policy code will no longer reside in the same method as the security relevant call. To handle this one can either:

— Do the analysis from the first inlined `BEFORE`-instruction, to the last inlined `AFTER` / `EXCEPTIONAL` instruction globally. (This is obviously not tractable in general, but for simple wrapper methods it would not pose any problems.)
— Perform a simple renaming of security relevant methods, so that the inner policies consider the new wrapper methods to be security relevant instead.

Callbacks can be accommodated as well, but with more significant changes. First, the notion of event must be changed, to include not only calls from the client program to the API and return, but also from the API to the client program. This affects not only the program model but also the policy language. The negative results will remain valid, but the inlining algorithm must be amended to inline pre- and post checks in each public client method.

Finally, we believe that our study of the impact of multithreading on program rewriting in the context of monitor inlining is a first step towards a formal treatment of more general aspect implementation techniques in a multithreaded setting. Indeed, our policy language is a domain-specific aspect language, and our inliner is a simple aspect weaver.

Acknowledgements

References

Aktug, I., Dam, M., and Gurov, D. (2009). Provably correct runtime monitoring. *J. Log. Algebr. Program.*, 78(5):304–339.

Aktug, I. and Naliuka, K. (2008). Conspec – a formal language for policy specification. *Science of Computer Programming*, 74(1-2):2 – 12. Special Issue on Security and Trust.

Apache Software Foundation (2002). Servlet api documentation. `http://download.oracle.com/docs/cd/E17802_01/products/products/servlet/2.5/docs/servlet-2_5-mr2/index.html`.

Bannwart, F. Y. and Müller, P. (2005). 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.

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

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

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

Chen, F. (2005). 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.

Chudnov, A. and Naumann, D. A. (2010). Information flow monitor inlining. In *CSF*, pages 200–214.

Dam, M., Jacobs, B., Lundblad, A., and Piessens, F. (2009). Security monitor inlining for multi-threaded Java. In *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genova, Italy, July 6-10, 2009, Proceedings,*, pages 546–569. Springer-Verlag.

Dam, M., Jacobs, B., Lundblad, A., and Piessens, F. (2010). Provably correct inline monitoring for multithreaded Java-like programs. *Journal of Computer Security*, 18:37 – 59.

DeLine, R. and Fähndrich, M. (2001). 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.

Desmet, L., Joosen, W., Massacci, F., Philippaerts, P., Piessens, F., Siahaan, I., and Vanoverberghe, D. (2008). Security-by-Contract on the .NET platform. *Information Security Technical Report*, 13(1):25–32.

Deutsch, P. and Grant, C. A. (1971). A flexible measurement tool for software systems. *Information Processing 71, Proc. IFIP Congress*, 1:320–326.

Erlingsson, U. (2004). *The inlined reference monitor approach to security policy enforcement*. PhD thesis, Dep. of Computer Science, Cornell University.

Erlingsson, U. and Schneider, F. B. (2000a). IRM enforcement of Java stack inspection. In *IEEE Symposium on Security and Privacy*, page 0246. IEEE Computer Society.

Erlingsson, U. and Schneider, F. B. (2000b). SASI enforcement of security policies: a retrospective. In *Proc. Workshop on New Security Paradigms (NSPW '99)*, pages 87–95. ACM Press.

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

Franks, J., Hallam-Baker, P., Hostetler, J., Lawrence, S., Leach, P., Luotonen, A., and Stewart, L. (1999). HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 (Draft Standard).

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional.

Hamlen, K. W. and Jones, M. (2008). Aspect-oriented in-lined reference monitors. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 11–20, New York, NY, USA. ACM.

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

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

HATS (2010). Evaluation of the core framework, deliverable 5.2 of project fp7-231620 (hats). `http://www.cse.chalmers.se/research/hats/sites/default/files/Deliverable52.pdf`.

Jones, M. and Hamlen, K. W. (2010). Disambiguating aspect-oriented security policies. In *AOSD*, pages 193–204.

Kim, M., Kannan, S., Lee, I., Sokolsky, O., and Viswanathan, M. (2001). Java-MaC: A runtime assurance tool for Java programs. *Electronic Notes in Theoretical Computer Science*, 55(2):218 – 235. RV'2001, Runtime Verification (in connection with CAV '01).

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

Ligatti, J., Bauer, L., and Walker, D. (2005). Edit automata: Enforcement mechanisms for run-time security policies. *International Journal of Information Security*, 4(1–2):2–16.

Ligatti, J. A. (2006). *Policy Enforcement via Program Monitoring*. PhD thesis, Princeton University.

Lindholm, T. and Yellin, F. (1999). *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc.

Lipton, R. J. (1975). Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18:717–721.

Lundblad, A. (2010). Inlined reference monitors for multithreaded java: Case-studies. `http://www.csc.kth.se/~landreas/mt_inlining/`.

N. Dragoni, F. Massacci, K. N. and Siahaan, I. (2007). 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.

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

OWASP (2010). Owasp top 10 - 2010. `http://owasptop10.googlecode.com/files/OWASP\%20Top\%2010\%20-\%202010.pdf`.

Reynal, S. (2010). Jpicedt website. `http://jpicedt.sourceforge.net/`.

S^{3}MS (2008). Project web page. `http://www.s3ms.org`.

Schneider, F. B. (2000). Enforceable security policies. *ACM Trans. Information and Systems Security (TISSEC)*, 3(1):30–50.

Skalka, C. and Smith, S. (2004). History effects and verification. In *Asian Programming Languages Symposium*.

Sridhar, M. and Hamlen, K. W. (2010a). Actionscript in-lined reference monitoring in prolog. In *PADL*, pages 149–151.

Sridhar, M. and Hamlen, K. W. (2010b). Actionscript in-lined reference monitoring in prolog. In *PADL*, pages 149–151.

Sridhar, M. and Hamlen, K. W. (2010c). Model checking in-lined reference monitors. In *Verification, Model Checking, and Abstract Interpretation*, pages 312–327.

Sridhar, M. and Hamlen, K. W. (2011). Flexible in-lined reference monitor certification: challenges and future directions. In *Proceedings of the 5th ACM workshop on Programming languages meets program verification*, PLPV '11, pages 55–60, New York, NY, USA. ACM.

Vanoverberghe, D. and Piessens, F. (2009). Security enforcement aware software development. *Inf. Softw. Technol.*, 51(7):1172–1185.

Viswanathan, M. (2000). *Foundations for the run-time analysis of software systems*. PhD thesis, University of Pennsylvania.

Walker, D. (2000). 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.

Appendix A. Program Model and JVM Instruction Semantics

A few inessential simplifications have been made to ease presentation. In particular we ignore all issues concerning inheritance and dynamic binding as this has been addressed elsewhere (Vanoverberghe and Piessens, 2009; Aktug et al., 2009).

A.1. *Basic Conventions*

We use c for class names, m for method names, and f for field names. Class names are fully qualified. A type *Type* is either a class name or a primitive type. A method definition is a pair of an instruction array and an exception handler array. Exception handlers (b, e, t, c) catch exceptions of type c (and its subtypes) raised by instructions in the range $[b, e)$ and transfers control to address t, if the handler is the topmost handler in the exception handler array that handles the instruction for the given type. Values (Java primitives and object references) are ranged over by v. An object reference is a (typed) location o, or the value *null*. Locations are mapped to objects, or arrays, by a heap h. Objects are finite maps of non-static fields to values. Static fields are identified with field references of the form $c.f$. To handle those, heaps are extended to assignments of values to static fields.

A.2. *Configurations, Transitions, and Programs*

A *configuration* $C = (h, \Lambda, \Theta)$ consists of a *heap* h, a *lock map* Λ which maps an object reference o to a thread id *tid* iff *tid* holds the lock of o, and a *thread configuration map* Θ which maps a thread identifier *tid* to a thread configuration, often denoted by θ. A thread configuration is a stack R of activation records. For normal thread configurations, the activation record at the top of an execution stack has the shape (M, pc, s, l), where M is the currently executing method, pc is the program counter, s is the operand stack (of values), and l is the local variables. For exceptional configurations, the top frame of an execution stack has the form (o) where o is the location of an exceptional object, i.e. of class Throwable. A transition semantics determining the transition relation $C \to C'$ is given for key instructions and configuration types (see Section A.4). A *program Prg* consists of a set of class declarations determining types of fields and methods belonging to classes in *Prg*, and a method environment assigning method definitions to each method in *Prg*.

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. (Leroy, 2003)).

A.3. *Field Accesses and Legal Executions*

In this paper, we wish to reason about the behavior of arbitrary multithreaded programs. Therefore, we cannot assume that the programs we consider are correctly synchronized.

This complicates our execution semantics, because non-correctly-synchronized programs may exhibit non-sequentially-consistent executions (Chapter 17 of the Java Language Specification (JLS3) (Gosling et al., 2005)). An execution is sequentially consistent if there is a total order on the field accesses in the execution such that each read of a field yields the value written by the most recent preceding write of that field in this total order. In order to ensure that our semantics captures all possible executions of a program, the transition relation $\rightarrow$ does not constrain the value yielded by a field read; specifically, it does not imply that this value is the value in the heap for that field. However, JLS3 does provide some guarantees, even for non-correctly-synchronized programs. Therefore, below we will consider only *legal executions*. A legal execution is an execution which satisfies both the transition relation $\rightarrow$ and the memory consistency constraints of JLS3.

The *happens-before order* (Gosling et al., 2005) is a partial order on the transitions in an execution. It consists of the program order (ordering of two actions performed by the same thread) and the synchronizes-with order (order induced by synchronization constructs), and the transitive closure of the union of these.

An important guarantee provided by JLS3 that we rely on in this paper, is that if in some legal execution a given field is protected by a given lock, then each read of that field yields the value written by the most recent preceding write of that field. We say that a given field is protected by a given lock in a given execution, if whenever a thread accesses the field, it holds the lock.

A.4. *Transition Semantics*

We present a transition semantics of JVM instructions used in proofs. The semantics applies to type-safe configurations and bytecode verified programs only, cf. (Leroy, 2003). We only present the rules for the bytecode instructions mentioned in the paper. The rules for the other bytecode instructions are similar and straightforward.

Notation

Besides self-evident notation for function updates, array lookups etc. the transition rules uses the following auxiliary operations and predicates:

— $v :: s$ pushes v on top of stack s
— $handler(M, h, o, pc)$ returns the proper target label given M's exception handler H, heap h, throwable o and pc pc in the standard way:

$$handler(M, h, o, pc) = handler'(H, h, o, pc)$$
$$handler'(\epsilon, h, o, pc) = \bot$$
$$handler'((b, e, t, c) \cdot H', h, o, pc) = \begin{cases} t & \text{if } b \leq pc < e \text{ and } h \vdash o : c \\ handler'(H', h, o, pc) & \text{otherwise} \end{cases}$$

— v is an argument vector
— Stack frames have one of three shapes (M, pc, s, l), (o) where o is throwable in the current heap, and $(\Box)$ used for API calls (see Section 2).

Local Variables and Stack Transitions

$$\frac{\Theta(tid) \to \theta}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto \theta])}$$

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

$$\frac{M[pc] = \texttt{astore } n}{(M, pc, v :: s, l) :: R \to (M, pc + 1, s, l[n \mapsto v]) :: R}$$

$$\frac{M[pc] = \texttt{athrow}}{(M, pc, o :: s, l) :: R \to (o) :: (M, pc + 1, o :: s, l) :: R}$$

$$\frac{M[pc] = \texttt{goto } L}{(M, pc, s, l) :: R \to (M, L, s, l) :: R}$$

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

$$\frac{M[pc] = \texttt{ldc } c}{(M, pc, s, l) :: R \to (M, pc + 1, c :: s, l) :: R}$$

$$\frac{M[pc] = \texttt{ifeq } L \qquad n = 0}{(M, pc, n :: s, l) :: R \to (M, L, s, l) :: R}$$

$$\frac{M[pc] = \texttt{ifeq } L \qquad n \neq 0}{(M, pc, n :: s, l) :: R \to (M, pc + 1, s, l) :: R}$$

Heap transitions
As discussed in Section 2, field reads return an arbitrary value, and these rules should
be complemented with the Java memory model constraints.

$$\frac{\Theta(tid) = (M, pc, v :: s, l) :: R \qquad M[pc] = \texttt{putstatic } c.f}{(h, \Lambda, \Theta) \to (h[c.f \mapsto v], \Lambda, \Theta[tid \mapsto (M, pc + 1, s, l) :: R])}$$

$$\frac{\Theta(tid) = (M, pc, s, l) :: R \qquad M[pc] = \texttt{getstatic } c.f}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto (M, pc + 1, v :: s, l) :: R])}$$

Locking instructions

$$\frac{\begin{array}{c}\Theta(tid) = (M, pc, v :: s, l) :: R \\ M[pc] = \texttt{monitorenter} \qquad \Lambda(v) = \bot\end{array}}{(h, \Lambda, \Theta) \to (h, \Lambda[v \mapsto tid], \Theta[tid \mapsto (M, pc + 1, s, l) :: R])}$$

$$\frac{\begin{array}{c}\Theta(tid) = (M, pc, v :: s, l) :: R \\ M[pc] = \texttt{monitorexit} \qquad \Lambda(v) = tid\end{array}}{(h, \Lambda, \Theta) \to (h, \Lambda[v \mapsto \bot], \Theta[tid \mapsto (M, pc + 1, s, l) :: R])}$$

Exceptional Transitions

$$\frac{\begin{array}{c}\Theta(tid) = (o) :: (M, pc, s, l) :: R \\ pc' = handler(M, h, o, pc) \qquad pc' \neq \bot\end{array}}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto (M, pc', s, l) :: R])}$$

$$\frac{\begin{array}{c}\Theta(tid) = (o) :: (M, pc, s, l) :: R \\ handler(M, h, o, pc) = \bot\end{array}}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto (o) :: R])}$$

API calls

API calls are treated specially, as discussed in Section 2. The rules below only deal with invocation of API methods. Other invocations (client code calling client code) are standard, and we don't spell out the rule here.

$$\frac{\begin{array}{c}\Theta(tid) = (M, pc, o :: v :: s, l) :: R \\ M[pc] = \texttt{invokevirtual } c.m \qquad c \in API\end{array}}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto (\Box) :: (M, pc + 1, s, l) :: R])}$$

Exceptional return from an API method:

$$\frac{\Theta(tid) = (\Box) :: R}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto (o) :: R])}$$

Normal return from an API method:

$$\frac{\Theta(tid) = (\Box) :: (M, pc, s, l) :: R}{(h, \Lambda, \Theta) \to (h, \Lambda, \Theta[tid \mapsto (M, pc, v :: s, l) :: R])}$$