# Security Monitor Inlining for Multithreaded Java

Mads Dam[1], Bart Jacobs[2⋆], Andreas Lundblad[1], and Frank Piessens[2]

[1]KTH, Sweden
{mfd,landreas}@kth.se

[2]K.U.Leuven, Belgium
{bartj,frank}@cs.kuleuven.be

**Abstract.** Program monitoring is a well-established and efficient approach to security policy enforcement. An implementation of program monitoring that is particularly appealing for application-level policy enforcement is monitor inlining: the application is rewritten to push monitoring and policy enforcement code into the application itself. The intention is that the inserted code enforces compliance with the policy (security), and otherwise interferes with the application as little as possible (conservativity and transparency).
For sequential Java-like languages, provably correct inlining algorithms have been proposed, but for the multithreaded setting, this is still an open problem. We show that no inliner for multithreaded Java can be both secure and transparent. It is however possible to identify a broad class of policies for which all three correctness criteria can be obtained. We propose an inliner that is correct for such policies, implement it for Java, and show that it is practical by reporting on some benchmarks.

## 1 Introduction

Program monitoring is a well-established and efficient approach to prevent potentially misbehaving software clients from causing harm, for instance by violating system integrity properties, or by accessing data to which the client is not entitled [1, 2]. The conceptual model is simple: Potentially dangerous actions by a client program are intercepted and routed to a policy decision point in order to determine whether the actions should be allowed to proceed or not. In turn, these decisions are routed to a policy enforcement point, responsible for ensuring that only policy-compliant actions are executed. For the purpose of this paper, we will assume that policies are given as security automata in the style of Schneider [3].

Program monitoring can be implemented in different ways. The monitor can be external to the program being monitored: it could for instance be implemented as a proxy API, as part of a virtual machine, or as part of an operating system kernel.

An alternative implementation approach which is particularly appealing for application-level policy enforcement is monitor inlining [2]. Here, code rewriting is used to push policy relevant functionality into the client programs themselves.

⋆ Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

For sequential programs, external monitoring and inlined monitoring enforce the same class of policies [4].[1] We show that, somewhat surprisingly, this is not true for multithreaded programs. The fact that the 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.

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 across the entire method call. This, however, can result in loss of transparency, performance degradation and, possibly, deadlocks. It is, however, possible to identify a large class of policies for which inlining remains a practical and efficient enforcement technique. We propose one such class, the *race-free policies*, and show that policies in this class can be enforced correctly by inlining in multithreaded Java. Moreover, we argue that the class of race-free policies is in fact the largest class of policies that is meaningful in a multi-threaded setting; the non-race-free policies by definition rely on execution constraints that go beyond those enforceable by inlining.

In particular, for many existing inlined monitoring systems whose formal treatment did not include multithreading but whose implementations could deal with multithreading [5–7], a non-race-free policy does most likely *not* express what the policy writer intended.

In summary, the paper makes the following contributions:

- We show that inlined monitoring in multithreaded Java is strictly less powerful than external monitoring.
- We characterize a class of policies that can be correctly enforced by inlining.
- We describe the design of an inlining algorithm and prove it correct for the identified class of policies.
- We report on our experience with a prototype implementation.

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.

### 1.1 Related Work

Schneider [3] proposed the use of automata as a tool to formalize security policies, and monitor inlining to enforce such policies was examined in [2, 8]. The PoET/PSLang toolset by Erlingsson [8] implements monitor inlining for Java. That work represents security automata directly in terms of Java code snippets, making it difficult to formally prove correctness properties of the approach. Subsequent work on monitor inlining that addresses correctness properties includes

[1] If we consider broader classes of policies than those expressible by security automata, program rewriting can enforce strictly more policies.

[9] and [10], but these papers only consider sequential programs. Several papers [8, 11, 7, 12] report on inliner implementations for multithreaded Java-like programs with locking regimes that appear essentially identical to the one used in our example algorithm. None of these works, however, analyze the implications of multithreading and locking on the enforceable class of policies. In previous work [13] we have examined the implication of locking across security relevant method calls, and to which extent transparency can be preserved in such a setting.

Edit automata [14, 15] are examples of security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event suppressions and insertions. As a consequence, edit automata can enforce a richer class of policies, the infinite renewal policies. A practical implementation of edit automata based on inlining is the Polymer system [6]. The main point of Polymer is to support composition of policies, and studying the impact of concurrency is left for future work.

There are many policy enforcement techniques, and the question of what classes of policies each policy enforcement technique can handle has received a considerable amount of attention. Schneider [3] kicked off this line of research, and his results were refined and extended by Viswanathan [16], Hamlen et al. [4] and others. Hamlen et al. distinguish three classes of enforcement mechanisms: static analysis, execution monitoring and program rewriting. They prove that when an execution monitor is afforded the same collection of intervention capabilities as an inliner, the inlining approach is strictly more powerful. This paper identifies an important domain where an external execution monitor has *more* intervention capabilities: in particular, an external execution monitor can freeze all threads in a program, whereas an inliner can only influence other threads by means of the synchronization primitives offered by the programming language.

Finally, inlining is closely related to aspect weaving. Aspects have been proposed by many authors as an implementation technique for security policy enforcement [14, 17–19]. Other authors have generalized the events that an inlined monitor can see from method invocations and returns to program events specified by more general pointcut expressions [12].

### 1.2 Overview of the Paper

The rest of this paper is structured as follows. In Section 2, we briefly discuss the formal model of the Java Virtual Machine that we use in the rest of the paper. Next, in Section 3, we discuss what security policies we consider in this paper, and we introduce notation for them. Then we define the notion of inliner, and the correctness properties for inliners. Section 5 shows that these correctness criteria cannot be met for the policies and programs that we consider. The following section introduces the class of race-free policies, and Section 7 proposes an inlining algorithm and shows it is correct for all race-free policies. Then we report on experience with our implementation, and we offer a conclusion.

## 2 Program Model

We want to prove properties of inliners that operate on Java bytecode. The inlined code will monitor the interaction of the bytecode with a given API. We abstract from the API implementation: it will in many cases be a native implementation as policies typically talk about methods that perform IO.

Hence, our formal model is a standard model of the JVM extended with facilities to call an external API. Most of the results in this paper do not depend on the details of this formal model: the limitations we identify for monitor inlining in a multithreaded setting hold for a wide class of imperative programming languages and execution environments. In this section, we discuss those aspects of the formal model that are relevant for the paper. An appendix gives a more detailed exposition, as well as proofs for the correctness of the example inliner that necessarily depend on these details.

The formal model is a standard small-step operational semantics that defines a transition relation $\rightarrow_{\mathrm{JVM}}$ on JVM configurations. An *execution* $E$ of a program $P$ is a (possibly infinite) sequence of JVM configurations $C_0C_1\ldots$ where $C_0$ is the initial configuration. The external API is modeled 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. 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. 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 evaluation stack; alternatively, when it throws an exception, an arbitrary, but correctly typed exceptional activation record is returned.

For the purpose of this paper, we assume sequential consistency of the JVM memory. This means we can reason about multithreaded executions as interleavings of single-thread executions, compatible with the happens-before order. The *happens-before order* [20] 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.

The real Java memory model is weaker and this impacts our work in interesting ways, but studying this impact is left for future work.

The JVM execution steps that are of interest in this paper are the steps where an API method is entered or exited. Given an execution $E$ the *observable trace* $\omega(E)$ of $E$ is defined as follows:

$$\begin{aligned}
\omega(C) &= \varepsilon \\
\omega(CC'E) &= \alpha\,\omega(C'E) \quad \text{if } C \xrightarrow{\alpha}_{\mathrm{JVM}} C' \\
\omega(CC'E) &= \omega(C'E) \quad \text{if } C \xrightarrow{\tau}_{\mathrm{JVM}} C'
\end{aligned}$$

where a transition from $C$ to $C'$ performs an observable action $\alpha$, denoted $C \xrightarrow{\alpha}_{\mathrm{JVM}} C'$, if and only if it transitions from the client code to the API or

vice versa. Specifically, we represent a call from client code bound at run time to an API method $c.m$ on an object $o$ with arguments $\mathbf{v}$ by a thread $tid$ as $C \xrightarrow{(tid,c.m,o,\mathbf{v})^{\uparrow}}_{\text{JVM}} C'$, and a normal return from this call with return value $r$ as $C'' \xrightarrow{(tid,c.m,o,\mathbf{v},r)^{\downarrow}}_{\text{JVM}} C'''$. We represent an exceptional return from this call with exception object $t$ as $C'' \xrightarrow{(tid,c.m,o,\mathbf{v},t)^{\Downarrow}}_{\text{JVM}} C'''$. All transitions other than the above are non-observable, denoted $C \xrightarrow{\tau}_{\text{JVM}} C'$.

We refer to actions $(tid, c.m, o, \mathbf{v})^{\uparrow}$, $(tid, c.m, o, \mathbf{v}, r)^{\downarrow}$, and $(tid, c.m, o, \mathbf{v}, t)^{\Downarrow}$ as before actions, after actions, and exceptional actions, respectively, and we collect them in sets $\Omega^{\uparrow}$, $\Omega^{\downarrow}$, and $\Omega^{\Downarrow}$. We refer to after and exceptional actions together as *end actions*, and we use *start action* as a synonym for before action.

The set of executions of a program $P$ is $exec(P)$. We define the set $\mathcal{T}(P)$ of traces of $P$ as $\mathcal{T}(P) = \{\omega(E) \mid E \in exec(P)\}$ .

We will assume for simplicity that program and API do not share fields. This is not a restriction, as shared data can be modeled using fields defined in the API implementation and accessed with getters and setters. This effectively makes these field accesses observable.

In our program model all interactions between client code and API happen through method invocations, and in such a setting sets of traces as defined above are an adequate model for program behavior [21, 22]: two programs with the same set of traces are observationally equivalent.

## 3 Security Policies

In this paper we consider only security policies that can be represented as security automata [3]. A *security automaton* is an automaton $\mathcal{A} = (Q, \delta, q_0)$ where $Q$ is a countable (not necessarily finite) set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Omega \rightharpoonup Q$ is a (partial) transition function, where $\Omega = \Omega^{\uparrow} \cup \Omega^{\downarrow} \cup \Omega^{\Downarrow}$. All states $q \in Q$ are viewed as accepting. Note that our notion of policy assumes that policies only talk about API method invocations and returns. Many existing enforcement systems make the same assumption ([8, 6]). This design decision limits our abilities to, for instance, perform any detailed data flow tracking. Policies in such a framework are typically sparse: Only a small number of API calls are actually security relevant, and calls to these methods are infrequent. But, the framework is sufficiently rich to allow a wide range of interesting policies to be expressed, and, in particular, it serves well as a generic setting in which to examine the effects of multithreading.

Our work uses the ConSpec language [23] for policy specification. ConSpec is similar to PSlang [8], but it has a formal semantics mapping ConSpec specifications to security automata.

An example of a ConSpec specification is given in Figure 1. The syntax is intended to be largely self-explanatory: The specification in Figure 1 states that the program has to ask the user for permission each time it intends to send a file over bluetooth. It does so by storing after a confirmation dialog what file the user has authorized to be sent, and to what URL it can be sent. Before an

invocation of the sendFile method, it is checked that the actual parameters of the invocation correspond to the stored filename and URL. Hence, if the program would not pop up a confirmation dialog before sending, or if it would send a different file or send to a different URL than those confirmed in the dialog, the policy will block the send.

```
SECURITY STATE    String requestorURL,
                  String requestedFile;

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

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

The macro `goodFileQuery(query)` returns true iff `query` is a well formulated file send query and `queryRequestor(query)` and `queryFile(query)` returns the requestor and file substrings of `query` respectively.

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

The example has two security relevant methods, `JOptionPane.showConfirmDialog` and `BluetoothToolkit.sendFile`. We refer to invocations and returns of such security relevant methods as *security relevant actions*. The specification expresses the constraints on security relevant actions in terms of guarded commands where the guards are boolean expressions and the updates are lists of assignments to security state variables. Both the guards and the assignments may mention the security state and the method call parameters. For an after action they may also mention the return value. In case the specification needs to talk about the current thread identifier, a ConSpec policy can call the `Thread.currentThread()` method. The only operation defined on thread identifiers is equality testing, so a policy can specify for instance that two invocations should happen in the same thread.

The *security state* declaration is a list of variable declarations. These variables represent the state space of the security automaton. For simplicity, we require that the initial values for the security state variables specified by the policy are the default initial values for their corresponding Java types. For example, the `requestedFile` variable in Figure 1 will initially be `null`.

An *event clause* defines how the security automaton reacts to a security relevant action. The event modifiers `BEFORE`, `AFTER` and `EXCEPTIONAL` specify if the event clause applies to a before action, after action or exceptional action. The method signature following the event modifier specifies the method that the event clause applies to. A sequence of guarded updates specifies the behaviour of

the security automaton in response to actions matching the event clause. Guards are evaluated top to bottom, in order to obtain a deterministic semantics. For the first guard that evaluates to true, the corresponding update block is executed. If no clause guards hold, the call is violating, i.e. the security automaton does not accept the action. We restrict our attention to security automata that always accept return and exceptional actions. That is, we require that if the event modifier is `AFTER` or `EXCEPTIONAL`, the guards are exhaustive.

A security automaton can be derived from a ConSpec policy in the obvious manner. We refer to [9] for details.

**Definition 1 (Policy Adherence).** *The program $P$ adheres to security policy $\mathcal{S}$, if for all executions $E$ of $P$, $\omega(E)$ is accepted by $\mathcal{S}$.*

We identify a policy $\mathcal{S}$ with the language of traces of observable actions that it accepts, and hence we write policy adherence as $\mathcal{T}(P) \subseteq \mathcal{S}$.

## 4 Inlining Correctness Properties

A security policy specified as a security automaton can be enforced by an *execution monitor* [3]. An execution monitor is an enforcement mechanism that can monitor the observable steps that a target program takes, and that can terminate the program if a step does not comply with the policy. Such a monitor could for instance be implemented in the Java Virtual Machine.

An alternative implementation mechanism for execution monitoring is *inlined reference monitors* [5]. *Inlining* refers to the procedure of compiling a policy into a bytecode based reference monitor and embedding it into a target program. Formally, an inliner is a function $\mathcal{I}$ which for each policy $\mathcal{S}$ and program $P$ produces an inlined program $\mathcal{I}(\mathcal{S}, P)$. The intention is that the inserted code enforces compliance with the policy, and otherwise interferes with the execution of the target program as little as possible.

In this section we look at traditional correctness properties for inlined monitors. There are three correctness properties of fundamental interest (cf. [15],[4]): namely, the inliner should enforce policy adherence (security), it should not add new behavior (conservativity), and it should not remove policy-adherent behavior (transparency). More formally:

**Definition 2 (Inliner Correctness Properties).** *An inliner $\mathcal{I}$ is:*

- Secure *if, for every program $P$, every trace of the inlined program $\mathcal{I}(\mathcal{S}, P)$ adheres to $\mathcal{S}$, i.e. $\mathcal{T}(\mathcal{I}(\mathcal{S}, P)) \subseteq \mathcal{S}$.*
- Conservative *if, for every program $P$, every trace of the inlined program $\mathcal{I}(\mathcal{S}, P)$ is a trace of $P$, i.e. $\mathcal{T}(\mathcal{I}(\mathcal{S}, P)) \subseteq \mathcal{T}(P)$.*
- Transparent*, if every adherent trace of the client program is also a trace of the inlined program, i.e. if $\mathcal{T}(P) \cap \mathcal{S} \subseteq \mathcal{T}(\mathcal{I}(\mathcal{S}, P))$.*

Inliners are only allowed to rewrite the program, and not the API. This is a realistic restriction. Even if an inliner rewrites all Java code, including the Java

API implementation, native calls for instance for IO will remain. In our model, the Java API would then be considered part of the program, and the monitored API would only consist of the natively implemented methods. In principle it would be possible to rewrite the native implementations as well, but the same issues would reoccur at the level of system calls, or, ultimately, of physical IO.

An upshot 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 we impose the restriction on policies that after actions and exceptional actions should always be allowed (have exhaustive guards in ConSpec, i.e. at least one guard should evaluate to true). These actions can still specify updates to the security state. In particular, they might cause the automaton to enter a state from which no further actions are possible.

## 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 Java Virtual Machine, it is impossible to do this with an inlined monitor.

A key factor that explains why there are policies that cannot be enforced by inlining is the fact that the inlined code can only control the scheduler indirectly through locking (whereas an external monitor can "freeze" the execution of the program while taking security decisions). Here is an example that illustrates this. Consider the policy in Figure 2. This policy says that $C.n()$ can only be

```
SECURITY STATE
    boolean ok = false;

BEFORE C.m()
    PERFORM
        true -> { ok = true; }
BEFORE C.n()
    PERFORM
        ok -> {}
```

**Fig. 2.** Not enforceable by inlining.

```
SECURITY STATE
    boolean ok = false;

AFTER C.m()
    PERFORM
        true -> { ok = true; }
BEFORE C.n()
    PERFORM
        ok -> {}
```

**Fig. 3.** Enforceable by inlining.

called after a call to $C.m()$ has been initiated (but not necessarily returned). So the trace $T_1 = (tid, C.m, o, \mathbf{v})^{\uparrow}, (tid', C.n, o', \mathbf{v}')^{\uparrow}$ is allowed, but the trace $T_2 = (tid', C.n, o', \mathbf{v}')^{\uparrow}, (tid, C.m, o, \mathbf{v})^{\uparrow}$ is not allowed by the policy.

But it is impossible to write any program $P$ that has the trace $T_1$ but that does not have the trace $T_2$ (unless API method $C.m$ collaborates, for instance by releasing a lock that is visible to the client on entry to $C.m$. But clearly this is not something one can assume about every API method).

Consider an example program $P_{ex}$ that has trace $T_1$, for instance the program that starts two independent threads where one calls $C.m()$ and the other calls $C.n()$. Assume also that no lock is shared between the API and the client program. There is no way an inliner can rewrite this program to securely and transparently enforce this policy, because the inliner has no way of synchronizing with the end of the before action of the $C.m()$ call. The inliner *can* synchronize with the return from $C.m()$, for instance by acquiring a lock across the call to $C.m()$ and forcing the thread that calls $C.n()$ to wait for that lock. But in that case, the inliner is actually enforcing the stronger policy shown in Figure 3.

The key observation is that such synchronization is impossible for the policy in Figure 2 (unless with help from the API, but the inliner cannot rewrite the API), and hence the ordering of the two before actions is up to the scheduler. Whatever the inliner does to the program, the inlined program will either have both traces (and thus the inliner was not secure) or it will have neither of the two traces (and thus the inliner was not transparent).

**Lemma 1.** *Any program that has an observable trace with two consecutive before actions, also has the same observable trace with these two before actions swapped.*

*Proof.* Two consecutive before actions are necessarily in different threads: within one thread, a before action is either the final action of that thread, or it is followed by an after or exceptional action.

For two consecutive before actions in different threads, there can be no happens-before relation between the two actions. This follows from the fact that the only way to introduce such a happens-before relation would be the synchronization on a lock: one thread would have to acquire the lock before doing the before action, and the other thread would have to release the lock after doing the before action. However, this would imply that this lock is shared between client program and API (as a thread is in client code immediately before a before action, and in the API immediately after a before action). Since we have ruled out such sharing, the result follows. □

The assumption that there is no shared lock between client and API is a reasonable assumption for many API's, and in particular for the native API.

**Theorem 1.** *No inliner can be secure and transparent for the policy in Figure 2.*

*Proof.* Consider the output $P'_{ex}$ of the inliner for the given policy and for the example program $P_{ex}$ above. The program $P_{ex}$ has the traces $T_1$ and $T_2$ discussed above. By lemma 1, $P'_{ex}$ either has both $T_1$ and $T_2$ (and hence the inliner was not secure on $P_{ex}$), or it has neither of these traces (and hence the inliner was not transparent for $P_{ex}$.) □

## 6 Race-free Policies

### 6.1 Definitions and Properties

Generalizing from the example in Figure 2, 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 does. These permutations that are always possible are captured by the notion of *client-order-preserving* permutations. (Recall that start actions are before actions, and that end actions are after or exceptional actions.)

**Definition 3.** *A permutation $\pi(T)$ of a trace $T$ of observable actions is* client-order-preserving *if, for any $i$ and $j$ such that $i < j$ and $T_i$ is an end action and $T_j$ is a start action, $\pi(i) < \pi(j)$.*

The intuition behind the definition is the following: the client can control start actions, and can only observe end actions. If a start action comes later than an end action, the client *could* have synchronized to ensure this ordering. The client cannot perform such synchronization for concurrent before actions or concurrent after actions. The definition also implies that actions within a single thread can never be permuted: within a thread, start and end actions are strictly interleaved.

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 client code. This is captured by the following definition:

**Definition 4.** *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, before actions require a permission to be present in this set and cause the permission to be removed, and after 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 bound is tight if we want the inliner to work for all possible API implementations. This follows from the following theorem.

**Theorem 2.** *No inliner can be secure and transparent for a non-race-free policy for all possible API implementations.*

*Proof.* Let $T$ be a trace accepted by the policy, and $T'$ a client-order preserving transformation of $T$ that is not accepted. Consider an API implementation that performs no synchronization. By an argument similar to the one in Lemma 1, any program that has the trace $T$ necessarily also has the trace $T'$: a client-order preserving permutation is always compatible with the happens-before ordering if the API does not perform any synchronization. Then, consider any program $P$ that has trace $T$. In order to be transparent, the inliner has to produce an inlined $P'$ that has $T$. But then $P'$ also has $T'$ and hence the inliner is not secure. □

An interesting question is how to check if a policy is race-free.

**Proposition 1.** *It is a necessary and sufficient condition for race-freedom that all start actions are right-movers and all end actions are left-movers in the set of allowed observable traces. (I.e., if a trace $T$ is allowed, then swapping a pair of consecutive actions $x, y$ in different threads where $x$ is a start action or $y$ is an end action yields an allowed trace.)*

*Proof.* Such swappings generate the client-order preserving permutations. □

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 $s_1$ of the security automaton corresponding to the policy, and for any pair of transitions with different thread identifiers starting in that state, $s_1 \xrightarrow{x} \xrightarrow{y} s_2$ where $x$ is a start action or $y$ is an end action, it holds that $s_1 \xrightarrow{y} \xrightarrow{x} s_2$.*

*Proof.* These conditions imply the conditions from Proposition 1. □

Sufficient syntactical conditions 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 condition is that start actions only consume permissions from the set, and after actions only add permissions.

### 6.2 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 even on the scheduler. Hence, we argue that such policies never 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 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 policy in Figure 2 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 policy 3 instead of policy 2 – 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).

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 Example Inliner

In this section we propose an inlining scheme that is secure, conservative and transparent for race-free policies.

The state of the inlined reference monitor 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.

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. We will see below that this is much trickier for an inliner that releases the lock during API calls. However, an inliner that locks across calls can introduce deadlocks in the inlined program and 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. The design and security proof of an inliner locking across calls is given in [13].

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 [10, 5, 6, 9] and the locking strategy is relatively straightforward. The novelty is the correctness proof. The same proof will be applicable to other inliners showing that, when one restricts oneself to race-free policies, these inliners are also correct.

### 7.1 The 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, in this example inliner, we impose the restriction on policies that they should have simple call matching. We say a policy has simple call matching if for

| Inlined label | Instruction |
|---|---|
| $L$: | `ldc` *SecState* |
| | `monitorenter` |
| | `astore` 0 |
| | ⋮ |
| | `astore` $n-1$ |
| *beforeG*$_1$: | `[eval before` $G_1$`]` |
| | `ifeq` *beforeG*$_2$ |
| | `[before update 1]` |
| | `goto beforeEnd` |
| | ⋮ |
| *beforeG*$_i$: | `[eval before` $G_i$`]` |
| | `ifeq` *exit* |
| | `[before update` $i$`]` |
| *beforeEnd*: | `aload` $n-1$ |
| | ⋮ |
| | `aload` 0 |
| | `ldc` *SecState* |
| | `monitorexit` |
| *invoke*: | `invokevirtual` *c.m* |
| *invokeDone*: | `ldc` *SecState* |
| | `monitorenter` |
| | `astore` $n$ |
| *afterG*$_1$: | `[eval after` $G_1$`]` |
| | `ifeq` *afterG*$_2$ |

| Inlined label | Instruction |
|---|---|
| | `[after update 1]` |
| | `goto` *afterEnd* |
| | ⋮ |
| *afterG*$_j$: | `[eval after` $G_j$`]` |
| | `ifeq` *exit* |
| | `[after update` $j$`]` |
| *afterEnd*: | `aload` $n$ |
| | `ldc` *SecState* |
| | `monitorexit` |
| *afterReleased*: | `goto` *done* |
| *exceptionalG*$_1$: | `ldc` *SecState* |
| | `monitorenter` |
| | `[eval exceptional` $G_1$`]` |
| | `ifeq` *exceptionalG*$_2$ |
| | `[exceptional update 1]` |
| | `goto` *exceptionalEnd* |
| | ⋮ |
| *exceptionalG*$_k$: | `[eval exceptional` $G_k$`]` |
| | `ifeq` *exit* |
| | `[exceptional update` $k$`]` |
| *exceptionalEnd*: | `ldc` *SecState* |
| | `monitorexit` |
| *exceptionalReleased*: | `athrow` |
| *exit*: | `iconst` $-1$ |
| | `invokestatic` *System.exit* |
| *done*: | |

Added entries in exception handler array:

| *From* | *To* | *Target* | *Type* |
|---|---|---|---|
| *invoke* | *invokeDone* | *exceptionalG*$_1$ | *any* |
| *L* | *exceptionalReleased* | *exit* | *any* |
| *exit* | *done* | *exit* | *any* |

**Fig. 4.** The inlining replacement of $L$: `invokevirtual` *c.m*.

any security-relevant method *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 the issues surrounding inheritance and dynamic binding. These are orthogonal to the results of this paper, and it has been described elsewhere how to deal with them [10].

The inliner we propose, $\mathcal{I}_{Ex}$, replaces each instruction $L$ : `invokevirtual` *c.m* where *c.m* is security-relevant by JVML code corresponding to the code in Figure 4. The replacement contains blocks of code to update the security state according to the before, after and exceptional clauses respectively. These three blocks are referred to as blocks of inlined code. The security state is maintained as static fields of an auxilliary class called *SecState*, created by the inliner. The inliner locks the security state by acquiring the lock associated with the *SecState*

class, and stores arguments to the method call for use in event handler code. 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.

The Java Virtual Machine Specification [24] states that some unchecked exceptions such as `InternalError` or `UnknownError` can occur at any instruction. In the theoretical development, we will ignore this possibility, i.e. we assume an error-free JVM. Our implementation defensively catches any such exception and exits the program. With such an implementation, security is guaranteed even on JVM's that do throw such exceptions, but clearly transparency is no longer guaranteed should the JVM not be error-free.

### 7.2 Correctness Properties

In this section we show that the inliner presented above is conservative, transparent, and secure for race-free policies. In view of theorem 2 this is the best we can do: The assumption of race freedom cannot be lifted without losing transparency. As mentioned, other design choices are possible: For instance we may choose to lock across the security relevant call [13]. Such a design choice sacrifices transparency in favour of security.

To first prove security, the key observation is the following: While the sequence of actions seen by the monitor might be different from the sequence of actual actions happening, the second is actually a client-order preserving permutation of the first. And hence, by the definition of race-free policy, if the first is accepted by the monitor, then the second is necessarily also accepted by the policy. So if the monitor allows the execution, it is actually compliant with the policy.

**Theorem 3.** *The example inliner $\mathcal{I}_{Ex}$ is secure for race-free policies.*

The full proof of the theorem is provided in the appendix of this paper. 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. Moreover, such an execution and its erasure have the same observable trace of actions, hence conservativity follows.

**Theorem 4.** *$\mathcal{I}_{Ex}$ is conservative.*

Again, a full proof is provided in the appendix. Finally, for transparency:

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

*Proof.* Consider a policy-adherent execution $E$ of $P$. Insert policy checking steps into $E$ to obtain a sequence of configurations $E'$. Then $E'$ is an execution of the inlined program. This follows, by induction on the length of $E$, from the fact that $E$ adheres to the policy. □

## 8 Case Studies and Benchmarks

The inlining algorithm described above has been implemented in Java using the ASM framework [25]. We present some results and benchmarks of this inliner in four case studies. The inliner was designed and implemented as part of the S$^3$MS project, a project that investigates the applicability of inlined reference monitoring for Java applications on mobile phones. Hence, the case studies are all Java Micro Edition applications. The applications and the corresponding security policies are available at `http://www.csc.kth.se/~landreas/inlining`. The inlining was performed off-device on an Intel Core 2 CPU at 1.83 GHz with 2 Gb memory. All policies were successfully enforced by our inliner.

**ImageExchange** (IE) ImageExchange is a combined server/client application that allows users to exchange images over a Bluetooth connection.
The policy in this case study restricts the program to only send the file that was last approved by the user. We adapt the bluetooth and gui API's slightly to allow this policy to be conveniently formulated.

**Snake** (SN) This is a classic game of snake in which the player may submit the current score to a server over a network connection.
The policy prevents data from being sent over the network after reading from phone memory.

**MobileJam** (MJ) The MobileJam application is a Bluetooth GPS based traffic jam reporter which utilizes the online Yahoo! Maps API.
The policy prevents the application from connecting to any URLs other than those starting with `http://local.yahooapis.com`.

**BatallaNaval** (BN) BatallaNaval is a multiplayer battleship game that communicates through SMS messages.
In this case the policy restricts the number of sent SMS's to a constant.

The benchmarks for the case studies are summarized in table 1. When the security relevant methods perform IO the runtime overhead of the monitor is dwarfed by the IO overhead, and is too small to be measured. Since most policies talk about methods that perform IO, it is fair to say that in practice, there is close to no performance penalty.

| | IE | SN | MJ | BN |
|---|---|---|---|---|
| Security Relevant Invokes | 2 | 2 | 4 | 2 |
| Original Size of Binaries (kb) | 35.2 | 23.2 | 196.2 | 210.7 |
| Inlining Duration (s) | 0.56 | 0.49 | 1.84 | 1.42 |
| Size increase (%): | 1.1 | 0.7 | 4.0 | 0.9 |

**Table 1.** Benchmarks for the case studies.

To determine the runtime overhead impact of inlining more precisely, a program that invoked an empty dummy security relevant method in a loop was

constructed. The execution time of this loop was then measured before and after inlining. The inlining caused the execution time to increase from 407 ms to 1358 ms when the loop was iterated $10^6$ times. This indicates an overhead in this experiment of 951 nanoseconds per security relevant call. This includes the time needed to do call disambiguation in the presence of dynamic binding (something we left out of scope for the theoretical study, see Section 7). Given that security relevant calls in our framework typically occur at session rate, this suggest that the runtime overhead of inlining is in practice negligible.

To summarize, our experiments support the existing evidence [5, 6] that inlining is a practical enforcement technique, even in a multithreaded setting.

## 9 Conclusions and Future Work

Inlining is a powerful and practical technique to enforce security policies. Several implementations of inliners exist, even for multithreaded programs. Hence, the study of the correctness 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 that do allow effective enforcement by inlining, and we have exhibited a concrete inlining algorithm which satisfies the required correctness properties.

A number of extensions of this work merit attention. First, we do not yet address inheritance. This extension 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 [10] for C#. We do not expect any issues to carry this over to the multithreaded setting.

Another interesting direction is to consider proof-carrying code (PCC) for monitor inlining. The advantage of such a framework would be to allow inlining to be performed outside the application loader's trust boundary. We have already realized this for the case of sequential Java, and an extension to multithreaded Java is currently under way.

**Acknowledgments** Thanks to Irem Aktug, Dilian Gurov and Dries Vanoverberghe for useful discussions on many topics related to monitor inlining. Thanks to Jan Smans and Fabio Massacci for providing useful feedback on a draft of this paper.

The work of Dam, Lundblad, and Piessens was partially supported by the S$^3$MS project. Bart Jacobs is a Postdoctoral Fellow of the Research Foundation

- Flanders (FWO). Jacobs and Piessens were partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy. Dam holds a research position with the Swedish Research Council (VR) and is affiliated with the VR Linnaeus Center ACCESS. Lundblad was partially supported by VR grant 2007-6436.

## References

1. Evans, D., Twyman, A.: Flexible policy-directed code safety. In: IEEE Symposium on Security and Privacy. (1999) 32–45
2. Erlingsson, U., Schneider, F.B.: SASI enforcement of security policies: a retrospective. In: Proc. Workshop on New Security Paradigms (NSPW '99), New York, NY, USA, ACM Press (2000) 87–95
3. Schneider, F.B.: Enforceable security policies. ACM Trans. Information and System Security **3**(1) (2000) 30–50
4. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Computability classes for enforcement mechanisms. ACM Trans. Program. Lang. Syst. **28**(1) (2006) 175–205
5. Erlingsson, U.: The inlined reference monitor approach to security policy enforcement. PhD thesis, Dept. of Computer Science, Cornell University (2004)
6. Bauer, L., Ligatti, J., Walker, D.: Composing security policies with polymer. In: PLDI. (2005) 305–314
7. Hamlen, K.W., Morrisett, G., Schneider, F.B.: Certified in-lined reference monitoring on .NET. In: PLAS. (2006) 7–16
8. Erlingsson, Ú., Schneider, F.B.: IRM enforcement of Java stack inspection. In: IEEE Symposium on Security and Privacy. (2000) 246–255
9. Aktug, I., Dam, M., Gurov, D.: Provably correct runtime monitoring. In: Proc. of 15th Int. Symposium on Formal Methods (FM '08). (May 2008) 262–277
10. Vanoverberghe, D., Piessens, F.: A caller-side inline reference monitor for an object-oriented intermediate language. In: FMOODS. (2008) 240–258
11. Chen, F., Rosu, G.: Java-MOP: A monitoring oriented programming environment for Java. In: TACAS. (2005) 546–550
12. Hamlen, K.W., Jones, M.: Aspect-oriented in-lined reference monitors. In: PLAS. (2008) 11–20
13. Dam, M., Jacobs, B., Lundblad, A., Piessens, F.: Provably correct inline monitoring for multithreaded Java-like programs. Journal of Computer Security (2009)
14. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Sec. **4**(1-2) (2005) 2–16
15. Ligatti, J.A.: Policy Enforcement via Program Monitoring. PhD thesis, Princeton University (2006)
16. Viswanathan, M.: Foundations for the run-time analysis of software systems. PhD thesis, University of Pennsylvania (2000)
17. Verhanneman, T., Piessens, F., De Win, B., Joosen, W.: Uniform application-level access control enforcement of orginzationwide policies. In: Twenty-First Annual Computer Security Applications Conference. (2005) 389–398
18. Dantas, D.S., Walker, D.: Harmless advice. In: POPL. (2006) 383–396
19. Shah, V., Hill, F.: An aspect-oriented security framework. In: Proceedings of the DARPA Information Survivability Conference. (2004) 143–145
20. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java Language Specification, Third Edition. Prentice Hall (2005)

21. Jeffrey, A., Rathke, J.: Java Jr: Fully abstract trace semantics for a core Java language. In: ESOP. (2005) 423–438
22. Jeffrey, A., Rathke, J.: A fully abstract may testing semantics for concurrent objects. Theor. Comput. Sci. **338**(1-3) (2005) 17–63
23. Aktug, I., Naliuka, K.: ConSpec – a formal language for policy specification. Electron. Notes Theor. Comput. Sci. **197**(1) (2008) 45–58
24. Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1999)
25. ObjectWeb: Asm - home page (February 2008)
26. Freund, S.N., Mitchell, J.C.: A type system for object initialization in the Java bytecode language. ACM Trans. Program. Lang. Syst. **21**(6) (1999) 1196–1250
27. Leroy, X.: Java bytecode verification: Algorithms and formalizations. J. Autom. Reasoning **30**(3-4) (2003) 235–269

## Appendix

This appendix contains the definitions for our formal model of the Java Virtual Machine, and proofs of the security and conservativity theorems for our example inliner.

### 9.1 Formal Model of the JVM

We assume that the reader is familiar with Java bytecode syntax, the Java Virtual Machine (JVM), and formalisations of the JVM such as [26]. Here, we only present components of the JVM, that are essential for the definitions in the rest of the text. A few simplifications have been made in the presentation. In particular, to ease notation a little we ignore issues concerning overloading.

**Preliminary Conventions** We use $c$ for class names, $m$ for method names, and $f$ for field names. For our purpose it suffices to think of class names as fully qualified.

To each method is associated a method definition as 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 transfer control to address $t$, if the handler is the topmost handler in the exception handler array that handles the instruction for the given type.

The set of values (of Java primitives and object references) is ranged over by $v$. Values of object type are (typed) locations $o$, 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 field references.

**Configurations and Transitions** A *configuration* $C = (h, \Lambda, \Theta)$ of the JVM consists of a *heap* $h$, a *lock map* $\Lambda$ which maps an object $o$ to a thread id *tid* iff *tid* holds the lock of $o$, and a *thread configuration map* $\Theta$ which maps a thread identifier *tid* to its thread configuration $\Theta(tid) = \theta$. A thread configuration $\theta$ is a stack $R$ of activation records. For normal execution, the activation record at the top of an execution stack has the shape $(M, pc, s, l)$, where:

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

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.

Activation records for API calls are special and are discussed below.

**Definition of the transition relation** 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 use 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 $M$, heap $h$, throwable $o$ and pc $pc$ in the standard way:
  $handler(M, h, o, pc) = handler2(H, h, o, pc)$ with $H$ the exception handler array of $M$
  $handler2(\epsilon, h, o, pc) = \bot$
  $handler2((b, e, t, c) \cdot H, h, o, pc) =$
  $$\begin{cases} t & \text{if } b \leq pc < e \text{ and } h \vdash o : c \\ handler2(H, h, o, pc) & \text{otherwise} \end{cases}$$
- $\mathbf{v}$ 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 $(\square)$ (used for API calls).

*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*

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

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

*Locking instructions*

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

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

*Exceptional Transitions*

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

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

*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.

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

Exceptional return from an API method:

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

Normal return from an API method:

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

**Programs and Executions** For the purpose of this paper we can view a *program* $P$ as a set of class declarations determining types of fields and methods belonging to classes in $P$, and a method environment assigning method definitions to each method in $P$. An *execution* $E$ of a program $P$ is a (possibly infinite) sequence of JVM configurations $C_0C_1\ldots$ where $C_0$ is an initial configuration consisting of a single thread with a single, normal activation record with an empty stack, no local variables, $M$ as a reference to the main method of $P$, $pc = 0$, and for each $i \geq 0$, $C_i \rightarrow_{\text{JVM}} C_{i+1}$. We restrict attention to configurations that are *type safe*, in the sense that heap contents match the types of corresponding locations, and that arguments and return/exceptional values for primitive operations as well as method invocations match their prescribed types. The Java bytecode verifier serves, among other things, to ensure that type safety is preserved under machine transitions (cf. [27]).

**Thread creation** To support thread creation we assume that there is a distinguished API method that has, besides the standard effect of an API call discussed above, an additional side effect of creating an additional thread in the configuration. The newly created thread starts with a single normal activation record initialized to call the run() method of the object passed as a parameter to the API method.

### 9.2 Proof of the Security Theorem

Since our inliner does not synchronize across security-relevant API method calls, it is not guaranteed that updates to the inlined security state are completely synchronized with the actual security relevant actions. For instance, if two security relevant method invocations $m_1$ and $m_2$ happen concurrently, the following scenario is possible. First, the inlined code before the $m_1$ call is executed, then the inlined code before the $m_2$ call is executed, then $m_2$ is invoked, and then $m_1$ is invoked. In other words, the sequence of actions as considered by the monitor might not be equal to the sequence of actions as it actually happens. An immediate consequence of this is that some policies cannot be enforced securely by our inliner: for instance the policy in Figure 2 can not be securely enforced.

Fortunately, for the class of race-free policies, we can show that our inliner is secure. The key observation is the following: while the sequence of actions seen by the monitor might be different from the sequence of actual actions happening, the second is actually a client-order preserving permutation of the first. And hence, by the definition of race-free policy, if the first is accepted by the monitor, then the second is necessarily also accepted by the policy. So if the monitor allows the execution, it is actually compliant with the policy. We set out to prove this.

First some notation: We have to distinguish clearly between the actual security relevant API actions (the *observable actions* of the program invoking the API) and the execution of the corresponding *monitor actions* (the inlined code manipulating the inlined security state). We use the notation $\text{mon}(\alpha)$ for the monitor action corresponding to the observable action $\alpha$. We define a monitor

action to take place at the step in the execution that performs the inlined monitorexit instruction. We refer to these points in an execution as the *policy commit points*.

The policy commit points can be seen as the points where the monitor "sees" an observable action: at the policy commit point, the changes to the inlined security state for a given observable action are made visible by releasing the lock on the inlined security state.

An execution $E$ now gives rise to two traces: the trace of the actual security relevant observable actions $T_s$, and the trace of the monitor actions $T_m$. In addition, the given execution $E$ determines an ordering that allows us to merge these two traces into the full trace $T_f$.

For example, in the scenario discussed above, if we let $\alpha_1$ be the observable action of calling $m_1$ and $\alpha_2$ the observable action of calling $m_2$, then the trace $T_f = \text{mon}(\alpha_1), \text{mon}(\alpha_2), \alpha_2, \alpha_1$ is the full trace that illustrates that observable actions and monitor actions can occur in different orders.

**Lemma 2.** *The trace $T_m$ of monitor actions in an inlined program always complies with the policy.*

*Proof.* All updates to the security state are done under a single lock, and hence can be serialized. Since the actions seen by the monitor correspond to the monitorexit steps on that single lock, they are synchronized with the updates to the security state. So this lemma is equivalent to saying that the inlined code correctly implements the security automaton in a sequential setting. □

For a given execution, we first want to make sure that any start actions that have been monitored but not yet executed are added to the traces $T_f$ and $T_s$. More precisely: if there are $\text{mon}(\alpha)$ actions with $\alpha$ a start action by a thread *tid* in $T_f$ such that no action by *tid* succeeds this action in $T_f$, then for any such action add $\alpha$ to the end of $T_f$ and to the end of $T_s$. Call the resulting traces $T'_f$ and $T'_s$. It follows that $T_f$ is a prefix of $T'_f$ and $T_s$ is a prefix of $T'_s$.

In a similar way, if there are end actions $\alpha$ by a thread *tid* in $T_f$ such that no action by *tid* succeeds this action in $T_f$, then add $\text{mon}(\alpha)$ to the end of $T_f$ and to the end of $T_m$. Call the resulting traces $T''_f$ and $T'_m$. It follows that $T_f$ is a prefix of $T''_f$ and $T_m$ is a prefix of $T'_m$. $T'_m$ complies with the policy because of Lemma 2, and because after actions can only update the security state, they can not break compliance with the policy.

**Lemma 3.** *For each mon($\alpha$) action with $\alpha$ a start action by a thread* tid *in $T''_f$, there is exactly one immediately succeeding action by* tid *in $T''_f$, and this is the action $\alpha$. Furthermore, for each mon($\alpha$) action with $\alpha$ an end action by* tid *in $T''_f$, there is exactly one immediately preceding action by* tid *in $T''_f$, and this is the action $\alpha$.*

*Proof.* By induction on the length of the execution $E$. □

As mentioned before, the trace $T_m$ of monitor actions is not necessarily identical to the observable trace $T_s = \omega(E)$. But we show that $T'_s$ is a client-order preserving permutation of $T'_m$.

**Lemma 4.** *Consider an execution $E$ of an inlined program. The trace $T'_s$ is a client-order preserving permutation of $T'_m$.*

*Proof.* Because of Lemma 3, we can define a function $f$ from $T'_m$ to $T'_s$ that maps each monitor action $\text{mon}(\alpha)$ to the immediately succeeding action within the same thread (for $\alpha$ a start action), or to the immediately preceding action in the same thread (for $\alpha$ an end action).

$f$ is injective, since for any start action by a thread *tid* in $T''_f$, only one action by *tid* precedes it immediately, and similarly for return actions. Because of the construction of $T'_m$ and $T'_s$, $f$ is also surjective, hence it is a bijection. Hence, when we consider $T'_m$ and $T'_s$ as sequences of observable actions (and we don't care anymore about the distinction of whether this action is seen by the monitor and hence in $T'_m$, or an actual observable action and hence in $T'_s$), $f$ is a permutation.

We show that $f$ is a client-order preserving permutation from $T'_m$ to $T'_s$. Consider an after action $i$ in $T'_m$ and a before action $j$ in $T'_m$ such that $i < j$. We must now prove that $f(i) < f(j)$.

Let us call $i_m$ and $i_s$ the injections from $T'_m$ and $T'_s$ in $T''_f$. Then $i_m(i) < i_m(j)$. We also have that, since $i$ is an after action, $i_s(f(i)) < i_m(i)$, and since $j$ is a before action, $i_m(j) < i_s(f(j))$. Therefore, we have that $i_s(f(i)) < i_s(f(j))$. Since $i_s$ is order-preserving, we have that $f(i) < f(j)$. This means $T'_s$ is a client-order preserving permutation of $T'_m$. □

**Theorem 6.** *The example inliner $\mathcal{I}_{Ex}$ is secure for race free policies.*

*Proof.* For any execution of the inlined program, by lemma 2, $T_m$ complies with the policy. Since $T'_m$ extends $T_m$ only with after actions, $T'_m$ also complies with the policy.

From Lemma 4 we know that $T'_s$ is a client-order preserving permutation of $T'_m$. Hence, by the definition of race-free policy, $T'_s$ also complies with the policy. Finally, since $T_s$ is a prefix of $T'_s$, it also complies with the policy. □

### 9.3 Proof of Conservativity

Our proof of conservativity 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. Moreover, such an execution and its erasure have the same observable trace of actions, hence conservativity follows.

To make this precise, we first define the notion of the *erasure* of an execution of an inlined program.

**Definition 5.** *Given an execution $E$ of $\mathcal{I}_{Ex}(\mathcal{S}, P)$. We define the* erasure *$E'$ of $E$ by recursion on the length of $E$. The erasure of an execution with a single configuration $C$ is $C$, with the* `SecState` *removed from the heap. Consider an execution $EC_nC'_{n+1}$. Let $E'C'$ be the erasure of $EC_n$. Let tid be the thread that performes the step $C_nC_{n+1}$. Then we define the erasure of $EC_nC_{n+1}$ as*

– $E'C'$ *if the pc of tid in* $C_n$ *points to an inlined instruction*
– $E'C'$ *followed by the configuration obtained by letting tid perform one step in the context of the original program and state* $C'$.

It follows that $E'$ is an execution of the uninlined program.

**Definition 6.** *Given an activation record* $r = (M, pc, l, s)$ *of* $\mathcal{I}_{Ex}(\mathcal{S}, P)$ *and an activation record* $r' = (M', pc', l', s')$ *of* $P$*, we say that* $r'$ corresponds to $r$ *iff*

– $M = M'$
– $l'$ *is* $l$ *without the local variables introduced by the inliner*
– *if pc points to a non-inlined instruction then* $pc'$ *points to the same instruction and* $s'$ *is equal to* $s$*, otherwise* $pc'$ *and* $s'$ *equal the states of pc and* $s$ *as they were right before entering the block of inlined code. For instance if pc equals before*$G_1$ *then* $pc'$ *equals* $L$ *and* $s'$ *equals* $l[0 \dots n-1]s[n \dots]$.

**Definition 7.** *Given a configuration* $C = (h, \Lambda, \Theta)$ *of an execution of the inlined program and a configuration* $C' = (h', \Lambda', \Theta')$ *of the original program* $P$ *we say* $C'$ corresponds to $C$ *iff*

– $h'$ *is the heap obtained by removing the* `SecState` *from* $h$
– $\Lambda'$ *is the lock map obtained by removing the* `SecState` *from* $\Lambda$
– $Dom(\Theta') = Dom(\Theta)$ *and for each* $(tid, R) \in \Theta$ *there is an* $R'$ *such that* $(tid, R') \in \Theta'$, $|R| = |R'|$ *and for each* $i \in [0, |R|)$, $R'_i$ *corresponds to* $R_i$.

**Lemma 5.** *Given a partial execution* $EC$ *of the inlined program* $\mathcal{I}_{Ex}(\mathcal{S}, P)$*, then for the erasure* $E'C'$ *of* $EC$ *it holds that* $C'$ *corresponds to* $C$ *and* $\omega(E'C') = \omega(EC)$.

*Proof.* By induction on the length of $EC$. The base case is trivial. Consider an execution $EC_nC_{n+1}$ of the inlined program. Let $E'C'$ be the erasure of $EC_n$. By the induction hypothesis, we may assume that $C'$ corresponds to $C_n$ and that $\omega(E'C') = \omega(EC_n)$. We have two cases

(1) if $C_nC_{n+1}$ is an execution of an inlined instruction then the erasure of $EC_nC_{n+1}$ equals $E'C'$. We prove that $C'$ corresponds to $C_{n+1}$ and that $\omega(EC_nC_{n+1}) = \omega(E'C')$ by case analysis on the label of the inlined instruction.
(2) otherwise, let $E'C'C''$ be the erasure of $EC_nC_{n+1}$. We prove that $C''$ corresponds to $C_{n+1}$ and that $\omega(E'C'C'') = \omega(EC_nC_{n+1})$ by case analysis on the non-inlined instruction.

□

**Theorem 7.** $\mathcal{I}_{Ex}$ *is conservative.*

*Proof.* For any execution of the inlined program, Lemma 5 gives us an execution of the uninlined program with the same trace. □