Journal of Computer Security 18 (2010) 37–59
DOI 10.3233/JCS-2010-0365
IOS Press

Provably correct inline monitoring for multithreaded Java-like programs

Mads Dam [a], Bart Jacobs [b], Andreas Lundblad [c] and Frank Piessens [d,*]

[a] *ACCESS Linnaeus Centre, Royal Institute of Technology (KTH), Sweden*
E-mail: mfd@kth.se; Tel.: +46 8 790 6229
[b] *Katholieke Universiteit Leuven, Belgium*
E-mail: bart.jacobs@cs.kuleuven.be; Tel.: +32 16 32 7825
[c] *School of Computer Science and Communication, Royal Institute of Technology (KTH), Sweden*
E-mail: landreas@kth.se; Tel.: +46 8 790 8408
[d] *Katholieke Universiteit Leuven, Belgium*
E-mail: frank.piessens@cs.kuleuven.be; Tel.: +32 16 32 7603

Inline reference monitoring is a powerful technique to enforce security policies on untrusted programs. The security-by-contract paradigm proposed by the EU FP6 S^{3}MS 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.

*Corresponding author.

The Security of Software and Services for Mobile Systems (S^3MS) 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^3MS 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 embedded 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 [21]. The PoET/PSLang toolset by Erlingsson [9] 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 [3,14,15], 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. [4,10,22,24]. Directly related to the work reported here is the type-based Mobile system due to Hamlen et al. [12]. 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 [14], 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 S^{3}MS 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 S^{3}MS project [20] 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 [5,7].

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 S^{3}MS 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 S^3MS 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 [19] 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 [21], 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 S^{3}MS 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 [17], the process of checking whether the security 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 S^{3}MS 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 S^{3}MS project), we consider a two-player chess game running on the .NET Compact Framework. This

```
SECURITY STATE
  int bytesSent = 0;
  int smsSent = 0;
BEFORE System.Net.Sockets.Socket.Send(byte[] array)
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. A ConSpec contract for the chess game.

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 (SMSs). 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

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

Fig. 2. An example device policy.

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 kilobytes. 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 [11]. 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 ι. The exception handler array H is a partial map from integer indices to exception handlers. An exception handler (b, e, t, c) catches exceptions of type c and its subtypes raised by instructions in the range $[b, e)$ and transfers control to address t, if it is the topmost handler that covers the instruction for this exception type.

Configurations and transitions

A *configuration* $C = (h, \Lambda, \Theta)$ of the JVM consists of a *heap* h, a *lock map* Λ, and a *thread configuration map* Θ which maps a thread identifier *tid* to its thread configuration $\Theta(\mathit{tid}) = \theta$. A thread configuration θ 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* $s \in \mathit{Val}^*$ 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 $\theta = (M, \mathit{pc}, s, r) :: R$ is *calling*, if $M[\mathit{pc}]$ is an invoke instruction, and it is *returning normally*, if $M[\mathit{pc}]$ 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 θ is *returning exceptionally* if θ is exceptional, and if $(h, \Lambda, \theta) \rightarrow_{\mathit{tid}} (h', \Lambda', \theta')$ implies that θ' is exceptional as well. I.e. the normal frame immediately succeeding the top exceptional frame in θ is popped in θ', if θ' 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 $C_0 C_1 \ldots$ 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 $i \geqslant 0$, $C_i \rightarrow_{\text{JVM}} C_{i+1}$. We restrict attention to configurations that are *type safe*, in the sense that heap contents match the types of corresponding locations, and that arguments and return/exceptional values

for primitive operations as well as method invocations match their prescribed types. The Java bytecode verifier serves, among other things, to ensure that type safety is preserved under machine transitions (cf. [13]).

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 $\rightarrow_{\mathrm{JVM}}$ 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 $\rightarrow_{\mathrm{JVM}}$ 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 $\rightarrow_{\mathrm{JVM}}$ 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:

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

where a transition from C to C' performs an observable action α, 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 $d \notin \mathcal{M}$ 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 $C \xrightarrow{(tid,c.m,o,v)^{\uparrow}}_{\text{JVM}} C'$, and a normal return from this call with return value r as $C'' \xrightarrow{(tid,c.m,o,v,r)^{\downarrow}}_{\text{JVM}} C'''$. We represent an exceptional return from this call with exception object t as $C'' \xrightarrow{(tid,c.m,o,v,t)^{\Downarrow}}_{\text{JVM}} C'''$. 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 $(tid, c.m, o, v)^{\uparrow}$, $(tid, c.m, o, v, r)^{\downarrow}$, and $(tid, c.m, o, v, t)^{\Downarrow}$ as before actions, after actions, and exceptional actions, respectively, and we collect them in sets $\Omega^{\uparrow}$, $\Omega^{\downarrow}$, and $\Omega^{\Downarrow}$.

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

$$\mathcal{T}(P) = \{\omega(E) \mid E \in exec_{\mathcal{M}}(P)\}.$$

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 $\mathcal{T}_{\text{c}}(P)$ 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 [21]. In this paper we view a *security*

automaton as an automaton $\mathcal{A} = (Q, \delta, q_0)$ where Q is a countable (not necessarily finite) set of states, $q_0 \in Q$ is the initial state, and $\delta : Q \times \mathbf{\Omega} \rightharpoonup Q$ is a (partial) transition function, where $\mathbf{\Omega} = \mathbf{\Omega}^{\uparrow} \cup \mathbf{\Omega}^{\downarrow} \cup \mathbf{\Omega}^{\Downarrow}$ is the set of observable actions. All states $q \in Q$ are viewed as accepting.

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

A security automaton can be derived from a ConSpec policy in the obvious manner. We refer to [1] for details. We assume 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 $\mathcal{P}_{\mathcal{A}}$, if for all executions E of P, $\omega(E) \in \mathcal{P}_{\mathcal{A}}$.

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 $\mathcal{I}$ which for each policy $\mathcal{P}_{\mathcal{A}}$ and program P produces an inlined program $\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)$. 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. [14] 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 $\mathcal{I}$ is:

- *Secure* if, for every program P, every trace of the inlined program $\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)$ adheres to $\mathcal{P}_{\mathcal{A}}$, i.e. $\mathcal{T}(\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)) \subseteq \mathcal{P}_{\mathcal{A}}$.

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

Fig. 3. Transparency counterexample.

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

Recall from Section 3 that the set of traces $\mathcal{T}(P)$ 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 `beforeA` and `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 `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 `r1` gets the value 1 and `r2` gets the value 0 is a legal execution. As a result, the program has a trace with three `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* $\mathit{trunc}_{\mathcal{P}_\mathcal{A}}(T)$ of a trace T under a policy $\mathcal{P}_\mathcal{A}$ is the greatest prefix T' of T that adheres to $\mathcal{P}_\mathcal{A}$.

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

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

$$\mathcal{T}_c(\mathcal{I}(\mathcal{P}_\mathcal{A}, P)) \subseteq \mathit{trunc}_{\mathcal{P}_\mathcal{A}}(\mathcal{T}_c(P)).$$

Example 1. An abstract version of the program in Fig. 3 might have traces AB, BA, ABC and BAC, all complete and all in $\mathcal{P}_\mathcal{A}$. Suppose the set of complete traces of $\mathcal{I}(\mathcal{P}_\mathcal{A}, P)$ is $\{AB, BA\}$. The inliner $\mathcal{I}$ is strongly conservative (for this particular program), but not transparent. As another example suppose P' has traces A, AB, AC, ABC such that $A, AB \in \mathcal{P}_\mathcal{A}$, $AC, ABC \notin \mathcal{P}_\mathcal{A}$, and AC, ABC, but not A, AB, are complete. Suppose the only trace of $\mathcal{I}(\mathcal{P}_\mathcal{A}, P')$ 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 $T \in \mathcal{T}(\mathcal{I}(\mathcal{P}_\mathcal{A}, P))$. Then we find $T' \geqslant T$ such that $T' \in \mathcal{P}_\mathcal{A}$, by strong conservativity, so also $T \in \mathcal{P}_\mathcal{A}$, by prefix closure. For conservativity the argument is similar. □

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 m. 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 $\mathit{exec}^c_{\mathcal{M}}(P)$ be the set of complete executions of P in the API $\mathcal{M}$. An inliner $\mathcal{I}$ is *strongly conservative relative to the API* $\mathcal{M}$, if for each policy $\mathcal{P}_{\mathcal{A}}$ and each program P,

$$\omega(\mathit{exec}^c_{\mathcal{M}}(\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P))) \subseteq \mathit{trunc}_{\mathcal{P}_{\mathcal{A}}}(\omega(\mathit{exec}^c_{\mathcal{M}}(P))).$$

An implication of this definition is that, if in some execution E in some API the inliner kicks in and blocks an sra α, then there will be an execution of the uninlined program which after the trace of E executes α. The condition does not guarantee, however, that E without the inliner next would have performed α. 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 $\mathit{call}_m(t)$, $\mathit{ret}_m(t)$, etc, performed by thread t. Then m is atomic in the traces $\mathit{call}_m(t)$ $\mathit{ret}_m(t)\mathit{call}_{m'}(t)$ and $\mathit{call}_m(t)\mathit{call}_{m'}(t')\mathit{ret}_m(t)$ (with $t' \neq t$) but not in the trace $\mathit{call}_m(t)\mathit{call}_{m'}(t)\mathit{ret}_m(t)$. 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 $\mathit{canon}_{\mathcal{M}}(T)$ 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 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 $\mathcal{I}$ is *weakly transparent* relative to an API $\mathcal{M}$, if for every policy $\mathcal{P}_{\mathcal{A}}$, every program P, and every trace T of P that adheres to $\mathcal{P}_{\mathcal{A}}$, the canonicalization of T equals the canonicalization of some trace of $\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)$, i.e.

$$\mathit{canon}_{\mathcal{M}}(\omega(\mathit{exec}_{\mathcal{M}}(P)) \cap \mathcal{P}_{\mathcal{A}}) \subseteq \mathit{canon}_{\mathcal{M}}(\omega(\mathit{exec}_{\mathcal{M}}(\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)))).$$

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: `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.

```
L: ldc SecState
   monitorenter
   astore 0
   ⋮
   astore n − 1

beforeG1: [eval before G1]
          ifeq beforeG2
          [before update 1]
          goto beforeEnd
   ⋮
beforeGi: [eval before Gi]
          ifeq exit
          [before update i]

beforeEnd: aload n − 1
   ⋮
          aload 0

invoke: invokevirtual c.m

invokeDone: astore n

afterG1: [eval after G1]
         ifeq afterG2
         [after update 1]
         goto afterEnd
   ⋮
afterGj: [eval after Gj]
         ifeq exit
         [after update j]

afterEnd: aload n

          ldc SecState
          monitorexit

afterReleased: goto done

exceptionalG1: [eval exceptional G1]
               ifeq exceptionalG2
               [exceptional update 1]
               goto exceptionalEnd
   ⋮
exceptinoalGk: [eval exceptional Gk]
               ifeq exit
               [exceptional update k]

exceptionalEnd: ldc SecState
                monitorexit

exceptionalReleased: athrow

exit: iconst −1
      invokestatic System.exit

done:
```

Extra entries in exception handler array:

From	*To*	*Target*	*Type*
invoke	*invokeDone*	*exceptionalG1*	*any*
L	*exceptionalReleased*	*exit*	*any*
exit	*done*	*exit*	*any*

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

The Java Virtual Machine Specification [16] 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 conservativity 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 [9]). 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 $E \in \mathit{exec}_{\mathcal{M}}(P)$ 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 $\mathcal{P}_\mathcal{A}$ and P be given, and assume that the API is non-blocking with respect to the srms of the policy. Consider an execution $E \in \mathit{exec}_\mathcal{M}(\mathcal{I}(\mathcal{P}_\mathcal{A}, P))$, 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*, $\Theta_n(\mathit{tid}) = (M_n, \mathit{pc}_n, s_n, r_n) :: R$, $\Theta_m(\mathit{tid}) = (M_n, \mathit{pc}_m, s_m, r_m) :: R$, pc_n points to label L in Fig. 4, pc_m points to label *done*, and $L \leqslant \mathit{pc}_i \leqslant \mathit{done}$ 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 Λ_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 $\mathcal{T}(E) = \mathit{trunc}_{\mathcal{P}_\mathcal{A}}(\mathcal{T}(E'))$. 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 $\mathcal{T}(E)(\mathit{tid}, c.m, o, v)$ 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 srm. □

□

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 srm is non-blocking, $\mathcal{I}$ is weakly transparent for well-synchronized programs and policies that are closed under canonicalization.*

Proof. Consider a policy $\mathcal{P}_\mathcal{A}$ that is closed under canonicalization, and a well-synchronized program P. Further consider a trace T of P that adheres to $\mathcal{P}_\mathcal{A}$. We need to prove that there is a trace of the inlined program $\mathcal{I}(\mathcal{P}_\mathcal{A}, P)$ whose canonicalization equals the canonicalization of T. Since each srm is non-blocking, the srms 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 srm return transition in E right after the corresponding call transition. Then E' is an execution of P and its trace is $\mathit{canon}_{\mathcal{M}}(T)$, the canonicalization of T; this is always true because the srms 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 $\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)$, because P is well-synchronized and therefore the extra synchronization has no influence on existing field accesses, and because $\mathit{canon}_{\mathcal{M}}(T)$ adheres to $\mathcal{P}_{\mathcal{A}}$. It follows that $\mathit{canon}_{\mathcal{M}}(T)$ is a trace of $\mathcal{I}(\mathcal{P}_{\mathcal{A}}, P)$. Since $\mathit{canon}_{\mathcal{M}}$ is idempotent, $\mathit{canon}_{\mathcal{M}}(\mathit{canon}_{\mathcal{M}}(T))$ equals $\mathit{canon}_{\mathcal{M}}(T)$ 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 [18]. 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 S^{3}MS project. All policies were successfully enforced by our inliner.

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

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

	IE	SN	MJ	BN
Security relevant invokes	2	2	4	2
Original size of binaries (kb)	35.2	23.2	196.2	81.8
Inlining duration (s)	0.56	0.48	1.80	1.25
Size increase (inlining) (%):	1.1	1.1	0.2	0.3

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 programs 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 S^3MS. 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 [9] 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.

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

[1] I. Aktug, M. Dam and D. Gurov, Provably correct runtime monitoring, in: *FM*, 2008, pp. 262–277.

[2] I. Aktug and K. Naliuka, Conspec – a formal language for policy specification, *Electron. Notes Theory Comput. Sci.* **197**(1) (2008), 45–58.

[3] L. Bauer, J. Ligatti and D. Walker, Composing security policies with polymer, in: *PLDI*, 2005, pp. 305–314.

[4] R. DeLine and M. Fähndrich, Enforcing high-level protocols in low-level software, in: *PLDI '01: Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, ACM, New York, USA, 2001, pp. 59–69.

[5] L. Desmet, W. Joosen, F. Massacci, K. Naliuka, P. Philippaerts, F. Piessens and D. Vanoverberghe, A flexible security architecture to support third-party applications on mobile devices, in: *CSAW*, 2007, pp. 19–28.

[6] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan and D. Vanoverberghe, Security-by-contract on the .net platform, *Inf. Secur. Tech. Rep.* **13**(1) (2008), 25–32.
[7] N. Dragoni and F. Massacci, Security-by-contract for web services, in: *SWS*, 2007, pp. 90–98.
[8] N. Dragoni, F. Massacci, K. Naliuka and I. Siahaan, Security-by-contract: Toward a semantics for digital signatures on mobile code, in: *EuroPKI*, 2007, pp. 297–312.
[9] Ú. Erlingsson, The inlined reference monitor approach to security policy enforcement, PhD thesis, Department of Computer Science, Cornell University, 2004.
[10] J.S. Foster, T. Terauchi and A. Aiken, Flow-sensitive type qualifiers, in: *PLDI*, 2002, pp. 1–12.
[11] S.N. Freund and J.C. Mitchell, The type system for object initialization in the jave bytecode language, *ACM Trans. Program. Lang. Syst.* **21**(6) (1999), 1196–1250.
[12] K.W. Hamlen, G. Morrisett and F.B. Schneider, Certified in-lined reference monitoring on .net, in: *PLAS*, 2006, pp. 7–16.
[13] X. Leroy, Java bytecode verification: Algorithms and formalizations, *J. Autom. Reason.* **30**(3,4) (2003), 235–269.
[14] J. Ligatti, Policy enforcement via program monitoring, PhD thesis, Princeton University, June 2006.
[15] J. Ligatti, L. Bauer and D. Walker, Edit automata: Enforcement mechanisms for run-time security policies, *Int. J. Inf. Sec.* **4**(1,2) (2005), 2–16.
[16] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, 1997.
[17] F. Massacci and I. Siahaan, Simulating midlet's security claims with automata modulo theory, in: *PLAS*, 2008, pp. 1–9.
[18] ObjectWeb, Asm – home page: http://asm.objectweb.org/, February 2008.
[19] B. Ray, Symbian signing is no protection from spyware: http://www.theregister.co.uk/2007/05/23/symbian_signed_ spyware, May 2007.
[20] S3MS, Security of software and services for mobile systems: http://www.s3ms.org/, 2007.
[21] F.B. Schneider, Enforceable security policies, *ACM Trans. Inf. Syst. Secur.* **3**(1) (2000), 30–50.
[22] C. Skalka and S.F. Smith, History effects and verification, in: *APLAS*, 2004, pp. 107–128.
[23] D. Vanoverberghe and F. Piessens, A caller-side inline reference monitor for an object-oriented intermediate language, in: *FMOODS*, 2008, pp. 240–258.
[24] D. Walker, A type system for expressive security policies, in: *POPL '00: Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, New York, USA, 2000, pp. 254–267.