Project No: **FP7-231620**

Project Acronym: **HATS**

Project Title: **Highly Adaptable and Trustworthy Software using Formal Models**

Instrument: **Integrated Project**

Scheme: **Information & Communication Technologies**

**Future and Emerging Technologies**

**Deliverable D3.4**

**Evolvability at Bytecode Level**

Due date of deliverable: (T0+24)

Actual submission date: 1st March 2011

Start date of the project: **1st March 2009**

Duration: **48 months**

Organisation name of lead contractor for this deliverable: **KTH**

---

<table>
<thead>
<tr>
<th>Dissemination level</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PU Public</td>
<td>✓</td>
</tr>
<tr>
<td>PP Restricted to other programme participants (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>RE Restricted to a group specified by the consortium (including Commission Services)</td>
<td></td>
</tr>
<tr>
<td>CO Confidential, only for members of the consortium (including Commission Services)</td>
<td></td>
</tr>
</tbody>
</table>
Executive Summary: 
Evolvability at Bytecode Level

This document summarises deliverable D3.4 of project FP7-231620 (HATS), an Integrated Project supported by the 7th Framework Programme of the EC within the FET (Future and Emerging Technologies) scheme. Full information on this project, including the contents of this deliverable, is available online at http://www.hats-project.eu.

This deliverable reports on the investigations of Task 3.4, Evolvability at Bytecode Level. The goal of this task is to develop runtime code evolution at the bytecode level through the use of software transformation techniques in the style of monitor inlining. The deliverable also reports on an experiment showing how deductive techniques can be used to automatically refine ABS models into bytecode. The main results are:

1. Extensions and adaptations of a security policy specification language ConSpec, developed in the context of an earlier project, to the case of multithreaded Java bytecode, and to the ABS language.

2. An investigation of correctness properties attainable in a multithreaded setting, with strong characterization results for several cases.

3. Adaptation (inlining) algorithms for both Java bytecode and ABS that are (in a suitable technical sense) optimal and proven correct at the bytecode level.


5. A number of case studies showing both applications of our techniques on real Java bytecode, and on ABS code examples.

6. A new approach to property preserving compilation exemplified with ABS-to-Java bytecode compilation.

The case studies are publicly available at http://www.csc.kth.se/~landreas/mt_inlining/

The main parts of the deliverable on runtime code evolution has been performed as a collaboration between KTH and KUL. The work on property preserving compilation has been carried out by CTH.

List of Authors
Richard Bubel (CTH)
Mads Dam (KTH)
Ran Ji (CTH)
Andreas Lundblad (KTH)
Lei Mou (KTH)
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 Introduction</strong></td>
<td>5</td>
</tr>
<tr>
<td>1.1 Overview</td>
<td>7</td>
</tr>
<tr>
<td><strong>2 Monitor Inlining as a Mechanism for Evolvability</strong></td>
<td>8</td>
</tr>
<tr>
<td>2.1 Notations</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Monitor Specifications</td>
<td>8</td>
</tr>
<tr>
<td>2.3 Reference Monitor Inlining</td>
<td>10</td>
</tr>
<tr>
<td>2.3.1 Inlining Correctness Properties</td>
<td>10</td>
</tr>
<tr>
<td>2.3.2 Complete Mediation</td>
<td>11</td>
</tr>
<tr>
<td>2.4 Limitations of Inlining in a Multithreaded Setting</td>
<td>11</td>
</tr>
<tr>
<td>2.5 Race-free Monitors</td>
<td>13</td>
</tr>
<tr>
<td>2.6 An Inlining Algorithm for Java Bytecode</td>
<td>13</td>
</tr>
<tr>
<td>2.6.1 Dynamic Method Resolution</td>
<td>14</td>
</tr>
<tr>
<td>2.6.2 Pseudo Code</td>
<td>14</td>
</tr>
<tr>
<td>2.6.3 Correctness</td>
<td>14</td>
</tr>
<tr>
<td>2.6.4 Certification</td>
<td>15</td>
</tr>
<tr>
<td>2.7 An Inlining Algorithm for ABS</td>
<td>18</td>
</tr>
<tr>
<td>2.7.1 ConSpec_{ABS}</td>
<td>18</td>
</tr>
<tr>
<td>2.7.2 Inlining Scheme</td>
<td>19</td>
</tr>
<tr>
<td>2.7.3 Monitoring of Synchronous vs Asynchronous events</td>
<td>19</td>
</tr>
<tr>
<td>2.7.4 Example Inlining</td>
<td>21</td>
</tr>
<tr>
<td>2.7.5 Monitor Event Ambiguities</td>
<td>22</td>
</tr>
<tr>
<td>2.7.6 Resolving Runtime Type Information</td>
<td>23</td>
</tr>
<tr>
<td>2.7.7 Correctness</td>
<td>24</td>
</tr>
<tr>
<td>2.8 Case Studies</td>
<td>25</td>
</tr>
<tr>
<td>2.8.1 Trading System</td>
<td>25</td>
</tr>
<tr>
<td>2.8.2 Swing API-usage</td>
<td>25</td>
</tr>
<tr>
<td>2.8.3 Session Security</td>
<td>26</td>
</tr>
<tr>
<td>2.8.4 HTTP Authentication Enforcement</td>
<td>27</td>
</tr>
<tr>
<td>2.8.5 URL redirection</td>
<td>30</td>
</tr>
<tr>
<td>2.8.6 Summary</td>
<td>30</td>
</tr>
<tr>
<td><strong>3 Compiling ABS into Java Bytecode</strong></td>
<td>32</td>
</tr>
<tr>
<td>3.1 Background</td>
<td>32</td>
</tr>
<tr>
<td>3.1.1 Sequent Calculus</td>
<td>33</td>
</tr>
<tr>
<td>3.2 Extending the Calculus</td>
<td>35</td>
</tr>
<tr>
<td>3.3 Example</td>
<td>38</td>
</tr>
<tr>
<td>3.4 Conclusion</td>
<td>39</td>
</tr>
<tr>
<td><strong>4 Conclusion</strong></td>
<td>41</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

During the maintenance phase of the software life-cycle, the software evolves from version to version. The updates to the software may vary from small patches to large improvements and new features. Generally speaking, the software needs to be recompiled/redeployed upon changes in the code base. For small, independent changes (bug fixes, security patches, modifications to logging, etc.), however, the updates may be applied directly to the bytecode, sometimes even during runtime. This is what we refer to as evolvability at bytecode level.

Bytecode evolvability may be associated with versioning, but it is equally possible to view evolvability at the bytecode level as part of a continuing process of adapting a piece of running software to new operating conditions and new requirements, for instance, by imposing new security constraints that are not easily managed by associated runtime support systems such as reference monitors.

The idea of bytecode evolvability is realized by rewriting the software directly at the bytecode level. In effect, we view evolvability as a bytecode level program transformation. There are many reasons why one might want to perform such a transformation. Performance optimizations, for instance, may be applied to obtain some desired latency or throughput properties, but otherwise leave functional behavior unchanged. This type of behavior preserving program transformation is the classical object of study in this area, and we will have occasion to address this issue in Task 3.5 “Autonomously Evolving Systems” when we come to address runtime self-adaptation, in particular, in relation to performance. The second type of program transformation, behavior transforming program transformation, has had less attention in the literature so far, and it is the primary subject of study in this deliverable, as we want to view evolvability as deliberate changes in functionality applied on a running software system, in order to achieve some externally observable effect, such as enforcing a new security policy. The question of key importance, then, is this:

How are properties of a software system affected by imposing a given behavior transforming program transformation?

Clearly, for a transformation aiming to enforce a given security policy (the example we study), the transformed program should satisfy the required policy, but we would expect/hope for other properties too, such as:

Conservativity After the transformation, does the program possess new secure and observable behavior it did not already possess before the transformation was applied?

Transparency If the program possessed some behavior before the transformation was applied, and that behavior did not contradict the required security policy, does the program after transformation still possess that behavior?

Note that these requirements are by no means cast in stone. Conservativity, for instance, is in many applications not desirable, as the very objective of performing the transformation may be to enable new behavior in some predictable way. However, for security enforcement, which is our main focus here, conservativity is often a reasonable goal, as the objective of the security enforcement mechanism is to terminate execution.
when a security violation is encountered, i.e., the only effect of the transformation should be to truncate the event stream.

Large parts of the deliverable are geared to answering the question above, for specific classes of transformations, and in particular, examining how concurrency models such as the Java memory model affect the kind of answers these questions can be given. As main outcome of the work we present strong results characterizing the type of correctness properties that can be attained using bytecode inlining for the case of multithreaded Java. The results typically come in two parts: A negative result showing that a certain property cannot be obtained with certain minimal assumptions on the inliner (specifically, whether it is able to block access to a given API method or not), and a positive result exhibiting a provably correct inliner which satisfies that minimal assumption.

The behavior transforming program transformations ("updates") are given in suitable domain specific languages. In this work we use variants of the language ConSpec for security policy ("contract") specification, originally developed in the context of an earlier EU project S$^3$MS. The general scheme is shown in Figure 1.1. Evolution is attained by parsing and compiling these updates into JVM or ABS code snippets, and then inlining these code snippets into the application. This is described in detail in this document and in the appendices.

![Figure 1.1: Bytecode / ABS rewriting scheme.](image)

For Creol, a predecessor of the ABS language, runtime deployment of the program transformations developed in the present task could be implemented by relying on the techniques for runtime class updates developed in Task 3.1. Some experimental solutions for runtime update of Java bytecode are also available, we refer to deliverable D3.1b for a discussion of the state of the art in this area.

Some updates are harder to confine and realize through bytecode rewriting than others, especially on live systems. Thus, for tractability, we for instance restrict our attention to updates that add state and code to the program. For the theoretical part of our study, we also put a restriction on the side-effects of the added code to be limited to the added state, plus termination of the program.

We also address the issue of certification. In many applications it is of interest to provide inlining as a tool for quality assurance available to the software developer, independent of when and under whose jurisdiction the software is eventually deployed. This can be achieved by public key certification in the usual way, but public key certification is only capable of verifying a signature, and not the actual compliance of the software with respect to some given policy. For this reason it may be of interest to develop certification methods in the style of proof-carrying code, which is capable of certifying correctness/security at a semantical level. We have investigated ways of implementing this approach for the case of sequential Java bytecode [12]. This has later been extended and simplified for the case of multithreaded Java bytecode [11].

A number of case studies have been developed mainly based on software updates which provide security patches for various known attacks.

The main body of work in the report addresses Java bytecode, as this is the setting which is of most practical relevance from a security perspective. We have also developed an inliner for the ABS language. This inliner is much simpler than the JVM inliner, as should be expected due to the much simpler runtime and concurrency models. However, the ABS type system turns out to add an interesting twist to the story.
The work is very much related to reference monitor inlining and the techniques we use to describe and implement updates are similar to aspect oriented programming (AOP). In this report we target both Java bytecode and ABS. Some technical details such as formal program models have been left out for brevity. For a detailed discussion we refer to the papers ([11, 10, 12]) found in Appendices A, B and C.

### 1.1 Overview

Chapter 2 discusses *Monitoring as a Mechanism for Evolvability*. This topic comprises all aspects of monitor inlining from an evolvability perspective. Most parts of the section apply to monitoring and monitor inlining in general, while some sections discuss details specific to our two target languages: Java bytecode and ABS. More specifically, the chapter covers the following subtopics:

**Reference Monitors** (Section 2.2) Reference Monitors are a special type of automata that define the allowed behavior of the program being monitored. The monitor defines its own state together with a set of update rules based on monitored events. The granularity of events can vary from field read and writes, to higher level protocol events. The choice in our case is on the level of API interaction in the form of API calls, returns and exceptional returns. In our work we have chosen to present the definitions of reference monitors using the ConSpec language which is also explained briefly.

**Monitor Inlining Algorithms** (Sections 2.6 and 2.7) Rewriting applications to embed a reference monitor is a delicate process. This section provides formal description of the properties of correct inlining: *Security* (the behavior of the client program should never violate the monitor), *conservativity* (no behavior should be *added* to the client program) and *transparency* (no behavior should be *removed* from the client program).

**Concurrency** (Section 2.4) To monitor calls and returns of API methods using inlined reference monitors may be problematic in the context of multiple threads (or concurrent object groups). In fact, as shown in the report, it is impossible to perform correct inlining (with respect to the properties mentioned above).

**Certification** (Section 2.6.4) In situations where inlining is performed by the executing host himself, the correctness properties mentioned above are enough to ensure a secure environment. In other situations however, when inlining is performed by someone else, there is a need to convey the guarantee that correct inlining has been performed to the executing host. This can be achieved through proof-carrying code [25]. This report discusses two approaches; one for the sequential setting, and another, higher level, for the multithreaded setting.

**Case Studies** (Section 2.8) The algorithms for Java bytecode and ABS presented in previous sections have been implemented and evaluated in five case studies of varying characteristics. This report provides descriptions of these together with a summary of the results.

Chapter 3 covers the process of compiling ABS into Java bytecode. In this chapter we focus on property preserving compilation of sufficiently refined ABS models into Java bytecode. The basic idea is to extend the program logic under consideration in such a way that the calculus rules synthesize a program at the same time in a give target language (here: Java bytecode). In Section 3.1 we introduce the basic notions necessary to understand the approach. Section 3.2 introduces our calculus extension and selected rules to treat the different aspects of ABS. Section 3.3 applies the presented approach on a small example clarifying the basic idea.

Finally, Chapter 4 reflects on the achieved results and concludes the document.
Chapter 2

Monitor Inlining as a Mechanism for Evolvability

Monitor inlining \[19, 17\] is a well established technique for enforcing security properties through code rewriting. The conceptual model is simple: the description of a patch enforcing some security property is parsed and compiled into variable declarations and snippets of code. These parts (referred to as the reference monitor) are then inlined into the application at appropriate places in an aspect-oriented style. The result is a modified, self monitoring application that adheres to the provided security property.

The main applications for monitor inlining are security patching and monitoring, but as will be shown in this chapter, this technique is also useful in non-security critical applications as well, such as testing.

In the setting we study here, we regard API calls and returns as the monitorable events. Furthermore, the application rewriting is restricted to the client program, and may not alter the implementation of the API.

One of the main challenges is the treatment of concurrent programs. Due to the fact that the monitoring occurs at the application level and that the inliner is restricted to rewrite the client code, the presence of multiple threads can be problematic. The reason is that the monitoring of the event, and the event itself cannot be performed atomically. This will be discussed in detail in Section 2.4.

2.1 Notations

For brevity the full program model has been omitted in this report. For details we refer to the appendix. A few central notations regarding executions follow.

A configuration, \(C\), consists of a heap, mapping fields to values, a lock map, mapping locked objects to threads, and a set of per-thread activation records. An execution is a sequence of configurations \(E = C_1, \ldots, C_n\) such that for each \(i = 1 \ldots n - 1\), \(C_i \xrightarrow{\alpha} C_{i+1}\) where \(\rightarrow\) denotes the JVM configuration transition relation and \(\alpha\) an action such as an API call or return, or an internal action (denoted by \(\tau\)). The trace \(T = \omega(E)\) of \(E\) is the sequence \(\alpha_0 \ldots \alpha_{n-1}\) with \(\tau\) actions removed, and \(T(P) = \{\omega(E) \mid E\text{ is an execution of }P\}\).

2.2 Monitor Specifications

The monitor descriptions that guide the rewriting process are specified in the ConSpec language \[1\]. This is basically a domain specific language used for specifying security policies. Strictly speaking, ConSpec specifications may only invoke pure methods. For tractability, this is also what we restrict our attention to in the theoretical parts of this report. By lifting this constraint on method calls to arbitrary method calls, however, the language suits a wider range of software updates. (Our prototype tools have no restrictions on which methods are allowed to be called.)
ConSpec is similar to Erlingsson’s PSLang [16], but it describes conditionals and state updates in a small purpose-built expression language instead of the object language (Java, for PSLang) itself. ConSpec specifications implicitly define an automaton by providing a representation of a monitor state together with a set of clauses describing how the state is affected by the occurrence of API call and return events.

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

\[
\text{modifier \ [Type } y = \] \text{Class.method(} Type_1 x_1, \ldots, Type_n x_n \text{)} \\
\quad \text{PERFORM } \text{guard}_1 \rightarrow \{ \text{update}_1 \} \ldots \text{guard}_m \rightarrow \{ \text{update}_m \} \quad \text{[ELSE } \{ \text{update} \} \text{]} \\
\]

where \text{modifier} is either \text{BEFORE}, \text{AFTER} or \text{EXCEPTIONAL}, and \text{Type}, \text{Type}_1, \ldots, \text{Type}_n are the return and argument types of \text{Class.method}. \text{BEFORE} rules refer to pre-actions, and \text{AFTER} and \text{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 monitor defines a rule for a method (of a given signature and modifier type), the method is said to be \text{monitor-relevant}. There is at most one rule defined for each of the three event modifiers. The return value specification is absent for \text{BEFORE} rules. Each \text{clause} of the shape \text{guard}_i \rightarrow \{ \text{update}_i \}, or the \text{clause} \text{ELSE } \{ \text{update} \} \text{expresses a (conditional) update of the monitor state in the obvious way. The \text{ELSE} clause is syntactic sugar for a clause with a constantly true guard and is mandatory for \text{AFTER} and \text{EXCEPTIONAL} rules. Hence a monitor can never forbid a return from an API method.}

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 so no \text{ELSE} clause is present) the rule is not allowed to fire. Since ConSpec has been developed with security properties in mind, this indicates a forbidden behavior and program execution should be terminated.

We describe the expression syntax only by example in this report. Additional examples are given in Section 2.4 below. The syntactical details are not critical.

**Example 1.** The monitor specification in Figure 2.1 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 monitor-relevant methods, \text{JOptionPane.showConfirmDialog} and \text{BluetoothToolkit.sendFile}. The specification uses the following three helper functions which we leave undefined:

- \text{goodFileQuery(query)} returns true iff \text{query} is a well formulated file send query, for instance because it matches a predefined pattern.
- \text{queryRequestor(query)} and \text{queryFile(query)} returns the requestor and file substrings of \text{query} respectively.

**Example 2.** The monitor specification in Figure 2.2 expresses that \text{C.initialize} can only be invoked once for each thread.

**ConSpec Semantics** Observable actions of a monitored program include API method calls (pre-actions) and normal and exceptional API method returns (normal and exceptional post-actions). The pre-actions and normal and exceptional post-actions and are written as \((tid, c.m, o, v)^\uparrow\), \((tid, c.m, o, v)^\downarrow\) and \((tid, c.m, o, v)^\downarrow\) respectively, where \text{tid} denotes the current thread identifier, \text{c.m} the API method in question, \text{o} the callee, and \text{v} the arguments of the method call.
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 {
}

Figure 2.1: A security specification example written in ConSpec.

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

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

Figure 2.2: Accessing the current thread identifier in ConSpec.

A monitor \( M \) defined in ConSpec specifies a deterministic automaton \( (Q, \Sigma, \delta, q_0) \), explained below, which observes an execution of some client program and changes state, and potentially aborts, according to the monitor 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 monitor state of \( M \) to its default, using, if necessary, a local heap. The alphabet \( \Sigma \) is the set of observable actions. The state space \( Q \) is the set of all type safe assignments to the monitor 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 monitor-relevant (of the given modifier type) the \( i+1 \)st 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 \( Prg \) is accepted by (the automaton determined by) \( M \), \( Prg \) is said to adhere to \( M \).

2.3 Reference Monitor Inlining

Formally speaking, a reference monitor inliner (or inliner for short) is a function \( I \) that for each monitor \( M \) and program \( Prg \) produces a program \( I(M, Prg) \) with embedded monitoring functionality.

2.3.1 Inlining Correctness Properties

There are three correctness properties of fundamental interest (cf. [22, 19]), namely security, conservativity and transparency.
Security states that all possible traces of the inlined program should be compliant with the monitor provided to the inliner.

**Definition 2.3.1** (Security). An inliner $I$ is secure if, for every program $Prg$ and monitor $M$, every trace of the inlined program $I(M, Prg)$ adheres to $M$, i.e.

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

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

**Definition 2.3.2** (Transparency). An inliner $I$ is transparent, if for every monitor $M$ and program $Prg$, each trace of $Prg$ that is compliant with $M$ is also a trace of the inlined program, i.e.

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

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

**Definition 2.3.3** (Conservativity). An inliner $I$ is conservative if, for every program $Prg$ and monitor $M$, every trace of the inlined program $I(M, Prg)$ is a trace of $Prg$, i.e.

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

Other correctness properties have been proposed, such as the concept of strong conservativity and relative strong conservativity, which was used in [10]. These correctness criteria refine the notion of conservativity and forbid arbitrary truncation of the traces. The latter criterion even takes ill synchronized programs (programs with data races) into account.

### 2.3.2 Complete Mediation

When monitoring API usage one has two options: Either one restricts the monitoring to events that occur at the boundary of the client application and the API, (i.e., one monitors calls from the client program to the API, and returns from the API to the client program) or, one practices so called complete mediation and monitors all interaction with the API. This means that one monitors all calls and returns from the API regardless whether it is done from the client code or from within the API itself. In our work we have limited the inliner to only rewrite client code and thus do not practice complete mediation.

### 2.4 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 property specified as explained in Section 2.2 by an external monitor implemented as part of the JVM, it is impossible to do this with an inlined monitor.

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 (IRM) cannot control the scheduling of such a thread, and this affects the enforceability of certain monitors.

Consider the monitor specification in Figure 2.3. This monitor specification states that $c.n$ may only be called after $c.m$ has been called (but not necessarily returned). So the trace $T_1 = (tid, c.m, o, v)^\dagger$, $(tid', c.n, o', v')^\dagger$ is allowed by the monitor, but the trace $T_2 = (tid', c.n, o', v')^\dagger, (tid, c.m, o, v)^\dagger$ is not. Now consider a program whose traces include both $T_1$ and $T_2$, for instance the one shown in Figure 2.5. For an inliner to exclude trace $T_2$ from this program (but keep the trace $T_1$), it would have to introduce a happens before relation between $(tid, c.m, o, v)^\dagger$ and $(tid', c.n, o', v')^\dagger$. There is, however, no way such a happens
before relation can be added by the inliner since after the call has been made, the control lies within the API method which is not to be altered. Thus the inlined program will either have both traces (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 2.4.1.** No inliner can be transparent and secure for the monitor $\mathcal{M}$ in Figure 2.3. □

Essentially, this means that an inliner targeting multithreaded Java is forced to either sacrifice transparency or security.

Security is arguably the most important property and usually the main reason for inlining in the first place. To maintain security (i.e., sacrifice transparency) the inliner could “over approximate” the locking and release the lock after the API method returned (in effect enforce the monitor in Figure 2.4). This type of inliner is referred to as a **blocking inliner** as it blocks the monitor code during the API call. The implications of this design choice was examined in [10]. The possibly most severe problem with this approach is that a blocking inliner can introduce deadlocks in the inlined program. Consider, for instance, an API with barrier method $B$ allowing 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 monitor-relevant, and that the inliner, to protect its state, acquires a global lock while performing each monitor-relevant call. For a client program that consists of two threads, each calling $B$ and then terminating, the inliner introduces 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 these reasons we explore the idea of sacrificing security (for the general case) and investigate which monitors are still enforceable through inlining.

An inliner that aims at avoiding the problems described above necessarily has to release the lock protecting the monitor state during the API call. This type of inliner is referred to as a **non-blocking inliner**. A non-blocking inliner can manage to maintain transparency, but can not securely enforce all monitor descriptions as described above. There is however a wide range of interesting monitors that can be correctly enforced even by a non-blocking inliner. These monitors include the so called **race-free** monitors, described in depth in the next section.
2.5 Race-free Monitors

Generalizing from the example in Figure 2.3, 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 2.5.1 (Client-Order Preserving Permutation). A permutation \( \pi(T) \) of a trace \( T \) of observable actions is client-order preserving if, for any \( i \) and \( j \) such that \( i < j \), \( T_i \) is a post-action and \( T_j \) is a pre-action, \( \pi(i) < \pi(j) \).

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. The definition also implies that actions within a single thread can never be permuted: within a thread, pre- and post-actions are strictly interleaved in our program model.

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

Definition 2.5.2. A monitor 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 monitor specification in Figure 2.1 is race-free. As a broader class of examples consider the class of monitors 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 monitors are race-free.

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

Theorem 2.5.3. No inliner can be secure and transparent for a non-race-free monitor.

Proof. (Similar to the proof of Theorem 2.4.1) Let \( \mathcal{M} \) be a non-race-free monitor. By definition there is a trace \( T \) of some program \( \text{Prg} \) which \( \mathcal{M} \) accepts and a client-order preserving permutation \( T' \) of \( T \) which \( \mathcal{M} \) rejects. Now for an inliner, \( I \), to be transparent, \( I(\mathcal{M}, \text{Prg}) \) has to admit the trace \( T \). But, since a client-order preserving permutation respects the happens before relations stipulated by any program, \( I(\mathcal{M}, \text{Prg}) \) must also admit the trace \( T' \), which means that \( I \) is not secure.

A monitor for which there exists a secure, transparent and conservative inliner is said to be inlineable. The corollary below follows immediately.

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

Proof. Let \( \mathcal{M} \) be an arbitrary inlineable monitor. By definition there exists a secure and transparent inliner for \( \mathcal{M} \), thus by Theorem 2.5.3, \( \mathcal{M} \) must be race-free.

2.6 An Inlining Algorithm for Java Bytecode

It is worth emphasizing that the novelty in this and the following sections are not the inlining algorithms themselves: the algorithms are similar to existing algorithms and the locking strategy is relatively straightforward. The contribution, rather, is the proof that the notion of race-free monitors gives an exact characterization of the class of monitors enforceable in concurrent programs by a non-blocking inlining scheme.
2.6.1 Dynamic Method Resolution

In order to tell whether an invoke instruction should be regarded as monitor-relevant or not, the runtime type of the callee has to be determined. This can be solved by, for instance, invoking the `Object.getClass()` method. Unfortunately, however, this method is not available on some of the configurations which we targeted in our preliminary studies [12]. Thus to resolve the runtime type we instead use the `instanceof` bytecode instruction to traverse the class hierarchy (which is known at the time of inlining), bottom up. See [12] for details and pseudo bytecode.

When discussing concurrency-related concerns below, we set this issue aside, and restrict our attention to programs which for all monitor-relevant methods $c.m$, an `invokevirtual d.m` call is bound at run time to method $c.m$ if and only if $d = c$.

2.6.2 Pseudo Code

The inliner, $I_{Ex}$, takes a monitor specification with state definition and event rules of the shapes (2.1) and (2.2) and applies it to a Java bytecode program. The inliner uses static fields $s_i$ of type $T_i$ of an auxiliary class `MonState` to store the shared monitor state, as in the ConSpec state declaration (2.1). (In general a unique name needs to be chosen for the monitor 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 (2.2). 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 local variable 0, arguments from local variables 1,...,$n$ and the monitor state variables from corresponding fields of the `MonState` class. The inliner then replaces each instruction $L: invokevirtual c.m$ of arity $n$ where $c.m$ is monitor-relevant by bytecode implementing the pseudo-code in Figure 2.6.

The inliner locks the monitor state by acquiring the lock associated with the `MonState` class, and stores callee and arguments to the method call for use in event handler code using fresh local variables. The monitor state lock is taken by executing `ldc MonState` and `monitorenter`. The use of a static class for the monitor state makes it easy to determine statically that locks taken or released outside the inlined code snippets do not affect the monitor 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, but 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 2.7 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 inlined handler at `excG1`. After processing of EXCEPTIONAL events the security state is unlocked and the exception rethrown. Exceptions caused by inlined instructions are routed to `exit`.

2.6.3 Correctness

The blocking version of the inliner presented above has been proven secure, conservative and strongly conservative [10]. The non-blocking inliner has been proven to be transparent and conservative for all monitors and secure for race-free monitors [9].

For brevity we here simply state the following two theorems:

**Theorem 2.6.1.** $I_{Ex}$ (blocking version) is secure, conservative and strongly conservative.
Inlined label Instruction | Inlined label Instruction
--- | ---
$L$: lock MonState | ifeq afterElse
store arguments | [eval(F_{m,a})]
goto afterEnd | afterElse: [eval(F_a)]
store callee | goto done
... | excG_{m}: lock MonState
beforeG_{1}: [eval(G_{1,b})] | store exception
ifeq beforeG_{2} | [eval(G_{1,e})]
[eval(F_{1,b})] | ifeq excG_{2,e}
goto beforeEnd | [eval(F_{1,e})]
| goto excEnd
... | excG_{m}: [eval(G_{m,b})]
beforeElse: [eval(F_b)] | ifeq excElse
beforeEnd: restore callee | [eval(F_{m,e})]
restore arguments | goto excEnd
release*: unlock MonState | excElse: [eval(F_e)]
invoke: invokevirtual c.m | excEnd: restore exception
invokeDone*: lock MonState | unlock MonState
store return value | excReleased: athrow
| exit: iconst −1
| invokestatic System.exit
done:

Figure 2.6: The inlining replacement of $L$: invokevirtual $c.m$ for a (transparent) non-blocking inliner. Removing instructions at labels marked with * (release and invokeDone) would yield a (secure) blocking inliner.

**Proof.** For the full proof, we refer to [9].

**Theorem 2.6.2.** $\mathcal{I}_{Ex}$ (non-blocking version) is transparent and conservative for all monitors and secure for all race-free monitors.

**Proof.** For the full proof, we refer to [10].

Finally, we arrive at the following corollary.

**Corollary 2.** The race-free monitors are the maximal set of correctly (securely, transparently and conservatively) inlineable monitors.

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

**2.6.4 Certification**

Theorems 2.6.1 and 2.6.2 provide a guarantee that a program produced by the inliner adheres to the constraints of the inlined monitor. This is a useful guarantee in those situations in which the inlining is performed by the end user himself. To do bytecode transformations after shipment of the software is however...
Deliverable D3.4 Evolvability at Bytecode Level

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke</td>
<td>invokeDone</td>
<td>excG1</td>
<td>any</td>
</tr>
<tr>
<td>L</td>
<td>excReleased</td>
<td>exit</td>
<td>any</td>
</tr>
<tr>
<td>exit</td>
<td>done</td>
<td>exit</td>
<td>any</td>
</tr>
</tbody>
</table>

Figure 2.7: Exception handler array modifications.

not always desirable, especially if the target device has limited resources such as in the case of embedded devices.

In order to convey the guarantee that correct inlining has been performed we have developed a proof-carrying code (PCC) framework. In the preliminary work on sequential Java bytecode [12], this framework is based on low-level per instruction annotations shipped with the code in the form of bytecode annotations. In the multithreaded setting we have chosen a slightly higher level certification mechanism.

Both approaches rely on the monitor specification describing the IRM and an adherence proof being shipped together with the application. Upon reception, the remote device first determines whether the received bundle should be accepted for execution, by comparing the received monitor specification with the on-device user preferences. This test uses a simulation or language containment test, and is explored in detail by K. Naliuka et al. [6]. Then the client proceeds by checking that the program actually adheres to the provided monitor specification, using the attached adherence proof.

It should be noted that the approaches presented are suitable for a wide range of inlining schemes and independent of our particular example implementation. This opens up for optimizing inliners and inliners specifically tailored for certain applications.

Approach for Sequential Java and Blocking Inliners

In the setting of sequential Java the proof generation process can be described in the following steps:

1. During the inlining phase the inliner records the following information
   - Locations of monitor-relevant calls
   - Locations of instructions inserted by the inliner
   - The name of the fields representing the monitor state

2. A so called ghost monitor is extracted from the monitor specification.
   The ghost monitor is essentially an in-memory, static single assignment representation of the monitor specification. The case in which no guard holds is encoded by assigning an error value, ⊥ (with no Java bytecode counterpart) to the monitor state.

3. The proof generator then goes through each instruction of the program, and annotates it with
   - The monitor invariant, \( ms = ms^9 \), stating that the ghost monitor state \( ms^9 \) is in sync with the state of the IRM \( ms \) if the instruction was not added by the inliner. Note that the monitor invariant implies that the IRM is in a valid state (has not been violated), since \( ms \) ranges over ordinary Java values which does not include ⊥.
   - The weakest precondition of the instruction in question, if the instruction was added by the inliner.

This process produces a bytecode annotated with per instruction assertions, serving as a proof outline. Proof verification now boils down to checking that the monitor invariant holds at each security relevant call site. This check is performed in the following steps:
1. The ghost monitor is extracted from the monitor specification.
   This step is identical to step 2 in the proof generation process described above. The reason that the end user redoes it here, is because he needs to trust the ghost monitor to be correct as it will be used as the reference when checking the correctness of the actual IRM.

2. The assertions are loaded into the program, and the checker makes sure that the security relevant calls are annotated with the monitor invariant.

3. The verification conditions are generated according to a well-defined weakest precondition calculus (WPC).

4. The verification conditions are checked one by one.
   This step may seem to be non-trivial. However, since ConSpec specifications have no loops or jumps, the conditions are quite predictable and can easily be handled by a simple term rewriting engine, at least when using our prototype inliner. In case a more sophisticated, optimizing inliner were to be used, the framework would possibly need to be extended to include proofs of the verification conditions written in a predefined proof system.

   See Figure 2.8 for an overview of the framework.

![Figure 2.8: Overview of the PCC framework for sequential Java.](image)

We refer to the technical report [12] found in the appendix for details regarding the WPC and ghost monitor. The technical report also presents two case studies performed on JavaME applications.

In case of multithreaded Java with a blocking inliner the approach described above suffices, provided that a single lock is used for the monitor. Unless it is clear from the context, the proof may also include the particular class used for locking in the reference monitor code.

**Approach for Multithreaded Java**

In case of a non-blocking inliner the situation is slightly more complicated as the validity of the assertions may be affected by interleaving threads. The certification mechanism developed for this case is similar to the approach described above, but makes sure that proper locking and unlocking is performed and takes into account that arbitrary threads may be scheduled between the monitor code inlined for a specific invoke instruction, and the corresponding invoke instruction itself.
This approach is also based on a WPC, but in addition to the labels of the inlined instructions, the
inliner records which invoke instruction the inlined code relates to. This information is sent along with the
program in the form of tuples.

The proof verification process includes the following checks:

- The tuples actually describe proper sections in the code, and that these sections are protected by a
  single lock (referred to as the \textit{monitor lock}).

- There are no jumps into or out from the middle of a section of inlined code.

- If (and only if) a thread enters a section of code corresponding to a \textit{BEFORE} clause, it should execute
  the corresponding API call, and corresponding after clause immediately afterward (unless it terminates
  the application by invoking \texttt{System.exit}).

- Accesses to the IRM state variables are always protected by the monitor lock.

These checks allow the remaining process to use sequential reasoning for the blocks of inlined code,
similar to the approach described for the sequential Java setting.

See [11] (found in Appendix A) for a detailed discussion on these checks.

2.7 An Inlining Algorithm for ABS

The task of inlining a reference monitor into an ABS program differs in some fundamental ways from the
Java bytecode scenario. The concurrency model is quite different and the aspect of distribution (in terms of
COGs) comes into play. The asynchronous method calls for instance, have no counter part in Java bytecode,
and the ABS language does not support shared memory. Due to this, the reasoning about transparency
changes. From a more practical point of view, the inlining scheme takes place on a higher level of abstraction,
and can be realized simply as an ABS abstract syntax tree (AST) transformation step.

From a theoretical point of view, the ABS inlining inherits the same limitations as the non-blocking Java
bytecode inliner due to concurrency, and adds to this, constraints on monitor-internal method calls to avoid
introducing deadlocks. More about this in Section [2.7.7]

2.7.1 \textit{ConSpec}\textsubscript{ABS}

ConSpec was originally designed with Java-like programs in mind [1]. To adapt the language for use in the
context of ABS, we have made a few small, but important changes. The adapted version of ConSpec is from
now on referred to as ConSpec\textsubscript{ABS}.

- The syntax of the expressions in the guards and statements in the update clauses are changed from
  Java to ABS. This means for instance that we have added ABS-style parametric data types.

- The \texttt{EXCEPTIONAL} modifier has been dropped altogether since ABS lacks support for exceptions.

- Due to ambiguity, in ConSpec\textsubscript{ABS} a monitor specification may not refer to inherited methods. That
  is, if the monitor specification mentions method \texttt{I.m}, then \texttt{m} must be declared in interface \texttt{I} and not
  in a super interface of \texttt{I}.

- The types of monitor state variables are changed from the original ConSpec Java types to the predefined
  ABS types: \texttt{Bool}, \texttt{Int} and \texttt{String} similar to [1].

- In addition to the usual arithmetic, relational and boolean operations, the applications of predefined
  global (side-effect free) functions are also allowed.
• As opposed to ConSpec, ConSpec\textsubscript{ABS} does not support any method calls or get expressions as part of guards or update clauses. This is in order to preserve transparency and will be discussed further in Section 2.7.7.

With these restrictions ConSpec\textsubscript{ABS} is more restrictive than ConSpec but is capable of expressing a similar class of monitors as in e.g. \cite{12, 15}.

2.7.2 Inlining Scheme

The ABS inliner is a standalone program which, similarly to the Java bytecode inliner, accepts a monitor specification and an ABS program as input, and produces an ABS (or Java bytecode) program with a non-blocking inlined reference monitor as output.

The ABS inliner starts by generating an interface (\textit{Monitor}) for the monitor and a class which implements it (\textit{MonitorImpl}). The monitor interface allows clients to ask for permission to perform monitor relevant method calls. The monitor implementation will react by either updating the monitor state and return (if the call was question was allowed), or by terminating the entire application in order to prevent a violation. It should be noted that the original design of ABS did not include a statement for terminating the application, so for this task a special \textit{exit} statement has been added to the language. The monitor interface and implementation is generated in the form of separate AST nodes which at a later point are attached to the AST of the final program.

The above step differs from the algorithm presented for Java bytecode. Here the decision point lies within the monitor itself, and not at the client call site. This is due to the different concurrency model and lack of shared memory in ABS.

The inliner proceeds by traversing the program AST to locate all possibly monitor-relevant method call nodes. For each such node, it inserts an asynchronous call node for asking the monitor for permission to proceed with the method call, together with get statement in order to wait for the result.

Finally, a few statements are added in the beginning of the main method which instantiates the monitor. It is created in its own COG in order to be able to respond to requests from all other COGs. The resulting, transformed AST, is then passed to the compiler back-end. The back-end has been equipped with an ABS module to allow for ABS to ABS inlining.

Implementation

The ABS inliner is built as a module in the front-end of the ABS compiler \cite{13}. This approach has many advantages. It is for instance easy to integrate the existing parser and type checker and the user can choose from all target languages supported by the compiler back-end (including the ABS back-end developed specifically for Task 3.4). The architecture is illustrated in Figure 2.9.

Before the inliner kicks in, the ABS source code is parsed and type-checked. The monitor specification is then parsed into a ConSpec\textsubscript{ABS} AST. The type checking of the ConSpec\textsubscript{ABS} description is performed along with the ABS program’s AST. Type checking the monitor specification is indispensable in order to ensure the correctness of the final program. The monitor specification however, entails no information about the entities in the target program, so the ABS AST (providing type information for interfaces, classes, method signatures) is a required input for this process. If the type checking goes through without errors, the inlining algorithm transforms the ABS AST according to the scheme outlined above. The transformed AST is then passed to the compiler back-end for code generation.

2.7.3 Monitoring of Synchronous vs Asynchronous events

As opposed to Java, ABS provides two ways of invoking a method: synchronously and asynchronously. From a monitoring perspective, however, we do not distinguish between the two, and only take the type of the callee, the method signature, and the arguments into account. In the current prototype implementation the monitoring code is, for simplicity, inserted right before and after the method call, regardless of whether it
Figure 2.9: ABS inlining architecture. Components implemented within this task are enclosed by a dashed line.

is synchronous or asynchronous. This scheme results a somewhat counter-intuitive implementation of the ConSpec semantics, since in the case of an asynchronous call, an AFTER clause may (possibly) be processed before the called method has returned.

An alternative scheme, which arguably is more faithful to the original ConSpec semantics but slightly more complex, would be to do the following: For each monitor relevant method a wrapper method is created in the monitor (Figure 2.10). Each asynchronous call to the monitor relevant method is replaced with an asynchronous call to the wrapper method instead (Figure 2.11 and 2.12). The wrapper method performs the
necessary BEFORE updates, calls the wrapped method, waits for the result (using the `await` statement),
performs the necessary AFTER updates and returns the result.

```java
class MonitorImpl {
    Int Imethod(I o, Int v) {
        [BEFORE I.m]
        Fut<Int> res;
        Int val = 0;
        res = o.method(v);
        await res?;
        [AFTER I.m]
        val = res.get;
        return val;
    }
}
```

Figure 2.10: Wrapper method.

```java
I o;
o = new cog SomeClass();
Fut<Int> res;
// A call to monitor relevant
// method I.method(Int)
res = o!method(17);
...
Int val = 0;
val = res.get;
...
```

Figure 2.11: Original call.

```java
I o;
o = new cog SomeClass();
Fut<Int> res;
// The call is replaced by a
// wrapper method.
res = monitor!Imethod(o, 17);
...
Int val = 0;
val = res.get;
...
```

Figure 2.12: Replaced call.

### 2.7.4 Example Inlining

A simple example of the inlining algorithm follows. The program illustrated in Figure 2.13 executes
`Relevant.method()` repeatedly, and the monitor specification in Figure 2.14 states that this method may
not be executed more than three times.

```java
module Count;
interface Relevant {
    Unit method();
}

class TestClass implements Relevant {
    Unit method() {}
}
{
    Relevant testClass;
    testClass = new TestClass();
    while (True) {
        testClass.method();
    }
}
```

Figure 2.13: A sample ABS program.

```java
SCOPE Session
SECURITY STATE
    Int count = 0;
BEFORE Relevant.method() PERFORM
    count < 3 -> {
        count = count + 1;
    }
```

Figure 2.14: A ConSpecABS monitor.

The monitor class generated for the above monitor specification and inlined ABS code is shown in
Figure 2.16 and 2.15 respectively.

The method `beforeRelevantMethod()` in interface `Monitor` is generated according to the event specification
`BEFORE Relevant.method()`, the body of this method is created based on the guarded updates in this event.
module Count;
import * from ABS.StdLib;
import * from ABSSMonitor;
import * from ABSSRTTI;

interface Relevant extends RuntimeType {
    Unit method();
}

class TestClass(Monitor monitor) implements Relevant {
    Unit method();
    String getRuntimeType(){
        return "TestClass";
    }
}

Monitor monitor;
monitor = new cog MonitorImpl();
SecurityRelevant testClass;
testClass = new TestClass(monitor);
while (True) {
    Fut<Unit> returned0;
    returned0 = monitor!beforeRelevantMethod();
    returned0.get;
    testClass.method();
}

Figure 2.15: ABS code resulting from inlining the program shown in Figure 2.13 with the monitor specification in Figure 2.14.

If the behavior is not accepted by the monitor specification, this method will terminate the program by executing the `exit(1)` statement.

2.7.5 Monitor Event Ambiguities

Under certain circumstances the monitor specification may be ambiguous. For example, if a class C implements interface I and I declares a method m, the inlining is ambiguous: if I.m() and C.m() both are declared to be monitor-relevant then more than one update block will be relevant for a call to C.m(). The type checker of ConSpecABS is responsible for discovering this type of ambiguities.

Another, slightly different example is the following. Suppose the target ABS program has two interfaces I_a and I_b and that they both declare a method m. If there is an interface I_c which extends both I_a and I_b (or class C_impl that implements both I_a and I_b), then the monitor description in Figure 2.17 is ambiguous for calls to I_c.m.

There are four possible ways to deal with these ambiguities:

1. Report any ambiguity to the programmer as an error and abort inlining.

   This may seem like a reasonable alternative since the monitor developer may be unaware of the presence
module ABSMonitor;

export *;
import * from ABS.StdLib;

interface Monitor {
    Unit beforeRelevantMethod();
}

class MonitorImpl implements Monitor {
    Int count = 0;
    Unit beforeRelevantMethod() {
        if (count < 3) {
            count = count + 1;
        } else {
            exit(1);
        }
        return Unit;
    }
}

Figure 2.16: The monitor interface and implementation.

Figure 2.17: An example of an ambiguous monitor specification.

of $I_c$ in the above example, and any other option may lead to unforeseen consequences. On the other hand, the developer may know about $I_c$ and still need to express this monitor.

2. Require that both G1 and G2 (see Figure 2.17) have to be satisfied and perform U1 and U2 in some well defined order.

This makes sense because a call to $I_c.m$ may be regarded as a call to either $I_a.m$ or $I_b.m$. In order to be on the safe side we need both options to be allowed. On the other hand, a single event would trigger two updates, which may seem counter-intuitive, and if the updates do not commute, the effect of the combined updates may be unpredictable.

3. Require at least one of G1 and G2 be satisfied. If G1 is satisfied, perform U1, otherwise perform U2.

This option is perhaps the one that makes least sense, since whatever behavior the monitor writer tried to prevent may still go through due to some other, seemingly unrelated event clause.

4. Allow the monitor writer to explicitly specify one of the above options.

Each of the above alternatives has its drawbacks. In our opinion option number one is the least complex and error-prone and it is thus the one we have chosen in our implementation.

2.7.6 Resolving Runtime Type Information

In general, it is impossible for the inliner to decide which type a certain variable will have during runtime. However, this information is important for correct enforcement of the monitor. In the inlining algorithm for Java bytecode this is resolved by letting the IRM query the runtime type of the callee through Java's runtime type information. ABS, however, provides no runtime type information. This problem is resolved by letting each interface declare a method `getRuntimeType()`. For each class, this method returns a string
constant corresponding to the name of the class. By comparing the class name of the event clauses with the string returned by the `getRuntimeType` method, the IRM can decide which (if any) event clause is applicable to a specific event or not.

2.7.7 Correctness

This section discusses the inliner correctness properties defined in Section 2.3.1 namely security, transparency and conservativity.

Security

First we note that the inlining scheme presented in this section is closely related to the non-blocking inliner for Java bytecode described in Section 2.6. Although the monitor state is not protected by a lock, as in the case with the bytecode inliner, the updates to the monitor state are still synchronized due to the fact that the monitor class has exclusive access to the variables representing the state. The monitor is non-blocking since it returns control to the client program before the monitor relevant method is called, and thus does not prevent concurrent monitor relevant calls from being made. Since the ABS inliner is non-blocking, it is inherently insecure for non race-free monitors [9].

To show that the security property holds (for race-free monitors) we pick an arbitrary execution, $E$, of the inlined program and show that the trace of of the execution, $\omega(E)$, is accepted by the inlined monitor.

By first looking at the execution of a single COG in $E$ in isolation, we note the following: Since each possibly monitor relevant method call has been identified by the inliner (using `getRuntimeType` if necessary) and proper calls to the permission checking methods in `Monitor` has been added accordingly, each pre-action is immediately preceded by a return from the corresponding permission checking method. Since each such permission-checking method in `Monitor` is a straight translation of the guarded updates in the ConSpecABS specification, it can be assumed that such method returns the control to the caller, if and only if the corresponding monitor relevant method is safe to invoke. (This could formally be verified by a simple weakest precondition analysis.) Thus for an isolated COG, each method call in $E$ has been checked (and permitted) by the monitor.

The execution of the different COGs can be interleaved arbitrarily in $E$. This means that we can have a situation in which the monitor gives permission for one COG to proceed with a a call to method $m_1$ and then gives permission to another COG to proceed with a call to $m_2$, while the COGs get scheduled so that $m_2$ executes before $m_1$. Since we require that the monitor to be race free however, these kind of situations are guaranteed to be safe and for this particular case for instance, the pre-actions corresponding to $m_1$ and $m_2$ would commute.

The proof outline above mentions only pre-actions, but the arguments for post-actions are analogous.

Transparency

For transparency we need to show that every adherent trace of the program is also a possible trace of the inlined version of the program.

As mentioned in Section 2.7.1, the ConSpecABS has been carefully constrained to prevent unwanted side effects of the monitoring code. The only statements allowed in the update clauses are assignments to monitor state variables, and all expressions must be side effect free and may not include any method calls. This excludes for instance any `await` and `get` expressions which would otherwise open up for the possibility to introduce new scheduling points or deadlocks.

This means that, for any monitor adherent execution of the original program, a similar execution of the inlined version could be constructed by inserting the transitions corresponding to monitor state updates immediately before and after each monitor-relevant call. The inserted code would not interfere with subsequent steps in the execution, since the trace of the original execution adhered to the monitor specification. This implies that the traces of the original program are also traces of the inlined program, and that the inliner is transparent.
Conservativity

For conservativity we need to show the inverse of transparency, namely that every trace of the inlined program is also a trace of the original program.

Due to the syntactical restriction on statements in ConSpec_{ABS}, clauses will not perform any method calls, thus the IRM will not add any observable events to the trace. (Calls to, and returns from the monitor are obviously added, but is excluded from the discussion since these particular calls can never be considered monitor relevant.) Furthermore, since the monitor methods does not return any value, it is guaranteed not to affect the future execution path of the client program. The result follows by an argument similar to the one for the Java bytecode inliner. By removing all configurations corresponding to computational steps performed by the inlined snippets of code, we get a valid execution of the original program, i.e., all the traces of the inlined program are traces of the original program.

2.8 Case Studies

We have implemented inliners that parse monitor specifications written in ConSpec/ConSpec_{ABS} and perform inlining according to the algorithms described in Section 2.6 and 2.7. These inliners have been evaluated in five case studies with varying characteristics. Case study descriptions and results are provided below. For detailed descriptions, case study applications and monitor specifications, we refer to the tool web pages [23, 24].

2.8.1 Trading System

In this case study we monitor the behavior of the concurrent model of a cash desk system. The application stems from an ABS model that was developed in [13]. The monitor 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 always positive. The case study has been performed both by the Java inlining tool and the ABS inlining tool.

The Application

The application is compiled from the Cash Desk ABS model and features bar code scanners, cash boxes, printers, inventories, banks, cashiers and customers. The original ABS model is implemented in 1200 lines, and the resulting Java code resulted in 7300 lines of code.

The Monitor

The monitor in this case counts the number of active sales (+1 for newSaleStarted, and -1 for saleFinished), and asserts that the number of active sales is always greater than or equal to 0. This is expressed in ConSpec as shown in Fig. 2.18.

2.8.2 Swing API-usage

The Java Swing API is designed to be powerful, flexible and easy to use. In particular, it is designed to be easy for programmers to build new Swing components, whether from scratch or by extending components that are provided in the API. For this reason, Swing components are not thread safe, i.e., do not allow access from different threads.

This rule may be hard to remember when writing code, and even if the programmer does remember it, it is sometimes hard to foresee all flows of a program, and whether or not some code will be executed on the event dispatch thread (EDT from now on). What is even worse is that a violation of the rule may result in random deadlocks that are hard to debug.
In this case study we monitor the Swing API usage of a large drawing program, and make sure that the relevant API methods are executed on the EDT.

The Application

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 1.4.103) [26].

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 monitor and by letting the monitor print the stack trace upon a violation we managed to locate and patch the errors.

The Monitor

The monitor has two states: realized and not realized and the monitor states that once realized, a Swing method may only be called if EventQueue.isDispatchThread() returns true—see Figure 2.19.

2.8.3 Session Security

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. This type of threat is part of item 3 (Broken Authentication and Session Management) in the OWASP Top 10 security threats [15].

To eliminate one source of session hijacking attacks one can forbid a session ID from being used from multiple IP addresses.

The Application

In this case study we examine a simple online banking application implemented using the Winstone Servlet Container and the HyperSQL database management system. 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 [2].
Figure 2.19: Monitor for the Swing usage case study.

The Monitor

The monitor forbids session IDs 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 to a known session ID performed from IP addresses not equal to the associated one.

The monitor is implemented using a HashMap for storing the IP to session ID association, and monitors (and restricts) invocations of the HttpServlet.service method. Figure 2.20 shows the ConSpec version.

2.8.4 HTTP Authentication Enforcement

HTTP authentication is a widely used mechanism for protecting administrative web pages. The security it provides is on the level of HTTP-commands, such as GET and POST. This may however not be as fine grained as one may want.

The Application

The application in this case study is the same as the application in the secure sessions case study above. In this case study, however, we focus on the administration interface, which allows administrators to create, search for, and delete account holders (i.e., regular users). The create and delete operations are done through
SCOPE Session

SECURITY STATE HashMap sessionMap = new HashMap();

BEFORE HttpServlet.service(HttpServletRequest req, HttpServletResponse resp) PERFORM
  !sessionMap.containsKey(req.getSession(true).getId()) -> {
  sessionMap.put(req.getSession(true).getId(), req.getRemoteAddr());
  }
  sessionMap.get(req.getSession(false).getId()).equals(req.getRemoteAddr()) -> {
  }

Figure 2.20: Monitor for the Session security case study.

JDBC’s PreparedStatement.executeUpdate and the search is done through PreparedStatement.executeQuery.

The administration pages (URLs on the form /admin/*) are protected using the HTTP authentication mechanism. It currently uses the BASIC authentication method, but it could just as well use, for instance, the form-based authentication method. The relevant snippet of the web.xml file is found in Figure 2.21.

<security-role>
  <role-name>administrator</role-name>
</security-role>
<security-role>
  <role-name>secretary</role-name>
</security-role>

<security-constraint>
  <display-name>Administration Security Constraint</display-name>
  <web-resource-collection>
    <web-resource-name>Protected Area</web-resource-name>
    <url-pattern>/admin/*</url-pattern>
    <http-method>DELETE</http-method>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
    <http-method>PUT</http-method>
  </web-resource-collection>
  <auth-constraint>
    <role-name>administrator</role-name>
    <role-name>secretary</role-name>
  </auth-constraint>
</security-constraint>

<login-config>
  <auth-method>BASIC</auth-method>
  <realm-name>Administration Area</realm-name>
</login-config>

Figure 2.21: web.xml configuration for the HTTP authentication application.
The configuration allows users in the administrator role or secretaries role to access the administration-page. Both types of users are allowed full access, that is, GET, POST, DELETE, PUT.

The Monitor

The monitor in this case states that only users in the role administrator should be allowed to update the database. Other users (the ones in role of secretary) are only allowed to query the database. See Figure 2.22.

```java
SCOPE Session

SECURITY STATE
    boolean initialized = false;
    ThreadLocal adminUrl = new ThreadLocal();
    ThreadLocal adminChecked = new ThreadLocal();

BEFORE HttpServlet.service(HttpServletRequest req, HttpServletResponse resp) PERFORM
    req.getRequestURI().startsWith("/admin") -> {
        initialized = true;
        adminUrl.set(Boolean.TRUE);
        adminChecked.set(Boolean.FALSE);
    }
    ELSE {
        initialized = true;
        adminUrl.set(Boolean.FALSE);
    }

AFTER boolean isInRole = WinstoneRequest.isUserInRole(String role) PERFORM
    role.equals("administrator") && isInRole -> {
        adminChecked.set(Boolean.TRUE);
    }
    ELSE {
    }

// Disallow arbitrary SQL-commands by making sure that no guards ever hold.
BEFORE PreparedStatement.execute() PERFORM
    false -> {
    }

// Allow executeUpdate, only if admin is checked and true.
BEFORE PreparedStatement.executeUpdate() PERFORM
    (!initialized) ||
    adminUrl.get().equals(java/lang/Boolean.FALSE) ||
    adminChecked.get().equals(java/lang/Boolean.TRUE) -> {
    }
```

Figure 2.22: Monitor for the HTTP authentication case study.
2.8.5 URL redirection

In this case study we look at a simple ad-banner that redirects the browser to another URL when the user clicks on it. The inliner enforces a same-origin-policy for redirections.

The Application

The application is in this case an applet embedded in a web page. The applet redirects the browser to a new URL as the user clicks on it.

The Monitor

The monitor states that the URL that the applet sends the browser to, must have the same host as the document base URL of the web page that the applet is embedded in. See Figure 2.23 for the ConSpec specification.

```
SCOPEx Session
SECURITY STATE String origin = "";

AFTER URL docBase = Applet.getDocumentBase() PERFORM
  true -> {
      origin = docBase.getHost();
  }

BEFORE AppletContext.showDocument(URL url, String target) PERFORM
  url.getHost().equals(origin) -> {
  }
```

Figure 2.23: Monitor for the URL redirection case study.

2.8.6 Summary

A summary of the case studies is given in Table 2.1. Benchmarks were performed on a computer with a 1.8 GHz dual core CPU (64 bit) and 2 GB memory, running Ubuntu 9.10 and Java Hotspot VM 1.5.0. The runtime overhead due to inlining was measured for the web application case studies (CS3 and CS4) and for the Swing case study (CS 2). 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 start up time (the time required to construct the user interface).

As can be seen in the table, the overhead is negligible in all cases except for CS2, which concerns testing and is not intended for production anyway. The inliner is by no means optimized for speed, but still performs the task in reasonable time.
<table>
<thead>
<tr>
<th>Case Study</th>
<th>ConSpec clauses</th>
<th>Size before inlining (kB)</th>
<th>Size after inlining (kB)</th>
<th>Size increase (%)</th>
<th>Monitor-relevant calls</th>
<th>Inlining time (s)</th>
<th>Runtime Overhead (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS1 (Trading)</td>
<td>2</td>
<td>652.9</td>
<td>654.0</td>
<td>0.17</td>
<td>2</td>
<td>2.52</td>
<td>n/a*</td>
</tr>
<tr>
<td>CS2 (Swing)</td>
<td>249</td>
<td>1888.6</td>
<td>2140.7</td>
<td>13.35</td>
<td>1038</td>
<td>26.68</td>
<td>11.27</td>
</tr>
<tr>
<td>CS3 (Sessions)</td>
<td>1</td>
<td>532.7</td>
<td>533.1</td>
<td>0.08</td>
<td>1</td>
<td>2.47</td>
<td>0.44</td>
</tr>
<tr>
<td>CS4 (HTTP Auth.)</td>
<td>4</td>
<td>532.7</td>
<td>535.6</td>
<td>0.54</td>
<td>12</td>
<td>2.66</td>
<td>0.87</td>
</tr>
<tr>
<td>CS5 (Redirection)</td>
<td>2</td>
<td>27.5</td>
<td>28.2</td>
<td>2.41</td>
<td>1</td>
<td>0.18</td>
<td>n/a*</td>
</tr>
</tbody>
</table>

*) runtime overhead too insignificant to measure.

Table 2.1: Quantitative results of the case studies.
Chapter 3

Compiling ABS into Java Bytecode

Most of the techniques for analysis and verification of behavioral properties developed within HATS focus on the ABS model level. One remaining question is how to preserve properties that have been established on the modeling level when compiling/refining ABS to executable code. In the context of proof-carrying code, solutions such as proof-obligation preserving compilation [5] or type systems have been investigated that ensure that certain optimizations are property preserving [28].

In this chapter we suggest an alternative solution, namely, a rule-based compiler which reuses the ABS dynamic logic (ABS DL) developed in Task 2.5 “Verification of General Behavioral Properties”, see also [14]. ABS DL utilizes the symbolic execution paradigm implementing implicitly a symbolic interpreter. In a first phase, the ABS model is executed symbolically computing the weakest precondition of the property to be proven. During that phase the program is stepwise analyzed and the verification problem reduced to a first-order plus theories theorem proving problem. The reduction is possible even in presence of loops and recursive methods as the calculus utilizes a loop invariant rule and method contracts, respectively, to ensure a finite proof tree.

The idea for the work presented here is to extend the calculus rules for symbolic execution (program analysis) with a component constructing an equivalent, potentially optimized program in another target language. From here on, the target language is the Java bytecode language. On the ABS side we focus on sequential Core ABS, even though the approach principally scales up to Full ABS. The latter is future work and will be addressed in Task 1.4 “System Derivation and Code Generation”.

3.1 Background

Dynamic Logic [20] belongs to the family of modal logics. ABS dynamic logic (ABS DL) is a sorted first-order logic (plus arithmetic) extended with modalities. In this work we use only the box modality $[\cdot]$ corresponding to partial correctness.

In ABS DL programs are first class citizens of the logic, i.e., similar to Hoare logics they occur directly and not in an encoded form as part of the formulas. Let $p$ denote an executable sequence of ABS statements and $\phi$ any formula in ABS DL then $[p]\phi$ is an ABS DL formula itself. The intuitive meaning of the formula is that if $p$ is executed and terminates then the formula $\phi$ holds in all possible final states. For deterministic sequential languages this coincides with the notion of partial correctness as there is at most one final state. Please note that sequential (Core) ABS does not provide any explicit non-deterministic statements like a non-deterministic assignment and other forms of indeterminism is taken care of by underspecification. The distinction between both is not of importance between the work presented here. Further, we work under the assumption that the ABS model to be compiled has been sufficiently refined.

Example 3. An example of an ABS DL formula is

$$i = i_0 \land j = j_0 \rightarrow [i\leftarrow i+j; j\leftarrow i-j; i\leftarrow j-i;](i = j_0 \land j = i_0)$$

The formula expresses that the enclosed program swaps the content of the program variables $i$ and $j$. 
ABS DL denotes a collection of logics where each logic is defined with respect to a context program, i.e., for each ABS program an instance of the logic is created defining the required sorts etc. Further, the context program defines which interfaces, classes and methods are defined. Formally, an ABS DL formula is evaluated with respect to an ABS DL Kripke structure. A Kripke structure $K = (D,S,\rho)$ consists of a first-order structure $D$ defining the domain $D$ and the interpretation $I$ of state independent function and predicate symbols. A set of states $S$, and a state transition relation $\rho$ define the program semantics.

A formula $\phi$ is satisfiable if there is an ABS DL Kripke structure $K$, an initial state $s_0$ and a variable assignment $\beta$ such that $\phi$ is evaluated to true ($K,s_0,\beta \models \phi$). A formula is valid if it is true for all ABS DL Kripke structures, states and variable assignments.

The final syntactic construct of ABS DL to be introduced here are updates [27]. Updates capture state changes and can be seen as explicit generalized substitutions. An elementary update

$$l := r$$

is a pair of a location $l$ and a term $r$. Its meaning is the same as an assignment. Elementary updates can be combined to parallel updates

$$u_1 \parallel u_2$$

There might be a conflict between $u_1$ and $u_2$, i.e., the same location is assigned different values. Conflict resolution follows the last-win semantics, i.e., the later assignment overrides the former. Updates are applied to formulas or terms, i.e., let $u$ denote an update and $\xi$ a term or formula in ABS DL then $\{u\}\xi$ is a term (or formula) in ABS DL.

**Example 4.** A few examples of formulas using updates:

- The formula $\{i := 0\}\phi$ evaluated in a state $s_0$ means that formula $\phi$ is evaluated in a state coinciding with $s_0$ except for program variable $i$ being 0.

- The formula $\{i := j \parallel j := i\}\phi$ evaluated in a state $s_0$ means that formula $\phi$ is evaluated in a state coinciding with $s_0$ except that the values of the program variables $i, j$ are swapped.

- The update in the formula $\{i := 0 \parallel i := 1\}\phi$ is an example for an update with a conflict as the program variable $i$ is assigned two different values. As such conflicts are solved using a last-win semantics, the above formula is equivalent to $\{i := 1\}\phi$.

### 3.1.1 Sequent Calculus

We use a sequent calculus to check an ABS DL formula for validity. In this section we define the basic notions tailored to the needs of the presented work. A sequent $\Gamma \Rightarrow \Delta$ relates two sets of formulas $\Gamma$ and $\Delta$. Its meaning is equivalent to

$$\bigwedge_{\phi \in \Gamma} \phi \rightarrow \bigvee_{\psi \in \Delta} \psi$$

The formulas on the left side of the sequent arrow are called antecedent of a sequent, the formulas on the right side are called succedent of a sequent.

A sequent proof is a tree of which each node is labelled with a sequent and there is a (sequent) rule

$$\text{rule } \frac{seq_1 \mid \ldots \mid seq_n}{seq}$$

for each node such that the parent matches the rule’s conclusion and the children its premises. Typical first-order sequent rules look like

**orRight** $\Gamma \Rightarrow A, B, \Delta$

$\frac{\Gamma \Rightarrow A \lor B, \Delta}{\Gamma \Rightarrow A, B, \Delta}$

**andRight** $\Gamma \Rightarrow A, \Delta$

$\frac{\Gamma \Rightarrow B, \Delta}{\Gamma \Rightarrow A \land B, \Delta}$
Deliverable D3.4 Evolvability at Bytecode Level

and are applied from bottom to top. Applying a sequent rule means matching its conclusion with the sequent of one of the leaves of a proof tree and then adding the instantiated premises as new children. The rule \texttt{andRight} matches on any formula with a conjunction as top level operator that occurs in its succedent. For example, \texttt{andRight} can be applied on the sequent

\[
\Gamma \Rightarrow \psi \land (\phi \rightarrow \xi)
\]

where \(A\) is instantiated to \(\psi\) and \(B\) is instantiated to \((\phi \rightarrow \xi)\). Applying the rule

\[
\begin{align*}
\Gamma \Rightarrow \psi & \Rightarrow \phi \rightarrow \xi \\
\Gamma \Rightarrow \psi \land (\phi \rightarrow \xi)
\end{align*}
\]

creates two new leaves each of them labelled with a sequent corresponding to the instantiation of one of the rule’s premises. The intuition behind the rule is that if we want to prove that the conjunction of \(A\) and \(B\) holds, we have to prove that \(A\) and \(B\) hold independently.

In addition to rules dealing with classical first-order formulae, the sequent calculus has to provide rules for ABS programs too. The rule \texttt{assignLocVar2LocVar}

\[
\texttt{assignLocVar2LocVar} \quad \Gamma \Rightarrow \{U\{l := r\}\{rest\} \varphi\}, \Delta
\]

rewrites the assignment statement between two local variables into an update where \(U\) stands for an update that may have been accumulated in previous rule applications. Other assignment rules treat more complex cases like assigning the sum of two expressions to a program variable or field. In these cases, the addition has to be rewritten into its logic counterpart. The rule for conditional statements

\[
\texttt{conditional} \quad \Gamma, \{U\} \Rightarrow \{p\} \text{ rest} | \phi, \Delta \quad \Gamma, \{U\} \Rightarrow \{q\} \text{ rest} | \phi, \Delta
\]

splits the proof into two branches: The left one considering the case where the condition is satisfied and the \texttt{then} branch executed. While the right branch treats the complementary case where the symbolic execution continues with the \texttt{else} branch assuming the condition does not hold.

One can observe that these rules work on the first active statement of a program and transform it into a sequence of simpler statements or translate it into a first-order logic construct. Our goal to translate a high-level programming language such as ABS into bytecode fits well into that workflow. Crucially, the rules decompose complex expressions into simpler ones. Figure 3.1 shows symbolic execution of a small program, executing first an assignment and then splitting symbolic execution into two branches for the conditional assignment.

Figure 3.1: Symbolic program execution in a sequent calculus.

Symbolic execution follows the control-flow of the program. In case of alternative control-flows, symbolic execution causes a split in as many branches as necessary to cover all cases (here: into two branches covering the conditional statement’s \texttt{then} and \texttt{else} branch). Two observations:

1. The remaining program \(r\) is executed twice and no subsequent merge of the control-flows occurs.

2. The realization of symbolic execution as part of the program logic calculus allows alternation between symbolic execution steps and classical first-order reasoning. This way, if we can deduce that one of the branches is infeasible (e.g., program variable \(b\) is known to be \texttt{false}), then the corresponding branch can be closed immediately and the remaining program needs not to be symbolically executed.
3.2 Extending the Calculus

As stated before, our aim is to realize a bytecode compiler on top of an existing program logic based on symbolic execution. The previous section described briefly how a complex program is transformed successively into a simpler program during symbolic execution. Complex expressions and statements are rewritten into sequences of statements of lesser complexity until an elementary statement is reached. The effect of such an elementary statement is then directly translated into first-order logic possibly causing proof splits.

This process is essentially similar to that of a compiler when decomposing higher level expressions into elementary machine code instructions (usually via an intermediate representation). Our approach adds annotations to formulas (sequents) capturing the compiled program and additional analysis information.

Elementary source code statements can usually be expressed using a few machine code instructions. Consequently, the actual compilation steps are performed by those calculus rules that translate elementary statements into first-order logic expressions. For example, the assignment rule for local variables from the previous section becomes the following extended rule:

\[
\text{asgnLocVar2LocVar} \quad \Gamma \Rightarrow \{U\} \{l := r\} \phi \bowtie (X)(\overline{\text{rest}}, \text{use}), \Delta
\]

\[
\Gamma \Rightarrow \{U\}[l = r; \text{rest}] \phi \bowtie (X) \left( \begin{array}{c}
\text{load } r; \text{store } l; \overline{\text{rest}}, (\text{use} - \{l\} \cup \{r\}) \\
\text{if } l \in \text{use} \\
\text{otherwise}
\end{array} \right), \Delta
\]

We describe this rule now in more detail: In contrast to the bottom-up application direction of the rule when performing symbolic execution (analyzing the formula/program), the program synthesis is performed top-down. We consider first the rule premise: the annotation \(\text{rest}\) denotes the Java bytecode compilation of the remaining program \(\text{rest}\), while the annotations \(X\) and \(\text{use}\) are used to encode use-definition chains of program variables and are explained later in detail. The rule’s conclusion needs to add the bytecode for the assignment such that the compiled program corresponds again to the program of the formula. This is actually done in the first case if the program variable on the left side occurs in the \(\text{use}\) set.

The \(\text{use}\) set contains all program variables on which a read access might occur in the remaining program before being overwritten. When the first case, when the left side \(l\) of the assignment is among those variables, we have to update the use set by removing the newly assigned program variable \(l\) and adding the variable \(r\) which is read by the assignment. The second case makes use of the knowledge that the value of \(l\) is not accessed in the remaining program and skips the compilation of the assignment.

The \(X\) annotation keeps track of used variables that do not occur syntactically in the program, but may be accessed in its continuation. Its use will become clear when we explain the loop invariant rule. We formally define the notion of an extended program formula:

**Definition 3.2.1 (Extended Formula).** An extended program formula is a formula of the following form:

\[
\{U\}[p]\phi \bowtie (X)(bc, use)
\]

- \(bc\): Java bytecode compilation of the ABS program \(p\)
- \(use\): program variables used in \(p\) (or continuation of \(p\)); collected in backward analysis
- \(X\): program variables potentially used in continuation of \(p\); maintained in forward analysis

The sequent calculus rules are then generally extended to rules working on extended program formulas:

\[
\text{ruleName} \quad \Gamma_1 \Rightarrow \{U_1\}[p_1]\phi \bowtie (X_1)(bc_1, use_1), \Delta_1 \ldots \Gamma_n \Rightarrow \{U_n\}[p_n]\phi \bowtie (X_n)(bc_n, use_n), \Delta_n \quad \Gamma \Rightarrow \{U\}[p]\phi \bowtie (X)(bc, use)
\]

The notion of correctness for this kind of rules consists of two parts: the first part being the classical soundness condition “if the premises of the rule are valid so is its conclusion”. The second one has to ensure
that the compilation is sound, which is done by establishing a bisimulation property between the source and the target program.

The rules can be read in a purely declarative manner, but to obtain an algorithmic approach to the construction of proof trees over extended we proceed in two phases. In the first phase the program logic calculus is applied as normal and the program is symbolically executed. In our current approach, the only additional information already gathered during the first phase is the used variable set \( X \). The variable set \( X \) contains all program variables which might be read later on, but do not occur in the analyzed ABS program itself. A typical example is the program variable capturing the result of a method. Assignments to this variable should not be discarded even though a read access will not be encountered. Other examples are given later.

In the second phase, when the ABS program has been fully symbolically executed, the program synthesis is started with the applications of the emptyBox rule:

\[
\text{emptyBox} \quad \frac{\Gamma \Rightarrow \{U\} \phi \otimes (X)(\_\_, \_\_)}{\Gamma \Rightarrow \{U\} [\ ] \phi \otimes (X)(\text{nop}, X), \Delta}
\]

where \( \_ \) are anonymous placeholders. The interesting aspect of this rule is that the variable set \( \text{use} \) tracking read access to variables is instantiated with the set \( X \), thus “protecting” assignments to those variables from being deleted. The bytecode translation component inserts only a \( \text{nop} \) instruction representing the empty program.

In the following we describe two further rules in more detail starting with the rule for conditional statements:

\[
\text{conditional} \quad \frac{\Gamma, \{U\} b \Rightarrow \{U\} [p; r] \phi \otimes (X)(p; p, \text{use}_{p; r}) \quad \Gamma, \{U\} \neg b \Rightarrow \{U\} [q; r] \phi \otimes (X)(q; q, \text{use}_{q; r})}{\Gamma \Rightarrow \{U\} [\text{if} (b) \{p\} \text{else} \{q\}; r] \phi \otimes (X)(\text{iload} b, \text{ifeq} \text{else} \text{lelse} \text{lthen}: p; p, \text{use}_{p; r} \cup \text{use}_{q; r} \cup \text{locs}(b)), \text{lelse}: q; q, \text{use}_{q; r})}
\]

On encounter of a conditional statement, symbolic execution splits into two branches, namely the \text{then}-branch and \text{else}-branch. The bytecode translation of the conditional statement reflects this behavior as follows: first the guard variable is loaded and checked. If its value is false then program execution continues by setting the program counter to the position labeled with \text{lelse}, otherwise the program counter is simply incremented and execution continues from there on, i.e., the position labeled with \text{lthen}. The bytecode executed at \text{lthen} and \text{lelse} consists of their respective bytecode translation.

As mentioned earlier, the statements following the conditional statement are symbolically executed on both branches. This leads to duplicated code in the compiled program, and, potentially to code size duplication at each occurrence of a conditional statement. One note in advance: code duplication can be avoided when applying a similar technique as presented later in connection with the loop translation rule. However, it is noteworthy that the application of this rule might have also advantages: as discussed in [8], symbolic execution and partial evaluation can be interleaved resulting in (considerably) smaller proofs. Interleaving symbolic execution and partial evaluation is orthogonal to the approach presented here and can be combined easily. In several cases this can lead to different and drastically specialized and therefore smaller versions of the remainder program \( r \) and its compilation \( r \). The used variable set is extended canonically by joining the use sets of the different branches and the guard variable.

The final rule to be discussed in detail is the loop invariant rule \text{loop}:
On the logical side the loop invariant rule is as expected and has three premisses: the top-most, initially valid, ensures that the provided invariant is valid at the start of the loop, the second, preserves invariant, ensures that \( \text{Inv} \) is actually an invariant of the loop body, and the use case continues with program execution after the loop.

Here we are interested in compilation of the analyzed program rather than proving its correctness. Therefore, it is sufficient to use \( \text{true} \) as a trivial invariant or to use any automatically obtainable invariant. In this case the first premise ensuring that the loop invariant is initially valid contributes nothing to the program compilation process and is ignored from here onwards (if \( \text{true} \) is used as invariant then it hold trivially).

Two things are of importance: the third premise executes only the program following the loop. Furthermore, this code fragment is not executed by any of the other branches and, hence, we avoid unnecessary code duplication. The second observation is that variables read by the program in the third premise may be assigned in the loop body, but not read in the loop body. Obviously, we have to prevent that the assignment rule discards those assignments when compiling the loop body. Therefore, we must add to the variable set \( X \) of the second premise the used variables of the third premise and, for similar reasons, the program variable(s) read by the loop guard. In practice this is achieved by first executing the use case premise of the loop invariant rule and then using the resulting \( \text{use}_3 \) set in the second premise. The work flow of the synthesizing loop is shown in Figure 3.2.

![Figure 3.2: Work flow of synthesizing loop.](image-url)
3.3 Example

We conclude this chapter by demonstrating the suggested approach on a small example. The ABS method to be compiled is shown in Figure 3.3.

```
Int tot = 0;
Int atot = 0;
Int i;
Bool cpn;
while (i > 0) {
    tot = tot + 20;
    atot = tot;
    i = i - 1;
}
if (cpn) {
    tot = tot - 50;
    if (tot < 0) {
        tot = 0;
    }
}
return tot;
```

Figure 3.3: ABS method to be compiled into bytecode.

This ABS program could possibly be used in an online store. It calculates the total amount the customer has to pay if buying \( i \) items at an item price of 20 SEK. The total sum is stored in both \( tot \) and \( atot \). If the customer can provide a coupon a reduction of 50 SEK will be applied. Finally, the total cost is returned as \( tot \).

We begin with symbolic execution of our example program. The first statements are simple variable declarations and initializations that are treated similar to assignments. The first steps until reaching the loop are shown in Figure 3.4.

```
Γ ⇒ \{ atot := 0 \} \land \{ tot := 0 \} \land \{ atot := 0 \} \land \{ tot := 0 \} \land \{ \text{while}(i>0) \} \land \{ \text{if}(cpn) \} \land \{ \text{if}(tot < 0) \} \land \{ \text{return} \}
```

Figure 3.4: Symbolic execution steps until loop is reached.

Notice, that the variable set \( X \) which is determined in the first (forward) phase is initially instantiated with \( tot \), because this variable is implicitly read by the return at method exit.

Applying the loop invariant rule creates two new goals (we ignore the first premise initially valid). The rule application and the resulting goals are shown in Figure 3.5. As the variable set in the premise preserves invariant \( X \) depends on the instantiation of the used variable set \( use_3 \) of the use case, we continue with the latter.

During symbolic execution of the use case branch two conditional statements have to be executed until, finally, reaching the end of the method. The resulting proof tree is shown in Figure 3.6.

After symbolic execution, the Java bytecode needs to be synthesized. This means that starting with the application of the emptyBox rule, the schema variables \( bc_{11} \) and \( use_{11} \) shown in Figure 3.6 are instantiated as \( \text{nop} \) and \( tot \), respectively.

Going backwards we can now derive the instantiations for \( bc_9 = \text{iconst}_0 \); \( \text{istore} \ tot \) and \( use_9 = tot \) according to the assignment rule. The previous rule application was executing a conditional statement.
Before we can continue, we have first to derive the instantiations for the other premise. By similar steps as acquiring locks in the Java bytecode translation, we can continue with the compilation of the conditional statement. As result we derive for the remaining program following the loop and the set of variables used in it.

Assume now that we can derive that \( \text{cpn} \) is false, i.e., that the customer does not possess a coupon. Infeasible path detection (and partial evaluation) allow then the translation of both conditional statements as the required instantiation for \( X_3 \), namely, \{\text{tot}, \text{cpn}, \text{i}\} is now known. Figure 3.8 shows the proof tree after symbolic execution of the loop body.

Synthesis of the compiled program follows the same pattern as described for the use case branch. Please, note that as \( \text{atot} \) is not in the used variable set and also not read within the loop, its assignment in the loop body is discarded. The Java bytecode generated under the assumption that \( \text{cpn} = \text{FALSE} \) by our approach is in Figure 3.9.

### 3.4 Conclusion

The presented compilation approach is currently only applicable to sequential ABS. Extending the approach to include concurrent constructs naïvely should be straightforward. Basically, the translation needs to map asynchronous method calls, await statements, etc., to corresponding Java library methods. Correct blocking of threads and prevention of reentrance in the asynchronous case have to be ensured by introducing and acquiring locks in the Java bytecode translation.
HATS Deliverable D3.4 Evolvability at Bytecode Level

\[ \Gamma, \mathcal{UV}_a(\text{Inv} \land i > 0) \Rightarrow \mathcal{UV}_a[\ldots || i := i-1][ |\text{Inv} @ (\{\text{tot}, \text{cpn}, i\})(bc_{14}, use_{14}) \]

\[ \Gamma, \mathcal{UV}_a(\text{Inv} \land i > 0) \Rightarrow \mathcal{UV}_a[\ldots || \text{atot} := \text{tot}][ i = i - 1; \ldots ]\text{Inv} @ (\{\text{tot}, \text{cpn}, i\})(bc_{13}, use_{13}) \]

\[ \Gamma, \mathcal{UV}_a(\text{Inv} \land i > 0) \Rightarrow \mathcal{UV}_a[\text{tot} := \text{tot}+20][ \text{atot} = \text{tot}; \ldots ]\text{Inv} @ (\{\text{tot}, \text{cpn}, i\})(bc_{12}, use_{12}) \]

\[ \Gamma, \mathcal{UV}_a(\text{Inv} \land i > 0) \Rightarrow \mathcal{UV}_a[\text{tot} = \text{tot} + 20; \ldots ]\text{Inv} @ (\{\text{tot}, \text{cpn}, i\})(bc_{4}, use_{4}) \]

Figure 3.8: Preserves invariant.

```java
iconst_0
istore_tot

exit:
  iload_tot

guard:
  iload_i
  ifle exit
  iload_tot
  bipush 20
  iadd
  istore_tot
  iinc i, -1
  goto guard
```

Figure 3.9: Generated optimized Java bytecode from ABS.

Our approach is closely related to rule-based compilation [3, 7]. Our approach differs in the sense that to the best of our knowledge their inference machine is by far not as powerful as here. Also closely related are recent approaches to translation validation of optimizing compilers (e.g., [4]) which also use a theorem prover to discharge proof obligations. These work usually on an abstraction of the target program. Both mentioned approaches encode the compilation strategy within the rules, while our approach separates the actual strategy from the translation rules. Our work is also related to the Verifying Compiler [21] project which aims at the development of a compiler that verifies the program during compilation.

The close connection between the program logic and the compilation allows to ensure the correctness of the compilation process as such. We see a great potential of our approach when encoding security or safety properties in terms of postconditions. This should allow to identify unsafe or unsecured execution paths during compilation and either to abort compilation or to wrap the undesired execution paths in a wrapper that at least ensures the safety or security property of interest. For example, execution paths that may leak information can be secured by omitting the assignments which violate secure information flow. Another possibility would be to ensure that if the program enters an unsecured execution path, then the program will not terminate. Exploring these avenues is future work.
Chapter 4

Conclusion

In this deliverable we studied the use of bytecode level program transformations for runtime code evolution, mainly focusing on security policy inlining as a technique to allow a running system to adapt to new security requirements as they become available. We have obtained both foundational new results in analyzing the kinds of correctness properties that are attainable in a multithreaded setting, we have exhibited several provably correct, and in some cases, namely, certifying security monitor inliners, we have shown that realistic security concerns can be addressed. Furthermore, the development has addressed both Java bytecode and ABS.

For future development several aspects merit further work:

- In technical terms, our approach needs to be extended in several directions, to handle, for example, inheritance (which is essentially straightforward) and callbacks (which is possible, but requires deeper modifications as new event types must be accounted for).

- Monitor inlining can be viewed as a special case of aspect weaving. Just as for monitor inlining, it is of interest to develop a theory of aspect weaving correctness, generalizing the approach developed in the present text. We conjecture that an approach based on action transducers rather than the action acceptors studied here would be the right approach, but for resource reasons we have had to leave this idea to future work.

- More work is needed, also, to further develop the monitoring framework. For instance, we are not yet able to track data flow as finely as we would like to, and information flow properties are outside the scope of our work. It would be of interest to alleviate these shortcomings.

- Our work has focused on security monitor inlining as examples of behaviour transforming bytecode transformations. One may want to transform bytecode for many other reasons, for instance for performance, by dynamically optimizing the bytecode in response to various forms of virtual machine instrumentations, as happens in commercial JIT compilers. Performance driven bytecode level optimizations will be studied in Task 3.5 “Autonomously Evolving Systems” on runtime adaptation.

Property preserving compilation of ABS to Java bytecode can be achieved by closely coupling program logic and compilation. The correctness of the compilation can then be proven as part of the correctness of the program logic calculus using bisimulation. The approach presented in this deliverable allows already certain optimisations and we expect that further program optimisation can be built-in. The seamless integration of symbolic execution, compilation and general theorem proving allows infeasible paths to be detected as contradictions on the program logic level. The presented approach ensures that no bytecode is compiled for detected infeasible paths. Variability points can be resolved on-the-fly if specified as preconditions such that only the relevant code fragments are compiled. The approach is modular (no global optimisations) such that local changes due to evolution require only re-compilation of the effected model elements, but not of unrelated parts.
Bibliography


Glossary

Terms and Abbreviations

ABS Abstract Behavioral Specification language. An executable class-based, concurrent, object-oriented modeling language based on Creol, created for the HATS project.

AOP Aspect Oriented Programming.

API Application programming interface, provided by a component to enable communication with other components.

AST Abstract Syntax Tree.

Bytecode A binary representation of an executable program designed to be executed by a virtual machine rather than by dedicated hardware.

COG Concurrent Object Group.

ConSpec ConSpec is a formal monitor specification language designed for the Java language.

DL Dynamic Logic. A modal logic where programs are first-class citizens similar to Hoare logics.

EDT Event Dispatch Thread. A thread dedicated for user interaction in the Java Swing/AWT API.

IRM Inlined Reference Monitor.

JVM Java Virtual Machine.

Monitor A monitor is a mechanism for monitoring application events of interest. It may also serve as a security enforcement mechanism by preventing certain sequences of events.

Monitor Inlining The process of weaving a reference monitor into the program to be monitored.

PCC Proof Carrying Code.

Program Logic A logic used to reason about program properties.

RTTI Runtime Type Information.

Sequent A sequent \( \Gamma \Rightarrow \Delta \) is a data-structure consisting of two distinguished sets \( \Gamma, \Delta \) of formulas.

Sequent Calculus Calculus based on sequents used to reason about validity of formulas in a syntactic manner.

Symbolic Execution Program execution where program locations have symbolic values instead of concrete values.

Use-Definition Chain Link between the definition of a program variable and its use.

WPC Weakest Precondition Calculus.
Appendix A

Security Monitoring and Certification for Multithreaded Java
Security Monitor Inlining and Certification for Multithreaded Java

MADS DAM\textsuperscript{1}, BART JACOBS\textsuperscript{2}, ANDREAS LUNDBLAD\textsuperscript{1} and FRANK PIESENS\textsuperscript{2}
\textsuperscript{1} Royal Institute of Technology (KTH), Sweden, \textsuperscript{2} Katholieke Universiteit Leuven, Belgium

Received 20 December 2010

Security monitoring 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 inlinable 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.

For sequential programs, execution monitoring and inlining enforce the same policies (Hamlen et al., 2006b).† Our first contribution is to show that, somewhat surprisingly, for multithreaded programs this is no longer true. 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 multi-threaded 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 details of the API implementation, 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, and it generally forces inlining and execution to take place under the same jurisdiction. To this end we turn to certification. Certification for IRM’s have been considered before, in Hamlen et al’s Mobile system (Hamlen et al., 2006a) for .NET using types, by Aktug et al (Aktug et al., 2009) for JVM and the ConSpec policy language (Aktug and Naliuka, 2008) using pre-post conditions, and by Sridhar and Hamlen (Sridhar and Hamlen, 2010c) for

† 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).
As our final contribution we show how security certificates could be generated for race-free ConSpec policies inlined into multithreaded Java bytecode using the IRM presented earlier. The main challenge in relation to our earlier work (Aktug et al., 2009) is that synchronization is needed to ensure mutually exclusive access to a shared security state. Certificates are presented as bytecode augmented with a reference (“ghost”) monitor. This allows a 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 S³MS project (S³MS, 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; Vanoverbergh 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.

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 (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 application 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 liter-
ature. 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 exe-
cutions 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 presen-
tation 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.

<table>
<thead>
<tr>
<th>Java Bytecode Programs</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P : c \rightarrow \text{Class} )</td>
</tr>
<tr>
<td>( c \in \text{String} )</td>
</tr>
<tr>
<td>( \text{Class} ::= (m \rightarrow M, f^*) )</td>
</tr>
<tr>
<td>( m \in \text{String} )</td>
</tr>
<tr>
<td>( M ::= (\iota^+, H^*) )</td>
</tr>
<tr>
<td>( \iota \in \text{Insn} )</td>
</tr>
<tr>
<td>( f \in \text{String} )</td>
</tr>
<tr>
<td>( H ::= (\ell_b, \ell_e, \ell_t, c) )</td>
</tr>
<tr>
<td>( \ell \in \mathbb{N} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JVM Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C ::= (h, \Lambda, \Theta) )</td>
</tr>
<tr>
<td>( h : ((o \times f) \cup (c \times f)) \rightarrow \text{Val} )</td>
</tr>
<tr>
<td>( o \in \mathbb{N} \cup {null} )</td>
</tr>
<tr>
<td>( \text{Val} ::= o \mid v )</td>
</tr>
<tr>
<td>( v \in \text{byte} \cup \text{short} \cup \text{int} \cup \text{long} \cup \text{float} \cup \text{double} \cup \text{boolean} \cup \text{char} )</td>
</tr>
<tr>
<td>( \Lambda : o \rightarrow \text{tid} )</td>
</tr>
<tr>
<td>( \text{tid} \in \mathbb{N} )</td>
</tr>
<tr>
<td>( \Theta : \text{tid} \rightarrow \theta )</td>
</tr>
<tr>
<td>( \theta \in \mathbb{R}^\star )</td>
</tr>
<tr>
<td>( R ::= (c.m, pc, s, l) \mid (a) )</td>
</tr>
<tr>
<td>( pc \in \mathbb{N} )</td>
</tr>
<tr>
<td>( s \in \text{Val}^\ast )</td>
</tr>
<tr>
<td>( l : \mathbb{N} \rightarrow \text{Val} )</td>
</tr>
</tbody>
</table>

Table 1. JVM Programs and configurations.

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, \( \alpha \) 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 \(\text{Throwable}\)) \(t\).
- \(\tau\) represents an internal computation step.

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

Executions, Traces A (finite) execution of a program \(Prg\) is a sequence \(E = C_0 \xrightarrow{\alpha_0} \cdots \xrightarrow{\alpha_{n-1}} C_n\) 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 \(Prg\), \(pc = 1\), such that for each \(i : 0 \leq i < n\), \(C_i \xrightarrow{\alpha_i} C_{i+1}\), and such that \(E\) is compatible with the happens-before relation as defined by JLS3 (Gosling et al., 2005). Refer to the appendix for a brief summary of those parts of the JLS3 memory model that are relevant for this paper. 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 \(\omega(E)\) of \(E\) is the sequence \(\alpha_0\alpha_1\cdots\alpha_{n-1}\) with \(\tau\) actions removed, and \(T(Prg) = \{\omega(E) \mid E \text{ is an execution of } Prg\}\). Infinite executions and traces (included in \(T(Prg)\)) are defined similarly. 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; Vanoverbergh 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.

ConSpec Policy Syntax A ConSpec policy \( P \) consists of a security state declaration of the shape

\[
\text{SECURITY STATE } T_{s_1}, \ldots, T_n s_n; \tag{1}
\]

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:

\[
\text{modifier } \begin{cases} \text{BEFORE} & \begin{cases} \text{ON } z \end{cases} \\ \text{AFTER} \text{ or } \text{EXCEPTIONAL} & \end{cases} \begin{cases} [T y =] \text{c.m}(T_{1} x_{1}, \ldots, T_{n} x_{n}) \end{cases} \begin{cases} \text{ON } z \end{cases} \begin{cases} \text{PERFORM } G_{1} \rightarrow \{ F_{1} \} \ldots G_{m} \rightarrow \{ F_{m} \} \end{cases} \begin{cases} \text{ELSE } \{ F \} \end{cases} \tag{2} \end{cases}
\]

where \( \text{modifier} \) is either \text{BEFORE}, \text{AFTER} or \text{EXCEPTIONAL}, and \( T_{1}, \ldots, T_{n} \) are the return and argument types of \( \text{c.m} \). \text{BEFORE} rules refer to pre-actions, and \text{AFTER} and \text{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 to be \textit{security relevant} and we refer to invocations and returns of this method as \textit{security relevant actions}. For instance, if a \text{BEFORE} rule for method \( \text{c.m} \) of a given signature is present then invocations of \( \text{c.m} \) of that signature are security relevant, but if no \text{AFTER} rule is present, normal returns are not regarded as security relevant. There is at most one rule defined for each of the three event modifiers. The return value specification is absent for \text{BEFORE} rules. Each clause of the shape \( G_{i} \rightarrow \{ F_{i} \} \), or the clause \text{ELSE } \{ F \} \) expresses a (conditional) update of the security state in the obvious way. The \text{ELSE} clause is syntactic sugar for a clause with a constantly true guard. The callee qualifier \text{ON } z and the \text{ELSE} clause are both optional except for \text{AFTER} and \text{EXCEPTIONAL} rules for which the \text{ELSE} clause is required. Hence a policy can never forbid a return from an API method.
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)
  } ELSE { }

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

The guard and update expressions $G_i$, $F_j$ and $F$ 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 so no ELSE clause is present) the rule is not allowed to fire. This indicates a security violation and program execution must be terminated.

We describe the expression syntax only by example in this paper. Additional examples are given in Section 5 below. 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.

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.sendFile. 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.
— 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.

ConSpec Semantics A ConSpec policy $\mathcal{P}$ specifies a deterministic automaton $(Q, \Sigma, \delta, q_0)$, 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 $E = C_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{n-1}} C_n$. 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 $\Sigma$ is the set of observable actions. The state space $Q$ is the set of all type safe

...
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.

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 \(Prg\) is 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 \(I\) that for each policy \(\mathcal{P}\) and program \(Prg\) produces a program \(I(\mathcal{P}, Prg)\) with embedded policy checking functionality. In the setting we study here, Inliners are limited to rewrite the program, and may not alter the implementation of the monitored events, i.e. that security relevant API calls. This design criterion is adopted as in our setting, security relevant actions represent events that have some externally observable effect such as sending or receiving a file, or switching on or off a light. We view the security relevant API calls as proxies for these events. In case an API method is non-native it could be considered as part of the program and so be subject to inlining as well. This could even be done for non-native methods in some cases. Ultimately, however, a level (system calls, physical I/O) will reached for which inlining no longer applies.

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 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 \(I\) is secure if, for every program \(Prg\) and policy \(\mathcal{P}\), every trace of the inlined program \(I(\mathcal{P}, Prg)\) adheres to \(\mathcal{P}\), i.e.

\[ T(I(\mathcal{P}, Prg)) \subseteq \mathcal{P}. \]
Transparency states that the policy adherent behavior of the client program should be preserved by the inliner.

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

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

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

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

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

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.

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 can not 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 after $c.m$ has been called (but not necessarily returned). So the trace $T_1 = (\text{tid}, c.m, o, v)^\uparrow, (\text{tid}', c.n, o', v')^\uparrow$ is allowed by the policy, but the trace $T_2 = (\text{tid}', c.n, o', v')^\uparrow, (\text{tid}, c.m, o, v)^\uparrow$ 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 would have to introduce a happens before relation between $(\text{tid}, c.m, o, v)^\uparrow$ and $(\text{tid}', c.n, o', v')^\uparrow$. There is, however, no way such a happens before relation can be added by the inliner since after the call has been made, the control lies within the API method which is not to be altered. Thus the inlined program will either have both traces (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 transparent and secure for the policy $\mathcal{P}$ in Figure 3. □

An inliner could “over approximate” and let the monitor release a lock after the API
SECURITY STATE
boolean ok = false;

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

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

Fig. 3. Not enforceable by inlining.

SECURITY STATE
boolean ok = false;

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

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

Fig. 4. 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.

method has returned, but in that case the monitor would actually be enforcing the
stronger policy shown in Figure 4.

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 any \( i \) and \( j \) such that \( i < j \),
\( T_i \) is a post-action and \( T_j \) is a pre-action, \( \pi(i) < \pi(j) \).

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. The definition also implies that actions within a single thread can never be permuted: within a thread, pre- and post-actions are strictly interleaved in our program model.

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 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 secure and transparent for a non-race-free policy.

*Proof.* (Similar to the proof of Theorem 1.) Let $\mathcal{P}$ be a non-race-free policy. 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 secure, transparent and conservative inliner is said to be *inlineable*. The corollary below follows immediately.

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

*Proof.* Let $\mathcal{P}$ be an arbitrary inlineable policy. By definition there exists a 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-actions are right-movers and all post-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 $\alpha_1, \alpha_2$ in different threads where $\alpha_1$ is a pre-action or $\alpha_2$ is a post-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 $q_1$ of the security automaton corresponding to a given policy, and for any pre-action $\alpha_1$ and post-action $\alpha_2$ 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. □

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 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).

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. 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 multi-threaded 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 runtime 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, \( I_{Ex} \), takes a policy with security state definition and event rules of the shapes (1) and (2) and applies it to a Java bytecode program. The inliner uses static fields \( s_i \) of type \( T_i \) 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
Inlined label Instruction

<table>
<thead>
<tr>
<th>Inlined label Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L$: lock SecState</td>
</tr>
<tr>
<td>store arguments</td>
</tr>
<tr>
<td>store callee</td>
</tr>
<tr>
<td>before$G_1$: [eval($G_{1,b}$)]</td>
</tr>
<tr>
<td>ifeq before$G_2$</td>
</tr>
<tr>
<td>[eval($F_{1,b}$)]</td>
</tr>
<tr>
<td>goto before$G_2$</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>before$G_m$: [eval($G_{m,b}$)]</td>
</tr>
<tr>
<td>ifeq beforeElse</td>
</tr>
<tr>
<td>[eval($F_{m,b}$)]</td>
</tr>
<tr>
<td>goto beforeEnd</td>
</tr>
<tr>
<td>beforeElse: [eval($F_{0}$)]</td>
</tr>
<tr>
<td>beforeEnd: restore callee</td>
</tr>
<tr>
<td>restore arguments</td>
</tr>
<tr>
<td>unlock SecState</td>
</tr>
<tr>
<td>invoke: invokevirtual $c.m$</td>
</tr>
<tr>
<td>invokeDone: lock SecState</td>
</tr>
<tr>
<td>store return value</td>
</tr>
<tr>
<td>after$G_1$: [eval($G_{1,a}$)]</td>
</tr>
<tr>
<td>ifeq after$G_2$</td>
</tr>
<tr>
<td>[eval($F_{1,a}$)]</td>
</tr>
<tr>
<td>goto after$G_2$</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>after$G_m$: [eval($G_{m,a}$)]</td>
</tr>
<tr>
<td>ifeq afterElse</td>
</tr>
<tr>
<td>[eval($F_{m,a}$)]</td>
</tr>
<tr>
<td>goto afterEnd</td>
</tr>
<tr>
<td>afterElse: [eval($F_{a}$)]</td>
</tr>
<tr>
<td>afterEnd: restore return value</td>
</tr>
<tr>
<td>unlock SecState</td>
</tr>
<tr>
<td>goto done</td>
</tr>
<tr>
<td>exc$G_1$: lock SecState</td>
</tr>
<tr>
<td>store exception</td>
</tr>
<tr>
<td>[eval($G_{1,e}$)]</td>
</tr>
<tr>
<td>ifeq exc$G_{2,e}$</td>
</tr>
<tr>
<td>[eval($F_{1,e}$)]</td>
</tr>
<tr>
<td>goto excEnd</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>exc$G_m$: [eval($G_{m,e}$)]</td>
</tr>
<tr>
<td>ifeq excElse</td>
</tr>
<tr>
<td>[eval($F_{m,e}$)]</td>
</tr>
<tr>
<td>goto excEnd</td>
</tr>
<tr>
<td>excElse: [eval($F_{e}$)]</td>
</tr>
<tr>
<td>excEnd: restore exception</td>
</tr>
<tr>
<td>unlock SecState</td>
</tr>
<tr>
<td>excReleased: athrow</td>
</tr>
<tr>
<td>exit: iconst $−1$</td>
</tr>
<tr>
<td>invokestatic System.exit</td>
</tr>
<tr>
<td>done:</td>
</tr>
</tbody>
</table>

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

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 $G_{i,t}, F_i, F_{i,t}, t \in \{b,a,e\}$ 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 $eval(G_{i,t}), eval(F_i), eval(F_{i,t})$ that behave as required. In particular, the callee is extracted from local variable 0, arguments from local variables $1, \ldots, n$, 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
Fig. 7. Exception handler array modifications

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke</td>
<td>invokeDone</td>
<td>excG_1</td>
</tr>
<tr>
<td>L</td>
<td>excReleased</td>
<td>exit</td>
</tr>
<tr>
<td>exit</td>
<td>done</td>
<td>exit</td>
</tr>
</tbody>
</table>

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 \( P_2 \) (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, but 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 \( \text{EXCEPTIONAL} \) 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 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 \( P_{rg} \) and race-free policy \( P, T(I_{Ex}(P, P_{rg})) \subseteq P \). 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 \( I_{Ex}(P, P_{rg}) \) with the corresponding monitor actions, i.e. actions of the inlined code manipulating the inlined security state. We use the notation \( \text{mon}(\alpha) \) for the monitor action corresponding to the observable action \( \alpha \). The monitor action \( \text{mon}(\alpha) \) 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 \text{monitorexit}, corresponding to one of the unlocking events in Figure 6 for the action \( \alpha \). 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 \( \text{mon}(\alpha) \) 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-\( \tau \) 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 \( \beta \) range over extended actions.

Pick now an execution \( E \) of an inlined program \( I_{Ex}(P, P_{rg}) \), 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 \( \alpha \) is immediately preceded by the corresponding monitor action \( \text{mon}(\alpha) \), and each post-action \( \alpha' \) is immediately succeeded by its corresponding monitor action \( \text{mon}(\alpha') \).

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

Proposition 3. Let \( E \) be an execution of \( I_{Ex}(P, P_{rg}) \). Then \( \tau_m(E) \in P \).

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 \( P \) is race-free. For any execution \( E \) of \( I_{Ex}(P, P_{rg}) \) 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 \( \tau_c(E) \). If \( n = m \) we are done so assume \( m < n \). Say that \( \beta_{m-1} \) is produced by thread \( t \). Note first that \( \beta_{m-1} \) can be either a pre-action or a monitor post-action as \( E' \) is serial, and that \( \beta_m \) can be either an post-action or a monitor pre-action. For the latter point assume for a contradiction that \( \beta_m \) is a pre-action. Then \( \beta_m \) must be produced by a thread \( t' \neq t \), by the control structure of the inlining algorithm, Figure 6. The last action in \( \tau_c(E') \) by
thread $t'$ must be a monitor pre-action $\beta_l = \text{mon}(\beta_m)$ for $0 \leq l < m - 1$ 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 $\beta_m$ is a monitor post-action is similar.

Now, if $\beta_m$ is an post-action, say, then thread $t$ is at one of the control points $\text{invokeDone}$ or $\text{excG}_1$. Either $\text{mon}(\beta_m) = \beta_{m'}$ for some $m' > m$ or else thread $t$ does not produce any extended actions in $\tau(E')$ 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 $\beta_k$ for $m \leq k$ and $k \neq m'$ remain schedulable, even after scheduling $\text{mon}(\beta_m)$ right after $\beta_{m'}$. 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 $\beta_m$ is a monitor pre-action $\text{mon}(\alpha)$. If $\beta_{m+1} = \alpha$ we are done. Otherwise $\beta_{m+1}$ is a monitor action or non-monitor action of another thread, and regardless which, by rescheduling, $\beta_m$ can moved right until it is left adjacent to $\alpha$. 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 $P$ is race-free then $I_{Ex}$ is secure, i.e. $T(I_{Ex}(P, Prg)) \subseteq P$.

**Proof.** Pick any execution $E$ of $I_{Ex}(P, Prg)$. Use Lemma 1 to convert $E$ to an execution $E'$ with the property that $\tau(E) = \tau(E') = \tau_m(E') \in P$ 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 $L$, $\text{invokeDone}$, or $\text{excG}_1$ can only exit that block through the corresponding labels $\text{invoke}$, $\text{done}$, or by rethrowing the original exception, or else by invoking $\text{System.exit}$. Moreover, up to variables accessible only to the inlined code fragments, and provided $\text{System.exit}$ 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 $I_{Ex}(P, Prg)$ obtain an execution $\text{erase}(E)$ of $Prg$ such that $\tau(E)$ is a prefix of $\tau(\text{erase}(E))$, and hence $\tau(E) \in T(Prg)$. We refrain from elaborating the details and merely state:

**Theorem 4.** The inliner $I_{Ex}$ 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 $I_{Ex}$ is transparent.

Proof. Consider an execution $E$ of Prg such that $\tau(E) \in \mathcal{P}$. From $E$ construct another execution $E'$ of $I_{Ex}(\mathcal{P}, Prg)$ 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 $I_{Ex}$ 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 (Foundation, 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 (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 `HttpServletRequest.service` method.
8.2. Case Study 2: HTTP Authentication

In this case study we look at the HTTP authentication mechanism (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.isUserInRole 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 behavior enforced by the inliner is in this case a “same origin policy” for browser redirects. That is, the applet is only allowed to redirect the browser to URLs from within the same domain as 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 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).

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

Table 2. Quantitative 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) for enforcing 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 certification, 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 one (a classic PCC scenario):

1. A code producer develops an application, and ensures it complies with the producer policy by inlining that policy. 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, they want to check that the code satisfies their consumer policy. Each consumer can have a different consumer policy.

3. In order to help a consumer with this check, the producer ships a certificate with the code. This certificate will depend on the consumer policy: it will contain (elements of) a proof that the code complies with the consumer policy.

4. The code consumer uses a checking algorithm that verifies whether 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 check the security of programs inlined using techniques similar to the one we discussed in this paper. The contribution of this section is that we show that, for this type of inliner, a checker for inlined multi-threaded Java programs can be built using established program verification techniques for 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 (\textit{N}), where the \textit{SecState} lock is not held. If all threads are in this \textit{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 (\textit{LB}), where the inliner is updating its state in anticipation of an upcoming security relevant call.

3. An unlocked before phase (\textit{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 (\textit{C}) where the actual security relevant call is executing.
An unlocked after phase (UA), where things might be happening between the (normal) return of the call, and the inlined security state update.

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

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 straightforward.

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 wider classes of inliners to be supported than the one introduced above. One such example is briefly discussed in the conclusion.

A key part of the checking algorithm is to recognize these phases, and certificates should contain sufficient information to allow the checker to statically determine that the inlined security state is in sync with an ideal automaton state which the receiver of the code can inject into the application as a reference, used only for verification purposes. This use of a reference automaton is similar to the approach taken in (Aktug et al., 2009) for the case of sequential Java.

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 \( I_{Ex} \) 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 \texttt{ldc SecState} instruction, followed by a \texttt{monitor-enter}.  

---

\*Security Monitor Inlining and Certification for Multithreaded Java*
— The \( L_{ub} \) and \( L_n \) labels point to a \texttt{monitorexit} instruction preceded by a \texttt{ldc \textit{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 \texttt{System.exit()}.

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

— No other \texttt{ldc \textit{SecState}} instructions occur anywhere in the program. This makes sure the \texttt{SecState} class object is only used for acquiring or releasing a lock, and no other aliases to the object are created.

— \texttt{putstatic} and \texttt{getstatic} for fields of the \texttt{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 \texttt{SecState} lock will be true at each program point where the \texttt{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 only 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 \textit{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 \textit{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 \textit{augmented programs}.

A ghost instruction has the shape

\[
(x^g := a_1 \rightarrow e_1 | \ldots | a_n \rightarrow e_n)
\]

where \( x^g \) is a vector of \textit{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 \texttt{fails} and the execution is said to be \texttt{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.
<table>
<thead>
<tr>
<th>Identifier</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>msg[g]</td>
<td>A global vector representing the ghost security state, i.e. a type correct assignment to the security state variables as in Section 3.</td>
</tr>
<tr>
<td>status[gl]</td>
<td>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.</td>
</tr>
<tr>
<td>arg[gl], tid[gl], o[gl], r[gl]</td>
<td>Local variables to hold the arguments of security relevant calls during the call, resp. calling thread, callee, and return value.</td>
</tr>
</tbody>
</table>

Table 3. Variables introduced by ghost inliner.

**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.

\[
\langle x^g := s_0 \land x^g < 10 \rightarrow x^g + 1 \mid \neg s_0 \rightarrow x^g \rangle
\]

invoke \(c.m\)

Other...

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

\[
\begin{align*}
&\langle tid^{gl} := \text{Thread.currentThread()} \rangle \\
&\langle o^{gl} := s_0 \rangle \\
&\langle arg^{gl} := (s_1, \ldots, s_n) \rangle \\
&\langle ms^{gl} := status^{gl} = \text{ready} \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl})^\uparrow) \rangle \\
&\langle status^{gl} := \text{before}_c.m \rangle
\end{align*}
\]

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

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

\[
\begin{align*}
&\langle status^{gl} := status^{gl} = \text{before}_c.m \rangle \\
&\quad \land o^{gl} = s_0 \land arg^{gl} = (s_1, \ldots, s_n) \rightarrow \text{ready}
\end{align*}
\]

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

\[
\begin{align*}
&\langle r^{gl} := s_0 \rangle \\
&\langle status^{gl} := status^{gl} = \text{ready} \rightarrow \text{return}_c.m \rangle
\end{align*}
\]

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

\[
\begin{align*}
&\langle ms^{gl} := status^{gl} = \text{return}_c.m \rightarrow \delta((tid^{gl}, c.m, o^{gl}, arg^{gl}, r^{gl})^\uparrow) \rangle \\
&\langle status^{gl} := \text{ready} \rangle
\end{align*}
\]

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 $\mathcal{I}^\emptyset(\mathcal{P}, \mathcal{P}_{rg})$ be the result of ghost inlining $\mathcal{P}_{rg}$ with respect to policy $\mathcal{P}$ and $\mathcal{P}_{rg}$’s certificate. Similar to Section 7 we compare the observable actions of $\mathcal{P}_{rg}$ with ghost actions $\alpha^g$ of $\mathcal{I}^\emptyset(\mathcal{P}, \mathcal{P}_{rg})$. 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 $\mathcal{I}^\emptyset(\mathcal{P}, \mathcal{P}_{rg})$. Then $\tau_g(E) \in \mathcal{P}$.

**Proof.** Let $\tau_g(E) = \alpha_0^g \ldots \alpha_n^g$ 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 $\alpha_n$ 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 $\alpha_n$ an post-action is virtually identical and left to the reader.

With prop. 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 $\mathcal{P}_{rg}$ is a correct program, then $T(I^g(\mathcal{P}, \mathcal{P}_{rg})) \subseteq \mathcal{P}$

### 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 ghost inlines the code 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 inline ghost instructions fail.

Checking that an arbitrary program with inline 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 \texttt{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 \texttt{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:

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

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.

9.5. Creating certificates for the example inliner

Finally, we show that a code producer that uses the concrete inliner $I_{Es}$ 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 $(c'.m', L_{lb}, L_{ub}, L_{call}, L_{la}, L_{na})$
marking the beginning and ending of the different phases of the inliner. Computing these for $\mathcal{I}_{Ex}$ is trivial.

— An invariant $I(ms, ms^9)$ 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. □

### 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 practically useful 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 (Vanoverberghede 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 straightforward, syntactic cross product construction for policies, $I(\prod_i P_i, \prod_i P_{rg})$. 
Alternatively, the monitors can be nested by inlining one policy at a time: $I(P_n, \ldots, I(P_2, I(P_1, P_{gy})) \ldots)$. If the example inliner, $I_{Ex}$, is used, the certification approach described above is general enough to easily certify the fully inlined program from certificates for each policy $P_i$ 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

Thanks to Irem Aktug, Dilian Gurov and Dries Vanoverberghe for useful discussions on many topics related to monitor inlining.

References


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).

Basic Conventions We use $c$ for class names, $m$ for method names, and $f$ for field names. Class names are fully qualified. A type $T$ 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.

Configurations, Transitions, and Programs A configuration $C = (h, \Lambda, \Theta)$ consists of a heap $h$, a lock map $\Lambda$ which maps an object reference $o$ 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 a thread configuration, often denoted by $\theta$. 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 $C \rightarrow C'$ is given in the appendix for key instructions and configuration types. 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)).

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 \( \to \) 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 \( \to \) 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.1. 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 use the following auxiliary operations and predicates:

- \( v :: s \) pushes \( v \) on top of stack \( s \)
- \( \text{handler}(M, h, o, pc) \) returns the proper target label given \( M \)'s exception handler \( H \), heap \( h \), throwable \( o \) and pc \( pc \) in the standard way:
  \[
  \text{handler}(M, h, o, pc) = \text{handler}_2(H, h, o, pc) \quad \text{with} \quad H \text{ the exception handler of } M
  \]
  \[
  \text{handler}_2(\epsilon, h, o, pc) = \bot
  \]
  \[
  \text{handler}_2((b, e, t, c) \cdot H, h, o, pc) =
  \begin{cases}
  t & \text{if } b \leq pc < e \text{ and } h \vdash o : c \\
  \text{handler}_2(H, h, o, pc) & \text{otherwise}
  \end{cases}
  \]
- \( v \) 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 \( (\square) \) (used for API calls). (See Section 2)

**Local Variables and Stack Transitions**

\[
\Theta[tid] \rightarrow \theta \\
(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \rightarrow \theta])
\]

\[
M[pc] = \text{aload } n
\]

\[
(M, pc, s, l) :: R \rightarrow (M, pc + 1, l(n) :: s, l) :: R
\]
Heap transitions As discussed in Section 2, field reads return an arbitrary value, and these rules should be complemented with the Java memory mode constraints.

\[
\begin{align*}
M[pc] &= \text{astore } n \\
(M, pc, v :: s, l) &:: R \rightarrow (M, pc + 1, s, l[n \rightarrow v]) :: R \\
M[pc] &= \text{athrow} \\
(M, pc, o :: s, l) &:: R \rightarrow (o) :: (M, pc + 1, o :: s, l) :: R \\
M[pc] &= \text{goto } L \\
(M, pc, s, l) &:: R \rightarrow (M, L, s, l) :: R \\
M[pc] &= \text{iconst}_n \\
(M, pc, s, l) &:: R \rightarrow (M, pc + 1, n :: s, l) :: R \\
M[pc] &= \text{ldc } c \\
(M, pc, s, l) &:: R \rightarrow (M, pc + 1, c :: s, l) :: R \\
M[pc] &= \text{ifeq } L \quad n = 0 \\
(M, pc, n :: s, l) &:: R \rightarrow (M, L, s, l) :: R \\
M[pc] &= \text{ifeq } L \quad n \neq 0 \\
(M, pc, n :: s, l) &:: R \rightarrow (M, pc + 1, s, l) :: R
\end{align*}
\]

Locking instructions

\[
\begin{align*}
\Theta(tid) &= (M, pc, v :: s, l) :: R \\
M[pc] &= \text{monitorenter} \\
\Lambda(v) &= \bot \\
(h, \Lambda, \Theta) &\rightarrow (h, \Lambda[v \mapsto \text{tid}], \Theta[\text{tid} \mapsto (M, pc + 1, s, l) :: R]) \\
\Theta(tid) &= (M, pc, v :: s, l) :: R \\
M[pc] &= \text{monitorexit} \\
\Lambda(v) &= \text{tid} \\
(h, \Lambda, \Theta) &\rightarrow (h, \Lambda[v \mapsto \text{tid}], \Theta[\text{tid} \mapsto (M, pc + 1, s, l) :: R])
\end{align*}
\]

Exceptional Transitions

\[
\begin{align*}
\Theta(tid) &= (o) :: (M, pc, s, l) :: R \\
p'c &= \text{handler}(M, h, o, pc) \\
p'c &\neq \bot \\
(h, \Lambda, \Theta) &\rightarrow (h, \Lambda, \Theta[\text{tid} \mapsto (M, pc', s, l) :: R])
\end{align*}
\]
\[\Theta(tid) = (o) :: (M, pc, s, l) :: R\]
\[\text{handler}(M, h, o, pc) = \bot\]

\[
(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto (o) :: R])
\]

**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.

\[
\Theta(tid) = (M, pc, o :: v :: s, l) :: R
\]
\[
M[pc] = \text{invokevirtual } c.m \quad c \in API
\]

\[
(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto (\Box) :: (M, pc + 1, s, l) :: R])
\]

Exceptional return from an API method:

\[
\Theta(tid) = (\Box) :: R
\]

\[
(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto (\Box) :: R])
\]

Normal return from an API method:

\[
\Theta(tid) = (\Box) :: (M, pc, s, l) :: R
\]

\[
(h, \Lambda, \Theta) \rightarrow (h, \Lambda, \Theta[tid \mapsto (M, pc, v :: s, l) :: R])
\]
Appendix B

Provably Correct IRMs for Multithreaded Java-like Programs
Provably Correct Inline Monitoring for Multithreaded Java-like Programs

Mads Dam
ACCESS Linnaeus Centre
Royal Institute of Technology (KTH), Sweden
mfd@kth.se +46 8 790 6229

Bart Jacobs
Katholieke Universiteit Leuven, Belgium
bart.jacobs@cs.kuleuven.be +32 16 32 7825

Andreas Lundblad
School of Computer Science and Communication
Royal Institute of Technology (KTH), Sweden
landreas@kth.se +46 8 790 8408

Frank Piessens
Katholieke Universiteit Leuven, Belgium
frank.piessens@cs.kuleuven.be +32 16 32 7603 (Corresponding author)

Abstract

Inline reference monitoring is a powerful technique to enforce security policies on untrusted programs. The security-by-contract paradigm proposed by the EU FP6 S3MS project uses policies, monitoring, and monitor inlining to secure third-party applications running on mobile devices. The focus of this paper is on multi-threaded Java bytecode. An important consideration is that inlining should interfere with the client program only when mandated by the security policy. In a multi-threaded setting, however, this requirement turns out to be problematic. Generally, inliners use locks to control access to shared resources such as an embedded monitor state. This will interfere with application
program non-determinism due to Java’s relaxed memory consistency model, and rule out the transparency property, that all policy-adherent behaviour of an application program is preserved under inlining. In its place we propose a notion of strong conservativity, to formalise the property that the inliner can terminate the client program only when the policy is about to be violated. An example inlining algorithm is given and proved to be strongly conservative. Finally, benchmarks are given for four example applications studied in the S$^3$MS project.

Keywords: Security-by-contract; Runtime monitoring; Monitor inlining.

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. Potentially dangerous actions by a client program are intercepted and routed to a policy decision point (pdp) in order to determine whether the actions should be allowed to proceed or not. In turn, these decisions are routed to a policy enforcement point (pep), responsible for ensuring that only policy-compliant actions are executed.

The Security of Software and Services for Mobile Systems (S$^3$MS) project has investigated the use of such program monitors for ensuring the security of communicating mobile applications. This paper focuses on one of the key scientific results of the S$^3$MS project: the design and implementation of inlined reference monitors in multithreaded Java.

The idea of monitor inlining is to push policy decision and enforcement functionality into the client programs themselves, by embedding a security state into the client program, and using code rewriting to ensure this em-
bedded state is correctly queried and updated at the appropriate points. When applicable, such an approach has a number of advantages:

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

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

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

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

The starting point for much previous work on monitor inlining has been security automata in the style of Schneider [20]. The PoET/PSLang toolset by Erlingsson [22] 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. As an alternative we propose to use a dedicated policy specification language ConSpec [2], similar to PSLang, but more constrained in order to allow for a decidable containment problem. The ConSpec language, in particular, is designed to monitor only accesses to some specific API, determined by the application program under consideration.

Formal correctness of inlining for the case of sequential bytecode has been examined in [1] for Java, and in [23] for .NET. In particular, [1] shows
how to generate bytecode level specification annotations under rather modest assumptions on the inliner, by fixing control points immediately before and after each method call at which the embedded state must be correctly updated.

Other recent work on monitoring and monitor inlining includes work on edit automata [3, 14, 13], security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event insertions, for instance to recover gracefully from policy violations. Type-based approaches for security policy enforcement have been considered by a number of authors, e.g. [21, 24, 4, 9]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al [11]. The Mobile system uses a simple library extension to Java bytecode to help managing updates to the security state. The use of linear types allows a type system to localize security-relevant actions to objects that have been suitably unpacked, and the type system can then use this property to check for policy compliance.

Our contribution is to propose correctness criteria for monitor inlining in the case of multi-threaded bytecode programs, and to formally prove correctness for an example inliner. In particular we address the implications of relaxed memory consistency models in intermediate bytecode languages such as JVML and MSIL. This turns out to be non-trivial, since locks introduced by the inliner to control access to shared resources such as the embedded security state will in general interfere with application program nondeterminism, and rule out the transparency property [13], that all policy-adherent behaviour of an application program is preserved under inlining. In its place we propose a notion of strong conservativity, to formalise the property that any complete trace of an inlined program is either a policy-compliant complete trace of the uninlined program, or else it is the truncation of a trace...
of the uninlined program at the point of policy violation.

The paper is structured as follows. In section 2 we survey the S³MS project context, and briefly introduce the ConSpec language. In section 3 we present those parts of a model for multi-threaded Java bytecode execution needed to understand the rest of the paper, in particular the concepts of legal execution and observable trace, and we discuss the treatment of API calls. Section 4 briefly introduces security automata, to pin down the key concept of policy compliance. Section 5 present the main results of the paper: Correctness criteria, example inliner, and the correctness proof. Section 6 gives benchmark results for 4 sample mobile applications, and section 7 concludes.

2 Security by Contract

The key objective of the S³MS project [19] is the creation of a framework and technological solutions for trusted deployment and execution of communicating mobile applications in heterogeneous environments. A contract-based security mechanism lies at the core of the framework [7, 5].

Application contracts specify the security behaviour of mobile applications, and can be matched with device policies specifying acceptable behaviour of applications on the device.

This section provides a brief summary of the security-by-contract (SxC) paradigm developed in the S³MS project. We start by analyzing the requirements for a security architecture for mobile applications and services, and go on to discuss how the SxC paradigm fulfills these requirements. Then we discuss how monitor inlining fits in this picture, and we show that the contribution of this paper – provably correct monitor inlining for multithreaded Java – is an essential ingredient of SxC.
2.1 Security for mobile applications and services

Mobile phones and personal digital assistants have evolved over the past years to become general purpose computation platforms. Many of these devices support downloading third party applications built on either the .NET Compact Framework, or Java Micro Edition. However, supporting applications from potentially untrustworthy sources comes with a serious risk: malicious or buggy applications on a phone can lead to denial of service, loss of money, leaking of confidential information on the device and so forth.

Current devices already provide certain countermeasures against these threats, with support for sandboxing and code signing. The key idea is that unsigned code is severely limited in what it can do on the device, i.e. it runs in a strict sandbox. Code that is signed by a trusted party can break out of the sandbox. The device has a keystore that contains the public keys of trusted parties.

This security model has a number of drawbacks. First, it is not flexible: applications either run in a restricted sandbox, or have full power. Many interesting types of applications can not run in a sandbox. Examples of case studies considered in the S³MS project include:

- multiplayer games, where communication between the players and/or a game server is essential,
- a traffic jam reporter, that interacts with the GPS device and that sends and receives traffic information to a server,
- social networking applications, where users can track the location of their friends on their mobile device.

None of these case studies can function in a sandbox. On the other hand, the risk of giving full power to third party applications is substantial.
A second disadvantage of the current security model is that no precise meaning is associated with the signatures of trusted third parties: a signature either means that the application comes from the software factory of the signatory or that the signatory vouches for the software, but there is no clear definition of what guarantees it offers. Hence, device owners trust the third party both for (a) appropriate vetting of applications, and (b) using a suitable notion of good behavior. Incidents [18] show that the current security model is inappropriate.

2.2 Application contracts and policies

The SxC paradigm addresses the shortcomings of the current mobile device security model.

A key ingredient is the notion of an application security contract. Such a contract specifies the security behavior of the application. Technically, a contract is a security automaton in the sense of Schneider [20], and it specifies an upper bound on the security-relevant behavior of the application: the sequences of security-relevant events that an application can generate are all in the language accepted by the security automaton.

Mobile devices are equipped with a security policy, a security automaton that specifies the behavior that is considered acceptable by the device owner. The key task of the S3MS device run-time environment is to ensure that all applications will comply with the device security policy. To achieve this, the run-time can make use of the contract associated with the application (if it has one), and of a variety of policy enforcement technologies:

- monitor inlining, a program rewriting technique to ensure that a program complies with a given policy,
- contract-policy matching [16], the process of checking whether the se-
curity behaviour specified in a contract is a subset of the allowed security behaviour specified in a policy,

- explicit run-time monitoring for compliance with policies.

All these enforcement technologies can run on-device. Some of them (matching and inlining) can also be provided as a web service that the device can call during the installation of an application on the device.

An application contract is a statement about the behavior of an application, and there is no a-priori guarantee that this statement is correct. Testing and static analysis can be used at development time to increase confidence in the contract. In addition, monitor inlining of the contract at development time can provide strong assurance of compliance.

If the device makes security decisions based on the contract (for instance when it uses contract-policy matching), then there is a clear need to transfer these development-time guarantees to the device that will eventually execute the application. Without a secure transfer of these guarantees, it would be easy for an attacker to modify either the application or the contract. Two key technologies support this transfer:

1. A cryptographic signature by a trusted third party can vouch for application-contract compliance. Note the difference with the use of signatures in the traditional mobile device security model. In the S3MS approach, a signature has a clear semantics: the third party claims that the application respects the supplied contract [8]. Moreover, what is important is the fact that the decision whether the contract is acceptable or not remains with the end user.

2. Proof-carrying-code techniques can be used, to enable verification on the mobile device of contract compliance proofs constructed by the
program developer. Building on [1], we have realized such a framework for sequential Java, and a publication about this is in preparation.

2.3 Example: Mobile 2-player Chess

As an example application (also used as a case study in the S3MS project), we consider a two-player chess game running on the .NET Compact Framework. This application supports standalone games (where two players play chess against each other on the same device), as well as games between two devices communicating over either a TCP/IP network, or using text messages (SMS’s). Chess games rarely take more than 70 moves per player to finish, and the chess program enforces a hard upper limit of 100 moves. As a consequence, the program’s contract can specify hard upper bounds on its use of communication resources. One move either takes 20 bytes of TCP traffic, or 1 SMS. Hence, one run of the program will consume at most 2000 bytes of network traffic, and send at most 100 SMS messages. The contract in Fig. 1 specifies this.

The contract is expressed in the ConSpec policy language [2]. A ConSpec specification tells when and with what arguments an API method may be invoked. If the specification has one or more constraints on a method, the method is said to be a security relevant method (srm).

The first part of the contract declares the security state. This security state contains a definition of all the variables that will be used in the contract, and defines the set of states of the corresponding security automaton. In the example contract, two state variables maintain (1) the number of bytes that have already been sent over the network, and (2) the number of SMS messages that have been sent.

The security state declaration is followed by one or more clauses. Each
clause represents a rule on a security-relevant API method call. These rules can be evaluated before the method is called, after the method is called, or when an exception occurs. A clause definition consists of the 'BEFORE', 'AFTER' or 'EXCEPTIONAL' keyword, the signature of the method on which the rule is defined, and a list of guard/update blocks. The guard is a boolean expression that is evaluated when a rule is being processed. The guard may mention variables from the security state declaration, arguments given in the method call and the return value (if it is part of an after clause). If the guard evaluates to true, the corresponding update block is executed. All state changes that should occur can be incorporated in this update block. When a guard evaluates to true, the evaluation of the following guards (and consequently the potential execution of their corresponding update blocks) is skipped.

If none of the guards evaluates to true, this means the contract does not allow the method call. For example, in Fig. 1, if the current state of the policy has bytesSent = 2000, then a call to the Send method with an array of length 20 will fail all the guards.

Note that the contract can be quite specific about the behavior of the application. For instance, the example contract specifies explicitly that the application will only send messages consisting of 20 bytes over the TCP/IP network. The contract also encodes the upper bound of 100 moves enforced by the application.

The contract in Fig. 1 matches with a device policy that limits network traffic to (for instance) 10 kilo bytes. Such a policy is shown in Fig. 2. Note the differences between the contract and the policy: while both are written in ConSpec, and both semantically correspond to security automata, the device policy for instance does not make any assumptions about the size
of messages sent (beyond the fact that the total size of traffic is limited to 10k).

For the remainder of the paper we focus on inlining of policies in multi-threaded Java bytecode. But, the techniques are equally applicable to contracts (instead of policies) and to .NET (instead of Java) [6].

3 Program Model

We assume that the reader is familiar with Java bytecode syntax, the Java Virtual Machine (JVM), and formalisations of the JVM such as [10]. 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.

Classes, Types and Methods  We use $c$ for class names and $m$ for method names. To simplify notation, method overloading is not considered, so a method is uniquely identified by a method reference of the form $M = c.m$. A method definition is a pair $(I, H)$ consisting of an instruction array $I$ and an exception handler array $H$. We use the notation $M[L] = \iota$ to indicate that $I_L$ is defined and equal to the instruction $\iota$. The exception handler array $H$ is a partial map from integer indices to exception handlers. An exception handler $(b, e, t, c)$ catches exceptions of type $c$ and its subtypes raised by instructions in the range $[b, e)$ and transfers control to address $t$, if it is the topmost handler that covers the instruction for this exception type.

Configurations and Transitions  A configuration $C = (h, \Lambda, \Theta)$ of the JVM consists of a heap $h$, a lock map $\Lambda$, 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 $(M, pc, s, r)$, where:

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

We assume a transition relation $\rightarrow_{\text{JVM}}$ on JVM configurations. A thread configuration of the shape $\theta = (M, pc, s, r) :: R$ is calling, if $M[pc]$ is an invoke instruction, and it is returning normally, if $M[pc]$ is a return instruction. For exceptional configurations the top frame has the form $(b)$ where $b$ is an exception object, i.e. an object of class Throwable. Such a configuration is called exceptional. We say that $\theta$ is returning exceptionally if $\theta$ is exceptional, and if $(h, \Lambda, \theta) \rightarrow_{\text{tid}} (h', \Lambda', \theta')$ implies that $\theta'$ is exceptional as well. I.e. the normal frame immediately succeeding the top exceptional frame in $\theta$ is popped in $\theta'$, if $\theta'$ is exceptional as well.

**Programs and Types** For the purpose of this paper we can view a program $P$ as a collection of class declarations determining types of fields and methods belonging to classes in $P$. 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 $P$ and for each $i \geq 0$, $C_i \rightarrow_{JVM} C_{i+1}$. 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. [12]).

Field Accesses and Legal Executions In this paper, we wish to reason about the behavior of arbitrary, possibly malicious, 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 (see Chapter 17 of the Java Language Specification, Third Edition (JLS3)). 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, our transition relation $\rightarrow_{JVM}$ 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 $\rightarrow_{JVM}$ and the memory consistency constraints of JLS3.

An important guarantee provided by JLS3 that we will need 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.

The only other assumption we make about JLS3 in this paper is that JLS3 is monotonic, in the sense that, informally speaking, adding synchronization to a program reduces the set of executions.

**API Method Calls** The only non-standard aspect of $\rightarrow_{\text{JVM}}$ is the treatment of API methods. We assume a fixed API $\mathcal{M}$, consisting of a set of classes for which we have access only to the signature, but not the implementation, of the methods in $\mathcal{M}$. 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. The thread can then proceed only by either 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 exceptional activation record is returned. We assume that the API does not declare any fields visible to the client; therefore, in our model, steps performed by a thread while it is inside an API method activation record do not modify the heap.

Our approach hinges on our ability to recognize API method calls. This property is destroyed by the *reflect* API, which is left out of consideration. Among the method invocation instructions, we discuss here only *invokevirtual*; the remaining invoke instructions are treated similarly.

Given an execution $E$ we define the notion of the observable trace $\omega(E)$
of $E$, as follows:

$$
\omega(C) = \varepsilon
$$

$$
\omega(CC'E) = \alpha \omega(C'E) \text{ if } C \xrightarrow{\alpha}_{\text{JVM}} C'
$$

$$
\omega(CC'E) = \omega(C'E) \text{ if } C \xrightarrow{\tau}_{\text{JVM}} C'
$$

where a transition from $C$ to $C'$ performs an observable action $\alpha$, denoted $C \xrightarrow{\alpha}_{\text{JVM}} C'$, if and only if it transitions from the client code to the API or vice versa. Specifically, we represent a call from a class $d \not\in \mathcal{M}$ bound at run time to a method $c.m$ on an object $o$ with arguments $v$ by a thread $tid$ where $c \in \mathcal{M}$ as $C \xrightarrow{(tid,c.m,o,v)}_{\text{JVM}} C'$, and a normal return from this call with return value $r$ as $C'' \xrightarrow{(tid,c.m,o,v,r)}_{\text{JVM}} C'''$. We represent an exceptional return from this call with exception object $t$ as $C'' \xrightarrow{(tid,c.m,o,v,t)}_{\text{JVM}} C'''$. All transitions other than the above are non-observable, denoted $C \xrightarrow{\tau}_{\text{JVM}} C'$.

If the action refers to a security relevant method it is said to be a security relevant action (sra).

There is one exception to the above definition of observable versus non-observable actions. We consider calls of method `System.exit` to be non-observable. (Furthermore, we assume that such a call is always the last transition of an execution.)

We refer to actions of the form $(tid,c.m,o,v)^\top$, $(tid,c.m,o,v,r)^\top$, and $(tid,c.m,o,v,t)^\flat$ as before actions, after actions, and exceptional actions, respectively, and we collect them in sets $\Omega^\top$, $\Omega^\downarrow$, and $\Omega^\flat$.

We denote the set of executions of a program $P$ against an API $\mathcal{M}$ as $\text{exec}_{\mathcal{M}}(P)$. We define the set $\mathcal{T}(P)$ of the traces of a program $P$ as the
traces of the executions of $P$:

$$\mathcal{T}(P) = \{\omega(E) \mid E \in \text{exec}_M(P)\}$$

We say an execution is complete if it cannot be extended with an additional transition. It follows that either the execution is infinite, or it ends with a call of System.exit, or in the final configuration, all threads are waiting on a lock held by another thread. We define the set $\mathcal{T}_c(P)$ of complete traces of a program $P$ as the traces of the complete executions of $P$.

4 Security Automata

ConSpec policies are formalized in terms of security automata. The notion of security automata was introduced by Schneider [20]. In this paper we view a security automaton as an automaton $A = (Q, \delta, q_0)$ where $Q$ is a countable (not necessarily finite) set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \Omega \rightarrow Q$ is a (partial) transition function, where $\Omega = \Omega^\dagger \cup \Omega^\downarrow \cup \Omega^\updownarrow$ is the set of observable actions. All states $q \in Q$ are viewed as accepting.

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

A security automaton can be derived from a ConSpec policy in the obvious manner. We refer to [1] for details. We assume that after clauses of the Conspec policy to be exhaustive such that an after action can never fail, but it can update the security state.

Definition 1 (Policy Adherence) The program $P$ adheres to security policy $P_A$, if for all executions $E$ of $P$, $\omega(E) \in P_A$. 

16
5 Inlining

By *inlining* we refer to the procedure of compiling a contract into a JVML-based reference monitor and embedding this monitor into a target program. Formally, an inliner is a function $I$ which for each policy $P_A$ and program $P$ produces an inlined program $I(P_A, P)$. The intention is that the inserted code enforces compliance with the policy, and otherwise interferes with the execution of the client program as little as possible.

In this section, we first look at various correctness properties for inliners. Then, we introduce the design of our inliner and we prove its correctness.

5.1 Inlining Correctness Properties

We first look at the traditional correctness properties for inliners: security, conservativity, and transparency. Then, we introduce a number of new correctness properties that deal with complications caused by the setting of multithreaded Java-like programs: strong conservativity, relative strong conservativity, and weak transparency.

For an inliner whose only expected functionality is to intercept and abort execution of an underlying client program in case of policy violation there are three correctness properties of fundamental interest (cf. [13] for the case of edit automata). 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 $I$ is:

- Secure if, for every program $P$, every trace of the inlined program $I(P_A, P)$ adheres to $P_A$, i.e. $T(I(P_A, P)) \subseteq P_A$.***
• Conservative if, for every program $P$, every trace of the inlined program $I(\mathcal{P}_A, P)$ is a trace of $P$, i.e. $T(I(\mathcal{P}_A, P)) \subseteq T(P)$.

• Transparent, if every adherent trace of the client program is also a trace of the inlined program, i.e. if $T(P) \cap \mathcal{P}_A \subseteq T(I(\mathcal{P}_A, P))$.

Recall from Section 3 that the set of traces $T(P)$ of a program $P$ is the set of the sequences $T$ of observable actions (i.e., API calls and normal and exceptional returns from API calls) such that there is a (partial or complete) execution of the program whose observable trace is $T$.

Unfortunately, in case the client program is not well-synchronized, transparency is infeasible in general, because it is not in general possible to perform inlining without introducing extra synchronization and consequently eliminating certain executions. To illustrate this, consider the program of Fig. 3. This program is not well-synchronized, since there are data races on fields $\text{beforeA}$ and $\text{afterA}$. Specifically, threads 1 and 2 do not synchronize their accesses of these fields. In the presence of data races, the semantics of Java allow field accesses to appear out of order; this is necessary to allow the JIT compiler (which compiles bytecode to machine code) and the hardware to perform important optimizations. In the example, suppose the body of method $\text{sra}$ is a simple field assignment. In that case, the JIT compiler can inline this method and then reorder the field accesses, since they are independent. This is why an execution where $r_1$ gets the value 1 and $r_2$ gets the value 0 is a legal execution. As a result, the program has a trace with three $\text{sra}()$ calls.

Now, consider the inlined version of this program. In general, the inlined code needs to access the security state; since multiple security-relevant calls may occur concurrently, these accesses must be synchronized. This means that in general, the inliner inserts synchronization constructs before and
after each sra() call. As a result, the JIT compiler is no longer allowed to move the accesses of beforeA and afterA across the sra() calls, and the execution where r1 equals 1 and r2 equals 0 is no longer legal. Therefore, the inlined program does not have a trace with three sra() calls, which means that the inliner is not transparent.

For this reason, the transparency property is only really meaningful for well-synchronized programs. For this restricted case, however, transparency still serves as a useful correctness check: An inliner which is transparent for well-synchronized clients (and, which is secure and conservative) must necessarily exploit race conditions to interfere in an undesirable way with a client program. However, to allow also for programs that are ill-synchronized we look for alternative correctness criteria.

**Definition 3** The truncation $\text{trunc}_{\mathcal{P}_A}(T)$ of a trace $T$ under a policy $\mathcal{P}_A$ is the greatest prefix $T'$ of $T$ that adheres to $\mathcal{P}_A$.

Thus, if $T$ adheres to $\mathcal{P}_A$, $\text{trunc}_{\mathcal{P}_A}(T) = T$, and otherwise $T$ is of the form $\alpha_0 \cdots \alpha_n$ such that, for some $i : 0 \leq i < n$, $\alpha_0 \cdots \alpha_i \in \mathcal{P}_A$ and $\alpha_0 \cdots \alpha_{i+1} \notin \mathcal{P}_A$.

**Definition 4 (Strong Conservativity)** An inliner $I$ for a given policy $\mathcal{P}_A$ is strongly conservative if, for each program $P$, every complete trace of the inlined program $I(\mathcal{P}_A, P)$ is the truncation of a complete trace of $P$ under $\mathcal{P}_A$:

$$T_c(I(\mathcal{P}_A, P)) \subseteq \text{trunc}_{\mathcal{P}_A}(T_c(P))$$

**Example 1** An abstract version of the program in fig. 3 might have traces $AB$, $BA$, $ABC$ and $BAC$, all complete and all in $\mathcal{P}_A$. Suppose the set of complete traces of $I(\mathcal{P}_A, P)$ is $\{AB, BA\}$. The inliner $I$ is strongly conservative (for this particular program), but not transparent. As another example suppose $P'$ has traces $A$, $AB$, $AC$, $ABC$ such that $A, AB \in \mathcal{P}_A$, but
AC, ABC \not\in \mathcal{P}_A, and AC, ABC, but not A, AB, are complete. Suppose the only trace of \mathcal{I}(\mathcal{P}_A, P') is A (so A is complete). Again, \mathcal{I} is strongly conservative (for the program P') but not transparent.

**Proposition 1** An inliner which is strongly conservative is secure and conservative.

**Proof** For security assume \( T \in \mathcal{T}(\mathcal{I}(\mathcal{P}_A, P)) \). Then we find \( T' \geq T \) such that \( T' \in \mathcal{P}_A \), by strong conservativity, so also \( T \in \mathcal{P}_A \), by prefix closure. For conservativity the argument is similar. \( \square \)

Strong conservativity implies that the inliner does not add new termination or deadlock behavior. But, in a threaded setting inliners typically use locks to access shared resources, in particular the security state. This may constrain the order of actions. In particular, as is the case in this paper, if the security state is locked across the entire security-relevant call, each such call must be completed before a new security-relevant call can take place. But this may not be compatible with constraints induced by the API, as the following example shows.

**Example 2** Consider an API \( \mathcal{M} \) with a barrier method \( m \) that allows two threads to synchronize as follows: When one thread calls \( m \), the thread blocks until the other thread calls \( m \) 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. This inliner is strongly conservative: The notion of complete trace simply does not take constraints induced by the API into account. On the other hand a client program may consist of two threads, each calling \( m \) and then terminating. The inlined version will have one complete trace where one of the threads enters \( m \) and then blocks. An uninlined complete trace will contain two calls and returns of
Thus the inlined complete trace will not be the truncation of an uninlined one at the point of policy violation.

So the definition of strong conservativity needs to be amended to take such order-inducing API calls into account. Note that the JVM semantics of API calls given in section 3 does not do this.

**Definition 5 (Relative Strong Conservativity)** For each program $P$, let $\text{exec}_M^c(P)$ be the set of complete executions of $P$ in the API $M$. An inliner $I$ is strongly conservative relative to the API $M$, if for each policy $P_A$ and each program $P$,

$$\omega(\text{exec}_M^c(I(P_A, P))) \subseteq \text{trunc}_{P_A}(\omega(\text{exec}_M^c(P))) .$$

An implication of this definition is that, if in some execution $E$ in some API the inliner kicks in and blocks an sra $\alpha$, then there will be an execution of the uninlined program which after the trace of $E$ executes $\alpha$. The condition does not guarantee, however, that $E$ without the inliner next would have performed $\alpha$. This is a consequence of our strictly observational definition of strong conservativity; if more precision is needed, one needs to take internal intermediate states into account, e.g. using bisimulation-based techniques.

As we noted above, inliners generally cannot be transparent for ill-synchronized programs. In fact, some reasonable inliners are not transparent even for well-synchronized programs, because they force the start action and the return action of a security-relevant call to occur atomically, for instance by locking (as we do in this paper). In that case there may be client program traces with nonatomic API calls and returns that can not be realized after inlining, only because of execution constraints induced by the inliner. However, these inliners may still be transparent after canonicalization of the
traces with respect to a set of atomic methods:

**Definition 6** A method $m$ is atomic in a trace $T$ if, for every normal or exceptional return action from $m$ performed by a thread $t$ in $T$, no observable action by $t$ intervenes between this return action and the corresponding call action.

Consider for instance methods $m$ and $m'$ with call and return actions $\text{call}_m(t)$, $\text{ret}_m(t)$, etc, performed by thread $t$. Then $m$ is atomic in the traces $\text{call}_m(t)\text{ret}_m(t)\text{call}_{m'}(t)$ and $\text{call}_m(t)\text{call}_{m'}(t')\text{ret}_m(t)$ (with $t' \neq t$) but not in the trace $\text{call}_m(t)\text{call}_{m'}(t)\text{ret}_m(t)$.

Notice that $m$ is atomic in $T$ is equivalent to stating that $m$ does not perform callbacks in $T$.

**Definition 7** Let an API $\mathcal{M}$ be given. The canonicalization of the trace $T$ with respect to $\mathcal{M}$ is the trace $\text{canon}_{\mathcal{M}}(T)$ obtained by moving each normal or exceptional return action from a method $m$ in $\mathcal{M}$ in $T$ right after the corresponding call action.

The following is an immediate consequence of our assumptions on the JLS3 execution model in section 3:

**Proposition 2** Suppose all methods of API $\mathcal{M}$ are atomic in all traces of $P$. If $T$ is a trace of $P$ so is $\text{canon}_{\mathcal{M}}(T)$. □

Proposition 2 presupposes the “order-oblivious” API semantics of section 3, as order-inducing API calls may prevent the shuffling around of return actions needed for the proof.

For inliners that force atomicity of API calls a suitable weakening of the transparency conditions restricts attention to canonic traces in the following way.
**Definition 8** An inliner $I$ is weakly transparent relative to an API $\mathcal{M}$, if for every policy $P_A$, every program $P$, and every trace $T$ of $P$ that adheres to $P_A$, the canonicalization of $T$ equals the canonicalization of some trace of $I(P_A, P)$, i.e.,

$$\text{canon}_{\mathcal{M}}(\omega(\text{exec}_{\mathcal{M}}(P)) \cap P_A) \subseteq \text{canon}_{\mathcal{M}}(\omega(\text{exec}_{\mathcal{M}}(I(P_A, P))))$$

Notice that weak transparency only makes sense for policies that are closed under canonicalization.

### 5.2 Example Inliner

In order to enforce a policy through inlining, it is convenient to be able to statically decide whether a given event 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 any security-relevant method $c.m$, an `invokevirtual` $d.m$ call is bound at run time to method $c.m$ if and only if $d = c$. We deal with the full inheritance problem in earlier work [1].

For simplicity, we also require that the initial values for the security state variables specified by the policy are the default initial values for their corresponding Java types.

The inliner we propose replaces each instruction $L : \text{invokevirtual} c.m$ where $c.m$ is security-relevant by JVML code corresponding to the pseudo code in Fig. 4. The replacement is referred to as a block of inlined code.

The inliner locks the security state and stores arguments to the virtual 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. Before passing control to the API method, the original arguments are restored, and immediately upon return the return value on the operand stack is stored to a local variable. On normal return, after successful completion of the normal return event handler code the security state is unlocked and the inlined code fragment is exited. On exceptional return the exception is instead rethrown.

The two main complications which we had to address when designing this inliner are the possibility of internal exceptions, and the interaction of our locking strategy with API-induced ordering constraints.

The Java Virtual Machine Specification [15] 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 code but it does not have enough memory to store the machine code, it can throw such an internal exception instead of having to terminate the entire program. Whereas internal exceptions are useful for JVM implementors, 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 corresponds to 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 but not strongly conservative, since we exit at a place where the original program does not exit. Below, we prove strong con-
servativity of our inliner under the assumption that the JVM is error-free, i.e. it never throws an internal exception.

The other complication is caused by our choice of locking strategy. Since the program may perform multiple security-relevant calls concurrently, accesses to the security state by the inlined code must be synchronized. We do so by protecting the security state using a lock. There are essentially two ways to do so: acquire the lock for the entire duration of the inlined code (strong synchronization), or acquire it once when processing the before action, release it before performing the security-relevant call, and then acquire it again for processing the after or exceptional action (weak synchronization, analogous to the behavior of the PoET/PSLang inliner [22]). In this paper, we adopted strong synchronization; it has the advantage that both actions associated with a given security-relevant call (i.e. the before action and the after or exceptional action) always occur together, whereas in the case of weak synchronization, the actions from multiple security-relevant calls may be interleaved, leading to a less intuitive policy semantics. A downside of strong synchronization, however, is that it is not applicable in the case where security-relevant methods have synchronization behavior themselves, as discussed above. Indeed, in that case, strong synchronization may introduce deadlocks that did not exist in the original program. Therefore, below, we prove strong conservativity under the assumption that security-relevant methods are non-blocking. Furthermore, strong synchronization is not appropriate when the security-relevant methods include long-running operations that benefit from concurrent execution.

We now proceed to state and prove two correctness theorems for our inliner. The first is general, and applies to both ill-synchronized and well-synchronized programs. The second additionally states weak transparency
for well-synchronized programs.

**Definition 9 (Non-blocking Method)** A method $c.m$ in API $\mathcal{M}$ is non-blocking, if for all programs $P$ and all executions $E \in \text{exec}_\mathcal{M}(P)$ either:

1. $E$ is infinite, or
2. $E$ is terminating, or
3. $E$ is deadlocked with final configuration $C$, and no thread in $C$ is inside $c.m$.

**Theorem 1** Let $\mathcal{I}$ be the inliner of Fig. 4.

1. $\mathcal{I}$ is secure and conservative.
2. For an error-free JVM, and relative to an API for which each srm is non-blocking, the inliner $\mathcal{I}$ is strongly conservative.

**Proof (Sketch)**

We prove only 2 here. The proofs of security and conservativity are similar but easier. Assume an error-free JVM and let $\mathcal{P}_A$ and $P$ be given, and assume that the API is non-blocking with respect to the srms of the policy. Consider an execution $E \in \text{exec}_\mathcal{M}(\mathcal{I}(\mathcal{P}_A, P))$, and let $T = \omega(E)$. There are three cases: Either (1) $E$ is infinite, (2) $E$ is terminating, or (3) $E$ is deadlocked.

(1) We claim it is possible to extract from $E$ another execution $E'$ which replaces each complete execution of an inlined block with the execution of the single invokevirtual instruction for which the block was inserted, and which replaces each partial execution with either nothing or the invokevirtual instruction, depending on whether the instruction concerned is eventually executed in $E$ or not (note that we do not assume
fairness so it is possible for a thread from some point onwards never to be scheduled again). Note that this replacement can be done in parallel, since `SecState.class` locks all accesses to the security state.

To see how this is done let $E$ have the shape $C_0 \cdots C_n \cdots C_m \cdots$ such that $C_i = (h_i, \Lambda_i, \Theta_i)$ for all $i \in \omega$, and such that, for some $tid$, $
abla_0(tid) = (M_n, pc_n, s_n, r_n) : R$, $\nabla_m(tid) = (M_m, pc_m, s_m, r_m) : R$, $pc_n$ points to label $L$ in Fig. 4, $pc_m$ points to label $done$, and $L \leq pc_i \leq done$ for all $i \in [n, m]$. This situation corresponds to the normal, complete execution of the inlined block in 4. Now transform each configuration $C_i$ as follows:

- If $\Lambda_i$(`SecState.class`) is set, unset it.

- Whenever $pc_i$ is less than the pc of the invokevirtual instruction, replace $s_i$ by $s_n$, and otherwise replace $s_i$ by $s_m$.

- Remove all register values inserted by the inliner from all $r_i$.

A similar construction is applied to exceptional, complete executions. Since virtual machine errors are disregarded, only the invokevirtual instruction and the rethrow instruction can raise exceptions. The transformation of exceptional thread configurations is as above, except that the entire frame is replaced, instead of just the operand stack and part of registers. Partial executions are handled in the obvious way. The claim, now, is that the execution thus obtained is an infinite execution of the inlined program with all inlined instructions replaced by noop’s and the exception tables restored accordingly. A further transformation step eliminates the noop’s and restores the exception tables completely, thus obtaining an execution of the original program. It is clear that the execution remains infinite under this transformation as
well. This completes the case.

(2) Assume then that $E$ is terminating. We claim that we can extract an execution $E'$ for the uninlined program which is terminating as well, and such that $T(E) = \text{trunc}_{P_A}(T(E'))$. If $E$ terminates because of a call to System.exit by an inlined block for a call of a security-relevant method $c.m$ with target $o$ and arguments $v$ in a thread $tid$, then this can happen only because either all before guards have been evaluated to false, or all after guards have. The latter cannot happen since the disjunction of the guards is a tautology, and since the guards are evaluated correctly on the call parameters. The former can happen only if the trace $T(E)(tid, c.m, o, v)$ is policy violating. In this case we can eliminate all inlined blocks from $E$, as above, and reroute control flow at the end of (the transformed) $E$ to the invokevirtual instruction, execution of which was prevented by the exception. In this way we obtain a prefix of $E'$ which can be completed to satisfy the requirements of the statement.

(3) The final case is where $E$ is deadlocked. This can only be the case if each live thread in the final configuration, say $C_k$, is waiting at a lock. The lock can be either SecState.class, or another lock set either from a client instruction, or from an API method. In the latter case, the method call is not inlined, since otherwise the method would be non-blocking. If all locks are set from a client instruction or a non-inlined API call then we extract from $E$ an uninlined complete execution with the same trace, as above. Finally, if a thread is waiting at a security state lock then it must be waiting at the initial monitorenter instruction of some inlined block. But that can only be the case if some
other thread is deadlocked inside an inlined block, which is impossible, as it would then be deadlocked inside a non-blocking srn. □

**Lemma 1** Consider a set of methods \( m \in M \). If the methods in \( M \) are non-blocking, then \( M \) is atomic in any trace \( T \) of any program \( P \).

**Proof** By contradiction. Suppose there is a program \( P \) and a trace \( T \) of \( P \) such that some method \( m \in M \) performs a callback in \( T \). Then \( P \) can be modified such that it deadlocks inside the callback. It follows that \( m \) is not non-blocking.

**Theorem 2** Relative to an API for which each srn is non-blocking, \( \mathcal{I} \) is weakly transparent for well-synchronized programs and policies that are closed under canonicalization.

**Proof** Consider a policy \( \mathcal{P}_A \) that is closed under canonicalization, and a well-synchronized program \( P \). Further consider a trace \( T \) of \( P \) that adheres to \( \mathcal{P}_A \). We need to prove that there is a trace of the inlined program \( \mathcal{I}(\mathcal{P}_A, P) \) whose canonicalization equals the canonicalization of \( T \). Since each srn is non-blocking, the srns are atomic in \( T \). Choose an execution \( E \) of \( P \). Then, let \( E' \) be the sequence of configurations obtained by moving each normal or exceptional srn return transition in \( E \) right after the corresponding call transition. Then \( E' \) is an execution of \( P \) and its trace is \( \text{canon}_A(T) \), the canonicalization of \( T \); this is always true because the srns are non-blocking. Now, further transform \( E' \) by inserting the inlined code prolog operations before each SRM call transition, and by inserting the inlined code epilog operations after each SRM return transition. The resulting sequence of configurations \( E'' \) is a legal execution of the inlined program.
\(I(P_A, P)\), because \(P\) is well-synchronized and therefore the extra synchronization has no influence on existing field accesses, and because \(\text{canon}_M(T)\) adheres to \(P_A\). It follows that \(\text{canon}_M(T)\) is a trace of \(I(P_A, P)\). Since \(\text{canon}_M\) is idempotent, \(\text{canon}_M(\text{canon}_M(T))\) equals \(\text{canon}_M(T)\) and we have proven the theorem.

6 Case Studies and Benchmarks

The inlining algorithm described above has been implemented in Java using the ASM framework [17]. We present some results and benchmarks of this inliner in four case studies. All case studies comprise a regular JavaME application and a relevant security policy and are available at url http://www.csc.kth.se/~landreas/inlining.

**ImageExchange** (IE) ImageExchange is a combined server/client application that allows users to exchange images over a Bluetooth connection. The user may choose to act as a server and publish selected images, or as a client and download published images.

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 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 applications are taken from the case studies of the S³MS project.

All policies were successfully enforced by our inliner.

The benchmarks for the case studies are summarized in Tab. 1.

### 6.1 Inlining Overhead

To determine the runtime overhead impact of inlining, a program that invoked an empty dummy SRM 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 iterated $10^6$ times on a Sony-Ericsson W810i. This indicates that the overhead in this experiment was 951 nanoseconds per security-relevant call. This suggests that even program that performs many security-relevant calls can be inlined with a close to negligible performance impact. The sample policy used mentioned the dummy SRM in one BEFORE and one AFTER clause with two guards each.

Note, however, that the above experiment did not measure the performance impact resulting from the loss of parallelism due to the serialization of security-relevant calls. Clearly, this impact is highly dependent on the specific application and its inputs.
7 Conclusions

We have surveyed the security-by-contract paradigm for mobile application security proposed by the EU FP6 project S3MS. A main technical component of this framework is monitoring and monitor inlining, and as the technical contribution of this paper we have discussed inlining correctness criteria suitable for multi-threaded bytecode in the style of Java and .NET, and used the criteria to prove correctness for a concrete inlining algorithm.

The inliner we examine is blocking in the sense that the embedded security state is locked across the security-relevant call, thus preventing concurrent accesses to those methods. This may cause serious performance degradation, in particular for methods involving I/O. Indeed, Erlingsson’s original inliner [22] avoids this problem by unlocking just at the point of executing the call itself. This, however, is sound only for policies that are race-free, in the sense of being insensitive to the sequencing of concurrent actions. In forthcoming work we address this issue and prove correctness of a non-blocking inliner, but for a restricted policy language. In the present setting one can alleviate the problem to some extent by splitting the security state into disjoint components that are locked separately.

A number of extensions of this work merit attention. First, we do not yet address inheritance. This has been considered for the case of sequential Java in [1], and multi-threading is not likely to add significant complications. Security automata as we consider here are allowed to be infinite state. This poses no problems for inlining, and it is very useful to correlate actions as in the IE application considered above. (But, contract-policy matching becomes undecidable, for obvious reasons.) We do not allow the heap to be used in policy guards; whereas this would be useful, allowing it creates significant theoretical and practical problems which merit further
investigation.

An 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 threaded Java is currently under way.

8 Acknowledgements

We acknowledge the members of the $S^3$MS consortium, and reviewers, for many valuable discussions on topics related to security policies, monitoring, and inlining. Special thanks go to Fabio Massacci for his competent leadership of the $S^3$MS consortium, and to our colleagues Gurov and Aktug at KTH and Desmet, Philippaerts and Vanoverberghe at K.U.Leuven.

The work on Dam, Lundblad, and Piessens was partially supported by the $S^3$MS project. Additionally, Dam and Lundblad received support through grants 2003-6108 and 2007-6436 from the Swedish Research Council. 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.

References


[12] Xavier Leroy. Java bytecode verification: Algorithms and formaliza-

Princeton University, June 2006.

[14] Jay Ligatti, Lujo Bauer, and David Walker. Edit automata: enforce-
ment mechanisms for run-time security policies. Int. J. Inf. Sec., 4(1-


[16] Fabio Massacci and Ida Siahaan. Simulating midlet’s security claims
with automata modulo theory. In PLAS, pages 1–9, 2008.

ary 2008.

www.theregister.co.uk/2007/05/23/symbian_signed_spyware,

[19] S3MS. Security of software and services for mobile systems.


9 Tables

<table>
<thead>
<tr>
<th></th>
<th>IE</th>
<th>SN</th>
<th>MJ</th>
<th>BN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Relevant Invokes</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Original Size of Binaries (kb)</td>
<td>39.2</td>
<td>39.6</td>
<td>253.6</td>
<td>234.2</td>
</tr>
<tr>
<td>Inlining Duration (s)</td>
<td>0.56</td>
<td>0.49</td>
<td>1.84</td>
<td>1.42</td>
</tr>
<tr>
<td>Size increase due to Inlining (%)</td>
<td>1.0</td>
<td>11.0</td>
<td>0.2</td>
<td>1.2</td>
</tr>
</tbody>
</table>

Table 1: Benchmarks for the case studies. Inlining was performed with an Intel Core 2 CPU at 1.83 GHz with 2 Gb memory.
10 Figure captions

Fig. 1: A ConSpec contract for the chess game.
Fig. 2: An example device policy.
Fig. 3: Transparency counterexample.
Fig. 4: The inlining replacement of $L$: `invokevirtual c.m`. 
11 Figures
SECURITY STATE
  int bytesSent = 0;
  int smsSent = 0;
PERFORM
  array.Length == 20 && bytesSent + array.Length <= 2000 ->
AFTER int sent = System.Net.Sockets.Socket.Send(byte[] array)
PERFORM
  true -> bytesSent += sent;
BEFORE Microsoft.WindowsMobile.PocketOutlook.SmsMessage.Send()
PERFORM
  smsSent <= 100 ->
AFTER Microsoft.WindowsMobile.PocketOutlook.SmsMessage.Send()
PERFORM
  true -> smsSent += 1;

Fig. 1:
SECURITY STATE
   int bytesSent = 0;
PERFORM
   bytesSent + array.Length <= 10000 ->
AFTER int sent = System.Net.Sockets.Socket.Send(byte[] array)
PERFORM
   true -> bytesSent += sent;

Fig. 2:
Thread1: beforeA = 1; || Thread2: r1 = afterA;
sra(); // A || sra(); // B
afterA = 1; || r2 = beforeA;
|| if (r1 == 1 && r2 == 0) {sra();}

Fig. 3:
\[ L: \text{ldc SecState} \]
\[ \text{monitorenter} \]
\[ \text{astore 0} \]
\[ \vdots \]
\[ \text{astore n} - 1 \]
\[ \text{beforeG}_{1}: \text{[eval before } G_1\text{]} \]
\[ \text{ifeq beforeG}_2 \]
\[ \text{[before update 1]} \]
\[ \text{goto beforeEnd} \]
\[ \vdots \]
\[ \text{beforeG}_{i}: \text{[eval before } G_i\text{]} \]
\[ \text{ifeq exit} \]
\[ \text{[before update } i\text{]} \]
\[ \text{beforeEnd}: \text{aload } n - 1 \]
\[ \vdots \]
\[ \text{aload } 0 \]
\[ \text{invoke: invokevirtual } c.m \]
\[ \text{invokeDone}: \text{astore } n \]
\[ \text{afterG}_{1}: \text{[eval after } G_1\text{]} \]
\[ \text{ifeq afterG}_2 \]
\[ \text{[after update 1]} \]
\[ \text{goto afterEnd} \]
\[ \vdots \]
\[ \text{afterG}_{j}: \text{[eval after } G_j\text{]} \]
\[ \text{ifeq exit} \]
\[ \text{[after update } j\text{]} \]
\[ \text{afterEnd}: \text{aload } n \]
\[ \text{ldc SecState} \]
\[ \text{monitorexit} \]
\[ \text{afterReleased}: \text{goto done} \]

Extra entries in exception handler array:

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Target</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>invoke</td>
<td>invokeDone</td>
<td>exceptionalG_1</td>
<td>any</td>
</tr>
<tr>
<td>L</td>
<td>exceptionalReleased</td>
<td>exit</td>
<td>any</td>
</tr>
<tr>
<td>exit</td>
<td>done</td>
<td>exit</td>
<td>any</td>
</tr>
</tbody>
</table>

Fig. 4:
Appendix C

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

Mads Dam, Andreas Lundblad
Royal Institute of Technology, KTH

December 15, 2010

Abstract

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

1 Introduction

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

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

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

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

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

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

The states of the two monitors are compared statically through a monitor invariant, expressing that the state of the embedded monitor is in synchrony with that of the ghost monitor. This monitor invariant is then inserted as an assertion at each security relevant method call. The assertions for the remaining program points could then in principle be computed using a weakest pre-condition (WP) calculus. Unfortunately, there is no guarantee that such an approach would be feasible. However, it turns out that it is sufficient to perform the WP computations for the inlined code snippets and not for the client code, under some critical assumptions:

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

The last constraint can be handled, in particular, by implementing the embedded monitor state as a static member of a final security state class. The important consequence is that instructions that do not appear in the inlined snippets, and do not include putstatic instructions to the security state field, may be annotated with the monitor invariant to obtain a fully annotated program. This means, in particular, that a simple syntactic check is sufficient to eliminate costly WP checks in almost all cases and allows a very open-ended treatment of the JVM instruction set.

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

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

1.1 Related Work

Our approach adopts the Security-by-Contract (SxC) paradigm (cf. [7, 25, 12, 20, 8]) which has been explored and developed mainly within the S3MS project [28].

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

1
Erlingsson and Schneider [14] represents security automata directly as Java code snippets, making the resulting code difficult to reason about. The ConSpec contract specification language used here is for tractability restricted to API calls and (normal or exceptional) returns, and uses an independent expression syntax. This corresponds roughly to the call/return fragment of PSLang which includes all policies expressible using Java stack inspection [15].

Edit automata [22, 23] are examples of security automata that go beyond pure monitoring, as truncations of the event stream, to allow also event insertions, for instance to recover gracefully from policy violations. This approach has been fully implemented for Java by J. Ligatti et al. in the Polymer tool [5] which is closely related to Naccio [16] and PoET/PSLang [14].

Certified reference monitors has been explored by a number of authors, mainly through type systems, e.g. in [31, 6, 35, 18, 11], but more recently also through model checking and abstract interpretation [33, 32]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al. [18]. The Mobile system uses a simple library extension to Java bytecode to help managing updates to the security state. The use of linear types allows a type system to localize security-relevant actions to objects that have been suitably unpacked, and the type system can then use this property to check for policy compliance. Mobile enforces per-object policies, whereas the policies enforced in our work (as in most work on IRM enforcement) are per session. Since Mobile leaves security state tests and updates as primitives, it is quite likely that Mobile could be adapted, at least to some forms of per session policies. On the other hand, to handle per-object policies our approach would need to be extended to track object references. Finally, it is worth noting that Mobile relies on a specific inlining strategy, whereas our approach, as mentioned in the previous section, is less sensitive to this.

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

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

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

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

2 Program Model

We assume that the reader is familiar with Java bytecode syntax and the Java Virtual Machine (JVM). Here, we only present components of the JVM, that are essential for the definitions in the rest of the text. Much of this is standard and can be skipped in a first reading. A few simplifications have been made in the presentation. In particular we disregard static initializers, and to ease notation a little we ignore issues concerning overloading. We use $c$ for (fully qualified) class names, $m$ for method names, and $f$ for field names. Types are either primitive or object types, i.e. classes, or arrays. Class declarations induce a class hierarchy, denoted by $\prec$. If $c$ defines $m$ (declares $f$) explicitly, then $c$ defines (declares) $c.m$ ($c.f$). Otherwise, $c$ defines $c.m$ (declares $c.f$) if $c$ is the smallest superclass of $c'$ that contains an explicit definition (declaration) of $c.m$ ($c.f$). Single inheritance ensures that definitions/declarations are unique, if they exist.

We let $v$ range over the set of all values of all types. Values of object type are (typed) locations, mapped to objects, or arrays, by a heap $h$. The typing assertion $h \vdash v : c$ asserts that $v$ is some location $\ell$, and that in the typed heap $h$, $\ell$ is defined and of type $c$, and similarly for arrays. Typing preserves the sub-class relation, in the sense that if $h \vdash v : c$ and $c \prec c'$ then $h \vdash v : c'$ as well. For objects, it suffices to assume that if $h \vdash v : c$ then the object $h(v)$ determines a field $h(v).f$ (method $h(v).m$) whenever $f$ (m) is declared (defined) in $c$. Static fields are identified with field references of the form $c.f$. To handle those, heaps are extended to assignments of values to field references.

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

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

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

An execution $E$ of a program $P$ is a (possibly infinite) sequence of JVM configurations $C_0 C_1 \ldots$ where $C_0$ is an initial configuration consisting of a single, normal activation record with an empty stack, no local variables, $M$ as a reference to the main method of $P$, $pc = 0$ and for each $i \geq 0$, $C_i \rightarrow_{\text{JVM}} C_{i+1}$.
We restrict attention to configurations that are type safe, in the sense that heap contents match the types of corresponding locations, and that arguments and return/exceptional values for primitive operations as well as method invocations match their prescribed types. The Java bytecode verifier serves, among other things, to ensure that type safety is preserved under machine transitions.

The only non-standard aspect of $\rightarrow_{JVM}$ is the treatment of API methods. We assume a fixed API for which we have access only to the signature (types), but not the implementation, of its methods. We therefore treat API method calls as atomic instructions with a non-deterministic semantics. In this sense, we do not practice complete mediation [29]. When an API method is called either the pc is incremented and arguments popped from the operand stack and replaced by an arbitrary return value of appropriate type, or else an arbitrary exceptional activation record is returned. Similarly, the return configurations for API method invocations contain an arbitrary heap, since we do not know how API method bodies change heap contents. Our approach hinges on our ability to recognize API calls. This property is destroyed by the reflect API, which is consequently not considered.

3 Assertions

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

$$ e ::= v \mid e.f \mid c.f \mid s_i \mid l_i \mid e \circ e \mid a \rightarrow e.e \mid (e,e) \mid \perp $$

$$ a ::= tt \mid ff \mid e \triangleright e \mid a \land a \mid \neg a \mid e : c $$

where $i \in \omega$. The semantics, as mappings $\ll e \gg C$ and $\ll a \gg C$ is given in Figure 2. The operations $\circ$ and $\triangleright$ are generic binary operators/relation symbols, respectively, with Kleene equality. The expression $s_i$ refers to the $i$'th element of the operand stack, and $l_i$ refers to the $i$'th local variable; the expression $a \rightarrow e_1 | e_2$ is a conditional, $\langle e_1, e_2 \rangle$ is pairing and $tt$ and $ff$ represent true and false respectively; a heap

| $\ll e.f \rr(h,R) = h(\ll e \rr(h,R)).f$ |
| $\ll e.f \rr(h,R) = h(c.f)$ |
| $\ll s_i \rr(h, (M, pc, s, r) : R) = s_i$ |
| $\ll l_i \rr(h, (M, pc, s, r) : R) = l_i$ |
| $\ll e_1 \circ e_2 \rr C = \ll e_1 \rr C \circ \ll e_2 \rr C$ |
| $\ll e_1 \triangleright e_2 | e_3 \rr C = \begin{cases} \ll e_2 \rr C, & \ll e_1 \rr C = tt \\ \ll e_3 \rr C, & \text{otherwise} \end{cases}$ |
| $\ll (e_1, e_2) \rr C = \ll (\ll e_1 \rr C, \ll e_2 \rr C) \rr C$ |
| $\ll (e_1, e_2) \rr C = \ll (\ll e_1 \rr C, \ll e_2 \rr C) \rr C$ |
| $\ll a \rr C = \ll (\ll a \rr C) \rr C$ |
| $\ll e : c \rr(h,R) = \begin{cases} tt & \text{if } h \vdash \ll e \rr(h,R) : c \\ ff & \text{otherwise} \end{cases}$ |

Figure 2: Semantics of expressions and assertions

assertion is an assertion that does not reference the stack, or any of the local variables. Disjunction ($\lor$) and implication ($\Rightarrow$) are defined as usual. We let $\text{if}(a_0, a_1, a_2)$ denote the conditional expression $\langle a_0 \Rightarrow a_1 \rangle \land \langle \neg a_0 \Rightarrow a_2 \rangle$ and $\text{select}(a_1, a_2, a_{else} \ldots)$ the generalized conditional expression $\text{if}(a_1, 0, a_{else,1}, \ldots, \text{if}(a_{1,n}, a_{2,n}, a_{else} \ldots) \ldots))$.

4 Extended Method Definitions

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

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

For extended programs, the notions of transition and execution are not affected by the presence of assertions. An extended program is valid, if all anno-
The WP-calculus used in the proof generation/recognition is given in Table 1. The definition uses the auxiliary functions \( \text{shift} \) and \( \text{unshift} \) which increments, resp. decrements, each stack index by one and \( \text{def}(c.m) \) which denotes the set of all classes \( c' \) such that \( c < c' \) and \( c' \) defines \( m \). The account of dynamic call resolution in Table 1 is crude, but the details are unimportant since, in this paper, pre- and post-conditions are always identical and common to all methods.

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

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

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

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

We note that local validity implies validity, as expected.

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

**Proof.** Follows by induction on the length of the execution. For details we refer to the Coq formalization [24]. \( \square \)

## 5 Security Specifications

We consider security specifications written in a policy specification language ConSpec [2], similar to PSlang [14], but more constrained, to be amenable to analysis. An example specification is given in Figure 3. The syntax is intended to be largely self-explanatory: The specification in Figure 3 states that the program can only send files using the Bluetooth Obex protocol upon direct request by the user. No exception may arise during evaluation of the user query.

A ConSpec specification tells when and with what arguments an API method may be invoked. If the specification has one or more constraints on a method, the method is security relevant. In the example there are two security relevant methods, \texttt{GUI.fileSendQuery} and \texttt{Bluetooth.obexSend}. The specification expresses constraints in terms of \texttt{BEFORE}, \texttt{AFTER} and \texttt{EXCEPTIONAL} clauses. Each clause is a guarded command where the guards are side-effect
SECURITY STATE String lastApproved = "";
AFTER file = GUI.fileSendQuery()
PERFORM true -> { lastApproved = file; }

EXCEPTIONAL GUI.fileSendQuery()
PERFORM

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

Figure 3: A security specification example written in ConSpec.

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

5.1 Security Automata

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

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

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

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

\[
SRT(E) = SRT(E, \epsilon) \\
SRT(\epsilon, \epsilon) = \epsilon \\
SRT((c.m, v)^\dagger SRT(E, v :: \gamma) \\
if C = (h, (c.m, pc, v :: s, l) :: R) is calling c.m' \\
(c.m, v, r)^\ddagger SRT(E, \gamma') \\
if C = (h, (c.m, pc, r :: s, l) :: R) is returning and \gamma = v :: \gamma' \\
(c.m, v)^\& SRT(E, \gamma') \\
if C = (h, (a) :: R) is returning exceptionally and \gamma = v :: \gamma' \\
SRT((c.m, v)^\& SRT(E, \gamma) otherwise
\]

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

Definition 5 (Contract Adherence). The program \(P\) adheres to contract \(C\) if for all executions \(E\) of \(P\), \(SRT(E) \in C\).
6 Example Inliner

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

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

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

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

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

7 The Ghost Monitor

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

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

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

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

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

**Proof.** Follows by induction on the length of the execution. For details we refer to the Coq formalization [24].
8 Contract Adherence Proofs

The key idea of a contract adherence proof is to show that the embedded monitor state $\text{ms}$ of the program $\mathcal{I}^9(P, C)$ and the ghost monitor state $\text{ms}^g$ are in agreement at certain program points. These points certainly need to include all potentially security relevant call and return sites. But, since we aim for a procedural analysis, and to cater for virtual call resolution actually all call and return sites are included.

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

**Definition 6 (Adherence Proof).** An adherence proof for program $P$ and contract $C$ assigns to each method $M = (I, H)$ in $\mathcal{I}^9(P, C)$ an assertion array $A$ such that the extended method $(I, H, A, \text{ms} = \text{ms}^g, \text{ms} = \text{ms}^g)$ is locally valid.

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

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

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

**Proof.** Assume $\Pi$ is an adherence proof for a program $P$ and a contract $C$. By theorem 1 we know that the corresponding extended program for $\mathcal{I}^9(P, C)$ is globally valid. This implies that $\text{ms} = \text{ms}^g$ at each configuration that is calling (or returning from) a security relevant configuration. Furthermore, since the $\bot$ value is an artificial "error" value of the security automaton with no Java counterpart, we know that if $\text{ms} = \text{ms}^g$, then $\text{ms}^g \neq \bot$. Thus, by lemma 1, $\text{SRT}(E) \in C$ and therefore $P$ adheres to $C$. 

8.1 Example Proof Generation

The process of generating contract adherence proofs is closely related to the process of embedding the ref-
erence monitor, thus the inlining and proof generation is preferrably done by the same agent. This section describes how proofs may be generated for code produced by the example inliner presented in Section 6.

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

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

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

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

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

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

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

**SCOPE Session**

```java
SECURITY STATE boolean haveRead = false;

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

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

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

### 8.2 Proof Recognition

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

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

**Proof.** To verify the validity of a given adherence proof we look at the requirements of definition 6. Verifying that the pre- and post-conditions equal the monitor invariant is a simple syntactic check and can be done in time linearly proportional to the number of methods in \( P \).
The verification condition can then be rewritten and simplified by iterated applications of the rule $x = y \Rightarrow \phi \rightarrow \phi[z/x][z/y]$ where $x$ and $y$ are instantiated with real variables and ghost counterparts respectively and where $z$ does not occur in $\phi$. These rewrites can be performed in time proportional to the length of the formula and does not increase the size of the expression since $x$, $y$ and $z$ are atomic. The result can then be rewritten to $tt$ using the rules $(\psi \Rightarrow \phi) \land (\neg \psi \Rightarrow \phi) \rightarrow \phi$ and $\phi \Rightarrow \phi \rightarrow tt$ in time polynomial in the size of the formula.

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

9 Implementation and Evaluation

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

- **MobileJam**: A GPS based traffic jam reporter which utilizes the Yahoo! Maps API. Policy: Only connect to http://local.yahooapis.com.
- **Snake**: A classic game of snake in which the player may submit current score to a server.
Table 2: Benchmarks for the two case studies.

<table>
<thead>
<tr>
<th></th>
<th>MobileJam</th>
<th>Snake</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security Relevant Invokes</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Original Size</td>
<td>428.0 kb</td>
<td>43.7 kb</td>
</tr>
<tr>
<td>Size increase for IRM</td>
<td>4.8 kb</td>
<td>1.1 kb</td>
</tr>
<tr>
<td>Size increase for Proofs</td>
<td>20.6 kb</td>
<td>2.6 kb</td>
</tr>
<tr>
<td>Inlining</td>
<td>10.1 s</td>
<td>8.6 s</td>
</tr>
<tr>
<td>Proof Generation</td>
<td>4.7 s</td>
<td>0.8 s</td>
</tr>
<tr>
<td>Proof Recognition</td>
<td>98 ms</td>
<td>117 ms</td>
</tr>
</tbody>
</table>

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

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

10 Conclusions

We have demonstrated the feasibility of a proof-carrying approach to certified monitor inlining at the level of practical Java bytecode, including exceptions and inheritance. This answers partially a question raised by K. W. Hamlen et al. [18].

We have proved correctness of our approach in the sense of soundness: Contract adherence proofs are sufficient to ensure compliance. This soundness proof has been formalized [24] in Coq. We also obtain partial completeness results, namely that proofs for inlined programs can always be generated, and such proofs are guaranteed to be recognized at program loading time, at least when contracts do not use equational tests that are too difficult. Other properties are also interesting such as transparency [29], roughly, that all adherent behaviour is preserved by the inliner. This type of property is, however, more relevant for the specific inliner, and not so much for the certification mechanism, and consequently not addressed here (but see e.g. [23, 34, 10, 9] for results in this direction).

The approach is efficient: Proofs are small and recognised easily, by a simple proof checker. An interesting feature of our approach is that detailed modelling of bytecode instructions is needed only for instructions appearing in the inlined code snippets. For other instructions a simple conditional invariance property on static fields of final objects suffices. This means, in particular, that our approach adapts to new versions of the Java virtual machine very easily, needing only a check that the static field invariance is maintained. Worth pointing out also is that the enforcement architecture can be realized in a way which is backwards compatible, in the sense that PCC-aware client programs can be executed without modification in a PCC-unaware host environment.

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

References


A Implementation of the Example Inliner

Our inliner lets the state of the embedded security monitor be represented by a static field ms of a final security state class, named to avoid clashes with classes in the target program. This choice of representation relies on the following fact of JVM execution and allows for our open-ended treatment of large parts of the instruction set.

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

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

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

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

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

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

The remaining invoke instructions (invokestatic,
B Proof of Lemma 2

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

SCOPE Session

SECURITY STATE DECLARATION

BEFORE \( c.m(\text{int a}) \) PERFORM \( \{\text{bef}_g\} \) -> {\(\text{bef}_g\)}

AFTER \( r = c.m(\text{int a}) \) PERFORM \( \{\text{aft}_g\} \) -> {\(\text{aft}_g\)}

EXCEPTIONAL \( c.m(\text{int a}) \) PERFORM \( \{\text{exc}_g\} \) -> {\(\text{exc}_g\)}

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

\[
ms = ms^8
\]

NON-INLINED INSTRUCTION

// INLINED CODE START

ms = ms^8
ASTORE a
ASTORE t
ALOAD t
ALOAD a

// BEFORE

26: \( \text{IF}(t = A_{38}, A_{30}) \)
INSTANCEOF c
IFEQ 30

28: \( \text{IF}(\text{ex}_a, \text{IF}(s_1 = c, \text{IF}(\text{bf}_g, ms \cdot \sigma_{bf}(a) = ms^8 \cdot \sigma_{bf}(s_0), \text{ms} \cdot \sigma_{bf} = \bot)), \text{ms} = ms^8) \cdot a = s_0 \land t = s_1, t_1) \)
[EVALUATE \( \text{bf}_g \)]
IFEQ 29
[PERFORM \( \text{bf}_g \)]
GOTO 30

29: \( \text{IF}(\text{ex}_a, \text{IF}(s_1 = c, \text{IF}(\text{bf}_g, ms = ms^8 \cdot \sigma_{bf}(s_0), \text{ms} = \bot), ms = ms^8) \land a = s_0 \land t = s_1, t_1) \)
[EVALUATE \( \text{bf}_g \)]
IFEQ 30

32: \( \text{ms} = ms^8 \land a = s_0 \land t = t^9 \)
\( \{r^9 := s_0\} \)
\( \{ms^9 := t^9 : c \rightarrow \delta(ms^8,(c,m,a^9,t^9)) \mid ms^9\} \)
\( A_{43}(r/s_0) \)
ASTORE r
ALOAD t
GOTO 43

// EXCEPTIONAL

34: \( \{t = c, A_{40}, A_{12}\} \)
INSTANCEOF c
IFEQ 42

40: \( \text{IF}(\text{exc}_a, ms \cdot \sigma_{exc}(a) = ms^8, A_{41}) \)
[EVALUATE \( \text{exc}_a \)]
IFEQ 41
[PERFORM \( \text{exc}_a \)]
GOTO 42

41: \( tt \)
[INSTANCETO System.exit]

42: \( ms = ms^8 \)
ATHROW

// AFTER

43: \( \{t = c, A_{44}, A_{46}\} \)
INSTANCEOF c
IFEQ 46

44: \( \text{IF}(\text{aft}_a, ms \cdot \sigma_{aft}(r,n) = ms^8, tt) \)
[EVALUATE \( \text{aft}_a \)]
IFEQ 45
[PERFORM \( \text{aft}_a \)]
GOTO 46

45: \( tt \)
[INSTANCETO System.exit]

46: \( ms = ms^8 \)
NON-INLINED INSTRUCTION

Figure 10: Schematic annotation for contract displayed Figure 9