Security-Policy Monitoring and Enforcement with JavaMOP

Soha Hussein\textsuperscript{1} \hspace{1cm} Patrick Meredith\textsuperscript{2} \hspace{1cm} Grigore Rosu\textsuperscript{3}

\textsuperscript{1}Univ. of Ain Shams, Egypt. \hspace{1cm} \textsuperscript{2}Univ. of Illinois at Urbana-Champaign, USA.

PLAS'12
INTRODUCTION & MOTIVATION

vs.
Introduction & Motivation

- **Security Enforcement Mechanisms:**

- Designed for enforcing security properties.

- VS.

- Execution Monitors, Rewriting Model, Edit Automata

- SASI, Naccio, PoET/PSLang, Polymer and SPoX.
Introduction & Motivation

Security Enforcement Mechanisms:
- Designed for enforcing security properties.

  Execution Monitors, Rewriting Model, Edit Automata

  SASI, Naccio, PoET/PSLang, Polymer and SPoX.

Runtime Monitoring and Verification:
- Designed to be generic.

  Used to enforce functional correctness of post-production programs or debugging and testing programs during the production phase.

  MOP.
Motivation

- Use RV-based system (JavaMOP) to specify and enforce security policies.

- Provide a means to support composition and conflicts among policies.

- Measure performance between JavaMOP and other security-enforcement systems.
Monitoring oriented programming (MOP) is a formalism-generic runtime verification and monitoring framework.

JavaMOP is the Java instance for MOP.

Output of JavaMOP is an AspectJ file.
MOP General Terms & Features

- **Generic Formalism**
- **Catching Validation or Violations**
- **Parametric Properties**
MOP General Terms & Features

- Generic Formalism
- Catching Validation or Violations
- Parametric properties.
MOP General Terms & Features

- **Generic Formalism**
- **Catching Validation or Violations**
- **Parametric properties.**

**MOP Monitor**

**Monitor’s Environment**

**Events:**

- JoinPoint + PointCut
MOP General Terms & Features

- **Generic Formalism**
- **Catching Validation or Violations**
- **Parametric properties.**

---

**Monitor’s Environment**

**Events:**
JoinPoint+PointCut

**Property:**
ere, cfg, pttl
MOP General Terms & Features

- **Generic Formalism**
- **Catching Validation or Violations**
- **Parametric properties.**

**MOP Monitor**

**Monitor’s Environment**

**Events:**
JoinPoint+PointCut

**Property:**
eres, cfg, ptlTL

**Handler**
Validation & Violation
Enforcing Security Policies

- Access Control Policies.
  - Safe Lock. §2.
  - Disable System Calls. §3.1.
  - SQL Injection. §3.3.
  - Separation of Duties. §3.2.
  - Variations of the Chinese Wall. §3.4.

- File Network Wall

- Original Chinese Wall
CHINESE WALL

The policy attempts to keep users in the system from accessing objects of different datasets that are in the same Conflict Class.

In the monitoring world this is translated to:

We want to check each and every access of objects in the subject and make sure that the object being accessed does not lie in conflict with previously accessed objects.
Chinese Wall in JavaMOP
Chinese Wall in JavaMOP

Subject S
Call Stack
Chinese Wall in JavaMOP

Subject S
Call Stack
Chinese Wall in JavaMOP

What we want is:
2. means to track the call stack of subjects.
3. carry the necessary checks when matched.
What we want is:
1. A parametric specification.
2. Means to track the call stack of subjects.
3. Carry the necessary checks when matched.

```
event methodCall
before(Subject S):
call(* Subject.*(..)) && target(S)
```
What we want is:
2. means to track the call stack of subjects.
3. carry the necessary checks when matched.

**Subject S**
**Call Stack**

**Event methodCall**
before(Subject S):
call(* Subject.*(..)) 
&& target(S)

**Event methodReturn**
after(Subject S):
call(* Subject.*(..)) 
&& target(S)
Chinese Wall in JavaMOP

What we want is:
2. Means to track the call stack of subjects.
3. Carry the necessary checks when matched.

Subject S
Call Stack

Event access
before(Obj O): call(* Obj.Read()) && target(O)

Event method call
before(Subject S):
call(* Subject.*(..)) && target(S)

Event method return
after(Subject S):
call(* Subject.*(..)) && target(S)
Chinese Wall in JavaMOP

```java
ChineseWall(Subject S) {
    SubjectWall monitoredSubjectWall;
    Obj readObject;
    event methodCall before(Subject S): call(* Subject.*(..)) && target(S) {
        if (monitoredSubjectWall == null)
            monitoredSubjectWall = new SubjectWall(S); }

    event methodReturn after(Subject S): call(* Subject.*(..))&&target(S) {}

    event access before(Obj 0): call(* Obj.Read()) && target(0) {
        readObject = 0; }

cfg: S -> S access | S M | epsilon,
    M -> M methodCall M methodReturn
    | epsilon

@fail{
    SubjectWall sw = __MONITOR.monitoredSubjectWall;
    Obj o = __MONITOR.readObject;
    if(sw.conflictClassContains(o) && !sw.dataSetContains(o)){
        System.out.println("Chinese Wall is violated. Halting..");
        Runtime.getRuntime().halt(1); }
    sw.addToConflictClass(o);
    sw.addToDataSet(o); }
```
Chinese Wall in JavaMOP

```java
ChineseWall(Subject S) {
    SubjectWall monitoredSubjectWall;
    Obj readObject;
    event methodCall before(Subject S): call(* Subject.*(..)) && target(S) {
        if (monitoredSubjectWall == null)
            monitoredSubjectWall = new SubjectWall(S);
    }

    event methodReturn after(Subject S): call(* Subject.*(..))&&target(S) {}

    event access before(Obj O): call(* Obj.Read()) && target(O) {
        readObject = 0;
    }

    cfg: S -> S access | S M | epsilon,
    M -> M methodCall M methodReturn
    | epsilon

    @fail{
        SubjectWall sw = __MONITOR.monitoredSubjectWall;
        Obj o = __MONITOR.readObject;
        if(sw.conflictClassContains(o) && !sw.dataSetContains(o)) {
            System.out.println("Chinese Wall is violated. Halting..");
            Runtime.getRuntime().halt(1);
        }
        sw.addToConflictClass(o);
        sw.addToDataSet(o);
    }
}
```
Chinese Wall in JavaMOP

```java
ChineseWall(Subject S) {
    SubjectWall monitoredSubjectWall;
    Obj readObject;
    event methodCall before(Subject S): call(* Subject.*(..)) && target(S) {
        if (monitoredSubjectWall == null)
            monitoredSubjectWall = new SubjectWall(S);
    }

    event methodReturn after(Subject S): call(* Subject.*(..)) && target(S) {};

    event access before(Obj O): call(* Obj.Read()) && target(O) {
        readObject = O;
    }

    cfg: S -> S access | S M | epsilon,
    M -> M methodCall M methodReturn
    | epsilon

    @fail{
        SubjectWall sw = __MONITOR.monitoredSubjectWall;
        Obj o = __MONITOR.readObject;
        if(sw.conflictClassContains(o) && !sw.dataSetContains(o)){
            System.out.println("Chinese Wall is violated. Halting..");
            Runtime.getRuntime().halt(1);
        }
        sw.addToConflictClass(o);
        sw.addToDataSet(o);
    }
}
```
Chinese Wall in JavaMOP

```java
ChineseWall(Subject S) {
    SubjectWall monitoredSubjectWall;
    Obj readObject;
    event methodCall before(Subject S): call(* Subject.*(..)) && target(S) {
        if (monitoredSubjectWall == null)
            monitoredSubjectWall = new SubjectWall(S);
    }
    event methodReturn after(Subject S) : call(* Subject.*(..))&&target(S) {}
    event access before(Obj O): call(* Obj.Read()) && target(O) {
        readObject = 0;
    }
    cfg: S -> S access l S M l epsilon, 
    M -> M methodCall M methodReturn
    l epsilon
    @fail{
        SubjectWall sw = __MONITOR.monitoredSubjectWall;
        Obj o = __MONITOR.readObject;
        if(sw.conflictClassContains(o) && !sw.dataSetContains(o)){
            System.out.println("Chinese Wall is violated. Halting..");
            Runtime.getRuntime().halt(1);
        }
        sw.addToConflictClass(o);
        sw.addToDataSet(o);
    }
}```
Policy Composition and Conflict Resolution in JavaMOP

JavaMOP, by default, allows multiple policies to coexist within a given target program.

However it makes no guarantees on how they will operate together if their events interfere with each other, that is, if they happen to select some of the same program points.
Composition and Conflict Problem
Composition and Conflict Problem
Composition and Conflict Problem

Tuesday, June 12, 12
Composition and Conflict Problem

<table>
<thead>
<tr>
<th></th>
<th>Halt</th>
<th>Skip</th>
<th>Proceed</th>
<th>Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>Halt</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Skip</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>Proceed</td>
<td>?</td>
<td>?</td>
<td>Proceed</td>
<td>?</td>
</tr>
<tr>
<td>Exec</td>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
COMPOSITION AND CONFLICT PROBLEM
Composition and Conflict Problem

Idea: Allow monitors to exchange events.
Policy Composition and conflicting JavaMOP
Policy Composition and conflicting JavaMOP

```java
RestrictSystemCalls() {
    event systemCalls Object around():
        call(* Runtime.exec(...)){}
    ere : systemCalls+
    @match {
        __SKIP; // action is bypassed
    }
}
```
Policy Composition and conflicting JavaMOP

```java
RestrictSystemCalls() {
    event systemCalls Object around():
        call(* Runtime.exec(...)){}
        ere : systemCalls +
    @match {
        __SKIP; // action is bypassed
    }
}
```

```java
FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(...))
        || call(* File.createNewFile())
        || ... //other file access methods
    {}
    event networkAccess before():
        call(URL.new(...))
        || call(Socket.new(...))
        || ... //other network access methods
    {}
    pttl1: (fileAccess => <-> networkAccess)
        or (networkAccess => <-> fileAccess)
    @validation {
        System.out.println(
            "Conflicting resource access. 
            + "Halting.."");
        Runtime.getRuntime().halt(1);
    }
}
```
Policy Composition and conflicting JavaMOP

RestrictSystemCalls() {
    event systemCalls OBJECT around():
        call(* Runtime.exec(...))
    where systemCalls+
    @match {
        __SKIP; // action is bypassed
    }
}

FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(...))
        || call(* File.createNewFile())
        || ... // other file access methods
    }
    event networkAccess before():
        call(URL.new(...))
        || call(Socket.new(...))
        || ... // other network access methods
    }
    ptltl: (fileAccess => <> networkAccess)
        or (networkAccess => <> fileAccess)
    @validation {
        System.out.println(
            "Conflicting resource access."
            + "Halting...");
        Runtime.getRuntime().halt(1);
    }
}
Policy Composition and conflicting JavaMOP

```
RestrictSystemCalls() {
    event systemCalls Object around():
    call(* Runtime.exec(...)){
        ere : systemCalls+
        @match {
            _SKIP; // action is bypassed
        }
    }
}
```

```
FileNetworkWall() {
    event fileAccess before():
    call(* Runtime.exec(...)),
|| call(* File.createNewFile())
|| ... //other file access methods
}

    event networkAccess before():
    call(URL.new(...))
|| call(Socket.new(...))
|| ... //other network access methods
{
    ptltl: (fileAccess => <> networkAccess)
    or (networkAccess => <> fileAccess)
    @validation {
        System.out.println(
            "Conflicting resource access."
            + "Halting..");
        Runtime.getRuntime().halt(1);
    }
}
```
Policy Composition and conflicting JavaMOP

```java
RestrictSystemCalls() {
    event systemCalls Object around():
        call(* Runtime.exec(...)){}
    ere : systemCalls+
    @match {
        SKIP; // action is bypassed
    }
}
```

```java
FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(...))
    || call(* File.createNewFile())
    || ... //other file access methods
    }
    event networkAccess before():
        call(URL.new(...))
    || call(Socket.new(...))
    || ... //other network access methods
    }
    ptltl: (fileAccess => <> networkAccess)
    or (networkAccess => <> fileAccess)
    @validation {
        System.out.println("Conflicting resource access. ",
        "Halting...");
        Runtime.getRuntime().halt(1);
    }
}
```
Policy Composition and conflicting JavaMOP

RestrictSystemCalls() {
  event systemCalls Object around():
    call(* Runtime.exec(...)){}
  where systemCalls +
  @match {
    __SKIP; // action is bypassed
  }
}

FileNetworkWall() {
  event fileAccess before():
    call(* Runtime.exec(...))
    || call(* File.createNewFile())
    || ... // other file access methods
  }
  event networkAccess before():
    call(URL.new(...))
    || call(Socket.new(...))
    || ... // other network access methods
  }
  pt1: (fileAccess => <*> networkAccess)
  or (networkAccess => <*> fileAccess)
  @validation {
    System.out.println("Conflicting resource access. " + "Halting...");
    Runtime.getRuntime().halt(1)
  }
}
Policy Composition and conflicting JavaMOP

```java
RestrictSystemCalls() {
    event systemCalls Object around():
        call(* Runtime.exec(...)) {}
    ere : systemCalls+
    @match {
        __SKIP; // action is bypassed
    }
}
```

```java
FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(...))
    || call(* File.createNewFile())
    || ... // other file access methods
    }
    event networkAccess before():
        call(URL.new(...))
    || call(Socket.new(...))
    || ... // other network access methods
    }
    pttl1: (fileAccess => <> networkAccess)
    or (networkAccess => <> fileAccess)
    @validation {
        System.out.println("Conflicting resource access. "+ "Halting...");
        Runtime.getRuntime().halt(1)
    }
}
The Problem is that every monitor here is actually firing action without coordination. What we need is coordination.
Monitoring of Monitors

```java
FileNetworkWall() {
    event fileAccess before() {
        call(* Runtime.exec(..))
        || call(* File.createNewFile())
        || ... // other file access methods
    }
    event networkAccess before() {
        call(URL.new(..))
        || call(Socket.new(..))
        || ... // other network access methods
    }
    @validation {
        if (fileAccess => <> networkAccess)
            or (networkAccess => <> fileAccess)
            System.out.println(
                "Conflicting resource access. "
                + "Halting..");
        Runtime.getRuntime().halt(1);
    }
}
Monitoring of Monitors

```java
FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(...))
        call(* File.createNewFile())
        ... //other file access methods
    }

    event networkAccess before():
        call(URL.new(...))
        call(Socket.new(...))
        ... //other network access methods
    }
```
Monitoring of Monitors

```javascript
FileNetworkWall() {
    event fileAccess before();
        call(* Runtime.exec(....))
        || call(* File.createNewFile())
        || ... //other file access methods
    }
    event networkAccess before();
        call(URL.new(....))
        || call(Socket.new(....))
        || ... //other network access methods
    }
    ptt1: networkAccess => <*>fileAccess
        @match{ __MONITOR.M1.haltFN(); }
    ptt1: fileAccess => <*>networkAccess
        @match{ __MONITOR.M1.haltNF(); }
    }
```
Monitoring of Monitors

```java
FileNetworkWall() {
    event fileAccess before();
    call(* Runtime.exec(....)) || call(* File.createNewFile()) || ... // other file access methods
}

event networkAccess before();
    call(URL.new(....)) || call(Socket.new(....)) || ... // other network access methods

pttl1 : networkAccess => <*>fileAccess @match{ __MONITOR.M1.haltFN(); }

pttl1 : fileAccess => <*>networkAccess @match{ __MONITOR.M1.haltNF(); }
```
Monitoring of Monitors

```java
FileNetworkWall() {
    event FileAccess before() {
        call(* Runtime.exec(...))
        || call(* File.createNewFile())
    }
    || ... // other file access methods

    event networkAccess before() {
        call(URL.new(...))
        || call(Socket.new(...))
    }
    || ... // other network access methods

    pttl1 : networkAccess => FileAccess
    @match{ __MONITOR.M1 haltFN(); }

    pttl1 : FileAccess => networkAccess
    @match{ __MONITOR.M1 haltNF(); }
}
```
Monitoring of Monitors

```java
ConflictManager() {
    event haltFN
        after(): call( void Manager.haltFN() ) {}
    event haltNF
        after(): call( void Manager.haltNF() ) {}
    event proceed
        after(): call( void Manager.proceed() ) {}
    event skip
        after(): call( void Manager.skip() ) {}  

    ptltl: haltFN => <+> proceed @validation{
        System.out.println("FileNetworkWall violated...Halting.");
        Runtime.getRuntime().halt(1);
    }
    ptltl: haltNF => (*) proceed @validation{
        System.out.println("FileNetworkWall violated...Halting.");
        Runtime.getRuntime().halt(1);
    }
}

FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(...))
        || call(* File.creatNewFile())
        || ... // other file access methods
    }
    event networkAccess before():
        call( URL.new(...) )
        || call( Socket.new(...) )
        || ... // other network access methods
    }

    ptltl : networkAccess => <+> fileAccess @match{ __MONITOR.M1 haltFN(); }
    ptltl : fileAccess => <+> networkAccess @match{ __MONITOR.M1 haltNF(); }
```
Monitoring of Monitors

ConflictManager() {
  event haltFN
    after():call(void Manager.haltFN()){}
  event haltNF
    after():call(void Manager.haltNF()){}
  event proceed
    after():call(void Manager.proceed()){}
  event skip
    after():call(void Manager.skip()){}
  pttl1: haltFN => <*>proceed
  @validation{
    System.out.println("FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
  }
  pttl1: haltNF => (*).proceed
  @validation{
    System.out.println("FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
  }
}

FileNetworkWall() {
  event fileAccess before():
    call(* Runtime.exec(...))
    || call(* File.creatNewFile())
    || ... //other file access methods
  }
  event networkAccess before():
    call(URL.new(...))
    || call(Socket.new(...))
    || ... //other network access methods
  }
  pttl1 : networkAccess => <*>fileAccess
  @match{ __MONITOR.M1.haltFN(); }
  pttl1: fileAccess => <*>networkAccess
  @match{ __MONITOR.M1.haltNF(); }
}
Monitoring of Monitors

ConflictManager() {
  event haltFN
    after():call(void Manager::haltFN()){}
  event haltNF
    after():call(void Manager::haltNF()){}
  event proceed
    after():call(void Manager.proceed()){}
  event skip
    after():call(void Manager.skip()){}
  pttl1: haltFN => <*>proceed @validation{
    System.out.println("FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
  }
  pttl1: haltNF => (*)(*)proceed @validation{
    System.out.println("FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
  }
}

FileNetworkWall() {
  event fileAccess before():
    call(* Runtime.exec(...))
    || call(* File.creatNewFile())
    || ... //other file access methods
  }
  event networkAccess before():
    call(URL.new(...))
    || call(Socket.new(...))
    || ... //other network access methods
  }
  pttl1 : networkAccess => <*>fileAccess @match{ __MONITOR_M1::haltFN(); }
  pttl1: fileAccess => <*>networkAccess @match{ __MONITOR_M1::haltNF(); }
# Monitoring of Monitors

```java
ConflictManager() {
    event haltFN
        after():call(void Manager::haltFN()){}
    event haltNF
        after():call(void Manager::haltNF()){}
    event proceed
        after():call(void Manager::proceed()){}
    event skip
        after():call(void Manager::skip()){}
    pttl1: haltFN => <*>proceed
        @validation{
            System.out.println("FileNetworkWall violated..Halting.");
            Runtime.getRuntime().halt(1);
        }
    pttl1: haltNF => (*)proceed
        @validation{
            System.out.println("FileNetworkWall violated..Halting.");
            Runtime.getRuntime().halt(1);
        }
}
```

```java
FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(..))
    || call(* File.creatNewFile())
    || ... //other file access methods
        {}
    event networkAccess before():
        call(URL.new(..))
    || call(Socket.new(..))
    || ... //other network access methods
        {}
    pttl1 : networkAccess => <*>fileAccess
        @match{ __MONITOR.M1::haltFN(); }
    pttl1: fileAccess => <*>networkAccess
        @match{ __MONITOR.M1::haltNF(); }
```
Monitoring of Monitors

```java
ConflictManager() {
    event haltF1
        after(): call(void Manager haltF1) {}
    event haltNF
        after(): call(void Manager haltNF) {}
    event proceed
        after(): call(void Manager proceed) {}
    event skip
        after(): call(void Manager skip) {}
    pttl1: haltF1 => <>proceed
    @validation{
        System.out.println("FileNetworkWall violated..Halting.");
        Runtime.getRuntime().halt(1);
    }
    pttl1: haltNF => (*)proceed
    @validation{
        System.out.println("FileNetworkWall violated..Halting.");
        Runtime.getRuntime().halt(1);
    }
}
```

```java
FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(..))
        || call(* File.createNewFile())
        || ... //other file access methods
    }
    event networkAccess before():
        call(URL.new(..))
        || call(Socket.new(..))
        || ... //other network access methods
    }

    pttl1 : networkAccess => <>fileAccess
    @match{ __MONITOR.M1 haltF1(); }
    pttl1 : fileAccess => <>networkAccess
    @match{ __MONITOR.M1 haltNF(); }
```
Monitoring of Monitors

ConflictManager() {
    event haltFN
        after(): call(void Manager_haltFN())
    event haltNF
        after(): call(void Manager_haltNF())
    event proceed
        after(): call(void Manager_proceed())
    event skip
        after(): call(void Manager_skip())
    @validation
        System.out.println("FileNetworkWall violated..Halting.");
        Runtime.getRuntime().halt(1);
}

ptl: haltFN => <>proceed
@validation
    System.out.println("FileNetworkWall violated..Halting.");
    Runtime.getRuntime().halt(1);
}

FileNetworkWall() {
    event fileAccess before():
        call(* Runtime.exec(..))
    || call(* File.creatNewFile())
    || ... //other file access methods
{}
    event networkAccess before():
        call(URL.new(..))
    || call(Socket.new(..))
    || ... //other network access methods
{}

    ptl: networkAccess => <>fileAccess
@match{ __MONITOR.M1_haltFN(); }

    ptl: fileAccess => <>networkAccess
@match{ __MONITOR.M1_haltNF(); }
}
ConflictManager() {
    event haltFN
        after(): call(void Manager haltFN()){}
    event haltNF
        after(): call(void Manager haltNF()){}
    event proceed
        after(): call(void Manager proceed()){}
    event skip
        after(): call(void Manager skip()){}
    pttl: haltFN => **>proceed
    @validation{
        System.out.println(
            "FileNetworkWall violated...Halting.");
        Runtime.getRuntime().halt(1);
    }
    pttl: haltNF => (*)proceed
    @validation{
        System.out.println(
            "FileNetworkWall violated...Halting.");
        Runtime.getRuntime().halt(1);
    }
}
Other Security Concerns §4

- Level of Enforcement.
- Class of Policies Enforced.
- Correctness & Monitors Integrity.
- Restricting Java Reflection.
- AspectJ Correctness.
Three experiments were carried out.

The first is specialized to test the ChineseWall policy.

The second and the third experiments use the DaCapo benchmark suite (version 9.12-bach) and several JavaAPI security policies specified using JavaMOP and later using other IRM systems.
4.3 Evaluation and Performance

In this section we investigate the performance of JavaMOP when monitoring security policies.

Three experiments were carried out. The first experiment is specialized to test the ChineseWall policy, which is not relevant to any programs in the 0apalo benchmark suite. The second and the third experiments use the 0apalo benchmark suite, version 6, and several JavaxPI security policies specified using JavaMOP and later using other IRM systems. All experiments were performed on a machine with 1 GB of RAM, a Pentium with 2 GHz processor, and Ubuntu Linux 6.

### 4.3.1 JavaMOP Chinese Wall Performance

To test the ChineseWall security policy, we have customized a simulation program for stock workflow where users (subjects) are allowed to access objects in datasets that do not lay in the same conflict class of previously accessed objects. We tested the program at different numbers of loads (running subjects), datasets, and conflict classes. We also tested multiple different call depths (1n, 1m, 1k) for the call depth of each subject. Table 1 summarizes the results in three groups of columns. The first group of columns (subjects, datasets, conflict) shows the load size of the test performed. It is worth noting that for each load, the average percent overhead is calculated.

<table>
<thead>
<tr>
<th>#Subjects</th>
<th>#Datasets</th>
<th>#Conflict</th>
<th>%Overhead</th>
<th>Method call/return</th>
<th>Access</th>
<th>Total #events</th>
<th>#Trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1000</td>
<td>10</td>
<td>6</td>
<td>2000</td>
<td>1000</td>
<td>3000</td>
<td>51500</td>
</tr>
<tr>
<td>200</td>
<td>4000</td>
<td>20</td>
<td>9</td>
<td>8000</td>
<td>4000</td>
<td>12000</td>
<td>406000</td>
</tr>
<tr>
<td>300</td>
<td>9000</td>
<td>30</td>
<td>5</td>
<td>18000</td>
<td>9000</td>
<td>27000</td>
<td>1363500</td>
</tr>
<tr>
<td>400</td>
<td>16000</td>
<td>40</td>
<td>3</td>
<td>32000</td>
<td>16000</td>
<td>48000</td>
<td>3224000</td>
</tr>
<tr>
<td>500</td>
<td>25000</td>
<td>50</td>
<td>6</td>
<td>50000</td>
<td>25000</td>
<td>75000</td>
<td>6287500</td>
</tr>
</tbody>
</table>
Two experiments done on DaCapo benchmark suite.

We used the default input for DaCapo, and we used the –converge option to ensure the validity of our test by running each test multiple times, until the execution time converges. After convergence, the runtime is stabilized within 3%.
## JavaMOP Performance on DaCapo Benchmark

<table>
<thead>
<tr>
<th>Subject</th>
<th>HiddenFileAccess</th>
<th>DisableNetwork</th>
<th>FileCreation</th>
<th>FileNetworkWall</th>
<th>DisSysCalls</th>
<th>AllPolicies</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>3 64 64</td>
<td>1 0 0</td>
<td>2 14 14</td>
<td>0 1388 0</td>
<td>1 0</td>
<td>2 1416 56</td>
</tr>
<tr>
<td>batik</td>
<td>-1 122 122</td>
<td>1 685 685</td>
<td>-1 0 0</td>
<td>0 1692 685</td>
<td>-1 0</td>
<td>1 2071 1542</td>
</tr>
<tr>
<td>eclipse</td>
<td>-1 642 642</td>
<td>1 438 438</td>
<td>2 28 28</td>
<td>1 1958 439</td>
<td>1 0</td>
<td>1 3047 1542</td>
</tr>
<tr>
<td>fop</td>
<td>1 121 121</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>1 667 0</td>
<td>1 0</td>
<td>1 548 49</td>
</tr>
<tr>
<td>h2</td>
<td>2 15 15</td>
<td>0 0 0</td>
<td>-1 0 0</td>
<td>1 73 0</td>
<td>-1 0</td>
<td>-1 70 9</td>
</tr>
<tr>
<td>jython</td>
<td>0 2726 2726</td>
<td>2 0 0</td>
<td>-1 2 2</td>
<td>1 7347 0</td>
<td>-2 0</td>
<td>-1 10049 2720</td>
</tr>
<tr>
<td>luindex</td>
<td>1 25 25</td>
<td>0 0 0</td>
<td>1 256 256</td>
<td>0 24534 0</td>
<td>-1 0</td>
<td>-2 16475 176</td>
</tr>
<tr>
<td>lusearch</td>
<td>-2 1549 1549</td>
<td>0 0 0</td>
<td>0 0 0</td>
<td>0 3807 0</td>
<td>0 0</td>
<td>0 3670 1033</td>
</tr>
<tr>
<td>pmd</td>
<td>-1 3138 3138</td>
<td>-1 0 0</td>
<td>-1 0 0</td>
<td>-3 6288 0</td>
<td>0 0</td>
<td>-1 7182 2242</td>
</tr>
<tr>
<td>sunflow</td>
<td>0 13 13</td>
<td>1 0 0</td>
<td>1 0 0</td>
<td>2 72 0</td>
<td>1 0</td>
<td>0 67 9</td>
</tr>
<tr>
<td>tomcat</td>
<td>1 37 37</td>
<td>2 3 3</td>
<td>1 0 0</td>
<td>2 20044 3</td>
<td>1 0</td>
<td>1 14680 39</td>
</tr>
<tr>
<td>tradebeans</td>
<td>0 13 13</td>
<td>0 0 0</td>
<td>1 0 0</td>
<td>0 3430 0</td>
<td>0 0</td>
<td>1 3427 9</td>
</tr>
<tr>
<td>tradesoap</td>
<td>0 15 15</td>
<td>0 0 0</td>
<td>1 0 0</td>
<td>0 2788 0</td>
<td>0 0</td>
<td>-1 2787 11</td>
</tr>
<tr>
<td>xalan</td>
<td>-1 23826 23826</td>
<td>0 1 1</td>
<td>-1 0 0</td>
<td>1 47741 1</td>
<td>-2 0</td>
<td>0 51144 17022</td>
</tr>
</tbody>
</table>
JavaMop Vs. SPOX & Polymer on DaCapo

Table 3 shows the number of events monitored for each specification inside the DaCapo benchmark, and Table 3 shows for each policy, three figures for the average percentage overhead resulted from using JavaMOP, SPoX, and Polymer respectively. Negative overheads are occasionally possible because additional code introduced by the weaving process changes the program structure in DaCapo, sometimes causing the benchmark to run slightly faster due to better instruction cache layout.

Again, JavaMop shows no significant performance overhead when any of the above specifications are enforced on DaCapo. We attribute this result to the fact that JavaMop had undergone extensive performance enhancement to allow multiple monitors to coexist right at the same time. SPoX on the other hand shows also no significant measurements for almost all DaCapo programs, except for fop, which takes an XSL.FO file, parses it, and formats it, generating a PDF file. We believe that the fop makes extensive usage of the file API methods that the policies used in the performance measurement are monitoring. Thus when the load of usage of the APIs used in SPoX specifications, which in turn increases the crosscutting points and requires more weaving to be done, SPoX shows average overhead.
Conclusion and Future Work

IRM can be considered as a specific instance of RV. Specifically, we demonstrated how JavaMOP, an RV system, is able to effectively and efficiently specify and monitor security policies.

We showed how JavaMOP can be used to resolve potential conflicts or composition among monitors.

Our experiments which showed that JavaMOP yields a better performance results when compared to SPoX and Polymer.

A formal framework for the composition of JavaMOP specifications is a direction for future research.
Security-Policy Monitoring and Enforcement with JavaMOP

Soha Hussein
Univ. of Ain Shams, Egypt.

Patrick Meredith
Univ. of Illinois at Urbana-Champaign, USA.

Grigore Rosu

PLAS’12