

# Provably Correct Runtime Monitoring

Irem Aktug, Mads Dam, Dilian Gurov

CSC KTH  
Stockholm, Sweden



Formal Methods 2008

# Runtime Monitoring

A *monitor* operates by observing the behavior of a target program and terminating the program when an action that violates the policy is about to occur.

A *policy* is a predicate on the set of all possible sequences of actions and selects only the **acceptable** sequences.

Monitoring has been implemented in the following two ways:

- *external monitoring*: external entities that run **in parallel** with the target program (e.g. firewalls),
- *monitor inlining*: the program is rewritten to make it **self-monitoring** (e.g. memory sandboxing).

- General purpose monitor inlining has been introduced by Evans and Twyman 1999/Erlingsson and Schneider 2000.
- Monitor inlining is an active area of research:
  - Bauer, Ligatti, Walker 2005
  - Hamlen, Morrisett 2006
  - Vanoverberghe, Piessens 2008
  - Hamlen, Jones 2008

Yet correctness of inlining is a neglected issue.

# Provably Correct Inlining

An inlined program is *correctly inlined* for policy  $\mathcal{P}$ , if the inlined monitor is

- **sound**, and
- transparent.

# Provably Correct Inlining

An inlined program is *correctly inlined* for policy  $\mathcal{P}$ , if the inlined monitor is

- **sound**, and
- transparent.

## Our mission

For a given inlined program, we want to create a proof of correct inlining such that:

- can be automatically generated,
- efficiently checkable.

# Provably Correct Inlining

An inlined program is *correctly inlined* for policy  $\mathcal{P}$ , if the inlined monitor is

- **sound**, and
- transparent.

## Our mission

For a given inlined program, we want to create a proof of correct inlining such that:

- can be automatically generated,
- efficiently checkable.

## Our approach

Use Floyd-like logic to specify correct inlining!

- Target programs
- Enforceable policies and the policy language
- Monitor inlining
- A two-stage annotation scheme:
  - 1 Stage 1 specifies policy adherence
  - 2 Stage 2 specifies that the program contains a monitor
- Construction of correctness proofs for inlined programs

# Target Programs

- We consider `java bytecode programs`.
- Our security relevant actions are calls to and returns from a `fixed API`.
- We handle:
  - only sequential programs,
  - exception handling, and
  - inheritance.

# Example: Target Program

$L$	$M[L]$
L1	aload r0
L2	getfield gui
L3	dup
L4	astore r1
L5	invokevirtual GUI/AskConnect()Z
L6	istore r2
L7	aload r1
L8	instanceof GUI
L9	ifeq L12
L10	iload r2
L11	putstatic SecState/permission
L12	iload r2
L13	ireturn

Figure: A target application method

*"Don't send after read"*

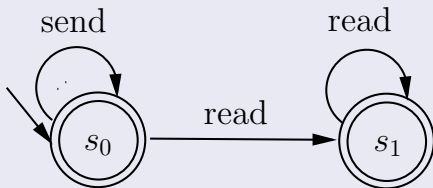


Figure: Security Automata for The Policy

*"Don't send after read"*

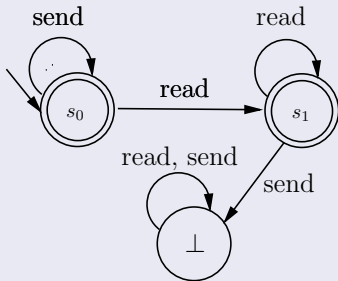


Figure: Automata for The Policy

# Policies

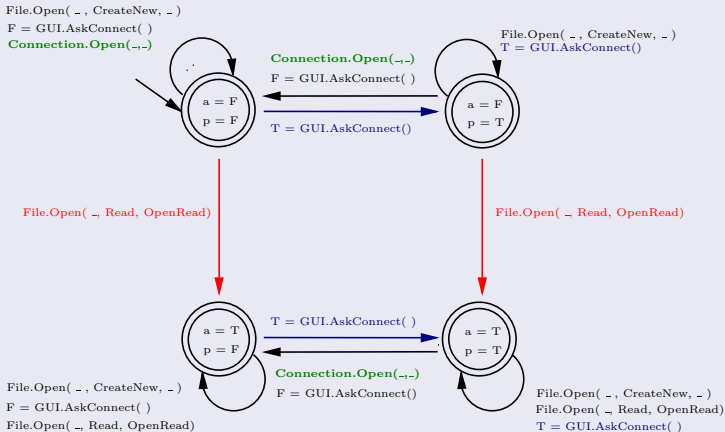
Example: ConSpec Language

*"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."*

# Policies

Example: ConSpec Language

*"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."*



# Policies

Example: ConSpec Language

*"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."*

```
SECURITY    STATE
            bool accessed = false;
            bool permission = false;

BEFORE      File.Open(string path, string mode, string access)
PERFORM     mode.equals(CreateNew)      → { skip; }
            mode.equals(Open) &&
            access.equals(OpenRead) → { accessed = true; }

AFTER       bool answer = GUI.AskConnect()
PERFORM     answer → { permission = true; }
            !answer → { permission = false; }

BEFORE      Connection.Open(string type, string address)
PERFORM     !accessed || permission -> { permission = false; }
```

# Monitor Inlining

- 1 The monitor state is inserted into the program by the inliner.  
*Embedded state:* The **concrete** representation of the monitor state in the program, usually in the form of global program variables.
- 2 Code is inserted around relevant actions to check if the action violates the policy; program is terminated in case the action is violating and the embedded state is updated otherwise.

Variants include:

- Wrapping (Naccio'99)
- Scattered (PSLang/POet'00)
- Central (Polymer'05)

## The problem

How can we specify that a program has been correctly inlined for a given policy?

## The problem

How can we specify that a program has been correctly inlined for a given policy?

## Reformulation of the Problem

How can we specify that a program has an embedded monitor for the policy?

# Specifying Correct Monitor Inlining

The target program is annotated in two steps:

# Specifying Correct Monitor Inlining

The target program is annotated in two steps:

- 1 Level I: we insert a correct monitor into the program using [specification variables](#).

# Specifying Correct Monitor Inlining

The target program is annotated in two steps:

- 1 Level I: we insert a correct monitor into the program **using specification variables**.
- 2 Level II: we specify that an embedded state exists such that:
  - the embedded monitor is in "synch" with the specified monitor **immediately prior** to execution of a security relevant action, and
  - the updates to the embedded state are made **locally**, that is by the method that executes the security relevant method call.

# Level I Annotations

(Policy Annotations)

- Characterize policy adherence,

# Level I Annotations

(Policy Annotations)

- Characterize policy adherence,
- Created using the policy:
  - The security state is represented by ghost variables (ghost state), updated to mimic the automaton transitions, and set to  $\perp$  if no transition is available.
  - The ghost state is asserted to be defined at critical points.

*"Applications are allowed to access existing files for reading only, and are required, once such a file has been accessed, to obtain approval from the user each time a connection is to be opened."*

```
SECURITY    STATE
            bool accessed = false;
            bool permission = false;

BEFORE      File.Open(string path, string mode, string access)
PERFORM     mode.equals(CreateNew)      → { skip; }
            mode.equals(Open) &&
            access.equals(OpenRead) → { accessed = true; }

AFTER      bool answer = GUI.AskConnect()
PERFORM     answer → { permission = true; }
            !answer → { permission = false; }

BEFORE      Connection.Open(string type, string address)
PERFORM     !accessed || permission -> { permission = false; }
```

# Level I Annotations

## Target Program

$L$	$M[L]$
L1	aload r0
L2	getfield gui
L3	dup
L4	astore r1
L5	invokevirtual GUI/AskConnect()Z
L6	istore r2
L7	aload r1
L8	instanceof GUI
L9	ifeq L12
L10	iload r2
L11	putstatic SecState/permission
L12	iload r2
L13	ireturn

Figure: An target application method

# Level I Annotations

Example: Level I Annotations

$A^I[L]$	$L$	$M[L]$
	L1	aload r0
	L2	getfield gui
	L3	dup
	L4	astore r1
$\{ \text{Defined}(gs) \}$	L5	invokevirtual GUI/AskConnect()Z
$\{ gs := \delta_{\perp}(gs, a) \}$	L6	istore r2
	L7	aload r1
	L8	instanceof GUI
	L9	ifeq L12
	L10	iload r2
	L11	putstatic SecState/permission
	L12	iload r2
	L13	ireturn

Figure: An application method with level I annotations for the example policy

## Theorem: Correctness of Level I Annotations

Program  $\mathbb{T}$  annotated with level I annotations for policy  $\mathcal{P}$  is valid, if and only if  $\mathbb{T}$  adheres to  $\mathcal{P}$ .

# Level II Annotations

(Synchronisation Check Annotations)

- Characterize the existence of a concrete monitor in the program
- Obtained by extending level I annotations

# Level II Annotations

(Synchronisation Check Annotations)

- Characterize the existence of a concrete monitor in the program
- Obtained by extending level I annotations

*Synchronisation Assertion* states the **equality** of the ghost state and the embedded state, and is asserted for every method at:

- method entry (pre-condition),
- method exit (post-condition),
- before each method call.

# Level II Annotations

Example: Level I Annotations

$A^I[L]$	$L$	$M[L]$
	L1	aload r0
	L2	getfield gui
	L3	dup
	L4	astore r1
$\{ \text{Defined}(gs) \}$	L5	invokevirtual GUI/AskConnect()Z
$\{ gs := \delta_{\perp}(gs, a) \}$	L6	istore r2
	L7	aload r1
	L8	instanceof GUI
	L9	ifeq L12
	L10	iload r2
	L11	putstatic SecState/permission
	L12	iload r2
	L13	ireturn

Figure: An application method with level I annotations for the example policy

# Level II Annotations

Example: Level II Annotations

$A^{II}[L]$	$L$	$M[L]$
$\{gs = \text{SecState}\}$	L1	aload r0
	L2	getfield gui
	L3	dup
	L4	astore r1
$\{\text{Defined}(gs) \wedge gs = \text{SecState}\}$	L5	invokevirtual GUI/AskConnect()Z
$\{gs := \delta_{\perp}(gs, a)\}$	L6	istore r2
	L7	aload r1
	L8	instanceof GUI
	L9	ifeq L12
	L10	iload r2
	L11	putstatic SecState/permission
	L12	iload r2
$\{gs = \text{SecState}\}$	L13	ireturn

Figure: An application method with level II annotations for the example policy

## Theorem: Level II Characterization

The level II annotations of  $\mathbb{T}$  for policy  $\mathcal{P}$  with the embedded state  $\vec{ms}$  is valid if, and only if,  $\vec{ms}$  identifies a method-local monitor for  $\mathcal{P}$ .

# Correct Monitor Inlining

Given a program  $T$ , a policy  $\mathcal{P}$  and an embedded state  $\vec{ms}$ ,  
how do we show that the level II annotated program is valid?

# Correct Monitor Inlining

Given a program  $T$ , a policy  $\mathcal{P}$  and an embedded state  $\vec{ms}$ , how do we show that the level II annotated program is valid?

If  $T$  is a "nicely" inlined program then level II annotations can be completed to full annotations and the problem is reduced to checking local validity.

Checking local validity consists of

- constructing verification conditions using the axiomatic semantics of single instructions
- discharging the resulting verification conditions

# Correct Monitor Inlining

Level III ("Full") Annotations for the Inliner

Full annotations generated by:

- 1 Adding the synchronization annotation as precondition to unlined instructions
- 2 Propagating the synchronization annotation from the bottom to the top of the inlined blocks using a weakest precondition calculator

**Proof of correct inlining** can be constructed for nicely inlined programs.

# Correct Monitor Inlining

## Level III ("Full") Annotations for the Inliner

Full annotations generated by:

- 1 Adding the synchronization annotation as precondition to uninlined instructions
- 2 Propagating the synchronization annotation from the bottom to the top of the inlined blocks using a weakest precondition calculator

**Proof of correct inlining** can be constructed for nicely inlined programs.

A program is "nicely" inlined if

- the problem of computing the weakest precondition of inlined blocks is decidable,
- the problem of discharging the verification conditions arising from the local validity of the full annotations is decidable

# Correctness of an Inliner

An inliner is a *well-behaved inliner* if it produces nicely inlined programs.

**Correctness of a well-behaved inliner** I can be proven by showing that given any program  $T$  and policy  $\mathcal{P}$ , a proof of inlining correctness can be constructed for the inlined program  $I(T, \mathcal{P})$ .

In this work we introduce a two-level annotation scheme on java bytecode programs where:

- ① level I annotations characterize policy adherence,
  - ② level II annotations characterize existence of a (method-local) monitor in the program.
- The annotation scheme can be used to show that a program has been correctly inlined.

In this work we also describe how to compute full annotations for "nicely" inlined programs which reduce the problem of proving correct inlining to checking local validity of the fully annotated program.

- The annotation scheme can be used in a proof-carrying code setting for certifying monitor compliance to the code consumer.
- The annotation scheme can be used to show correctness of an inliner.

- Constructing a **proof-carrying code** framework
- Extending the annotation scheme to handle **multi-threaded programs**
- Adding new features to the policy language, e.g. to define per-object policies
- Finding abstraction functions that identify the embedded state in unlined but yet self-monitoring programs
- Extending the correctness result to cover transparency