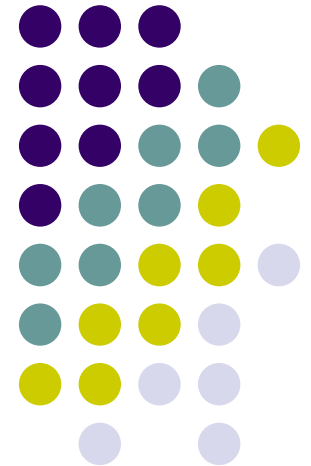


# Weaving Rewrite- Based Access Control Policies

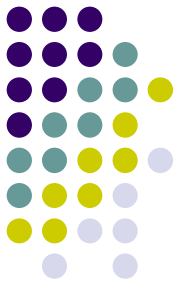
Anderson Santana de Oliveira  
Claude Kirchner and Helene Kirchner  
INRIA&LORIA

Presenter: Eric Ke Wang  
The U. of Hong Kong & INRIA&LORIA



# Outline

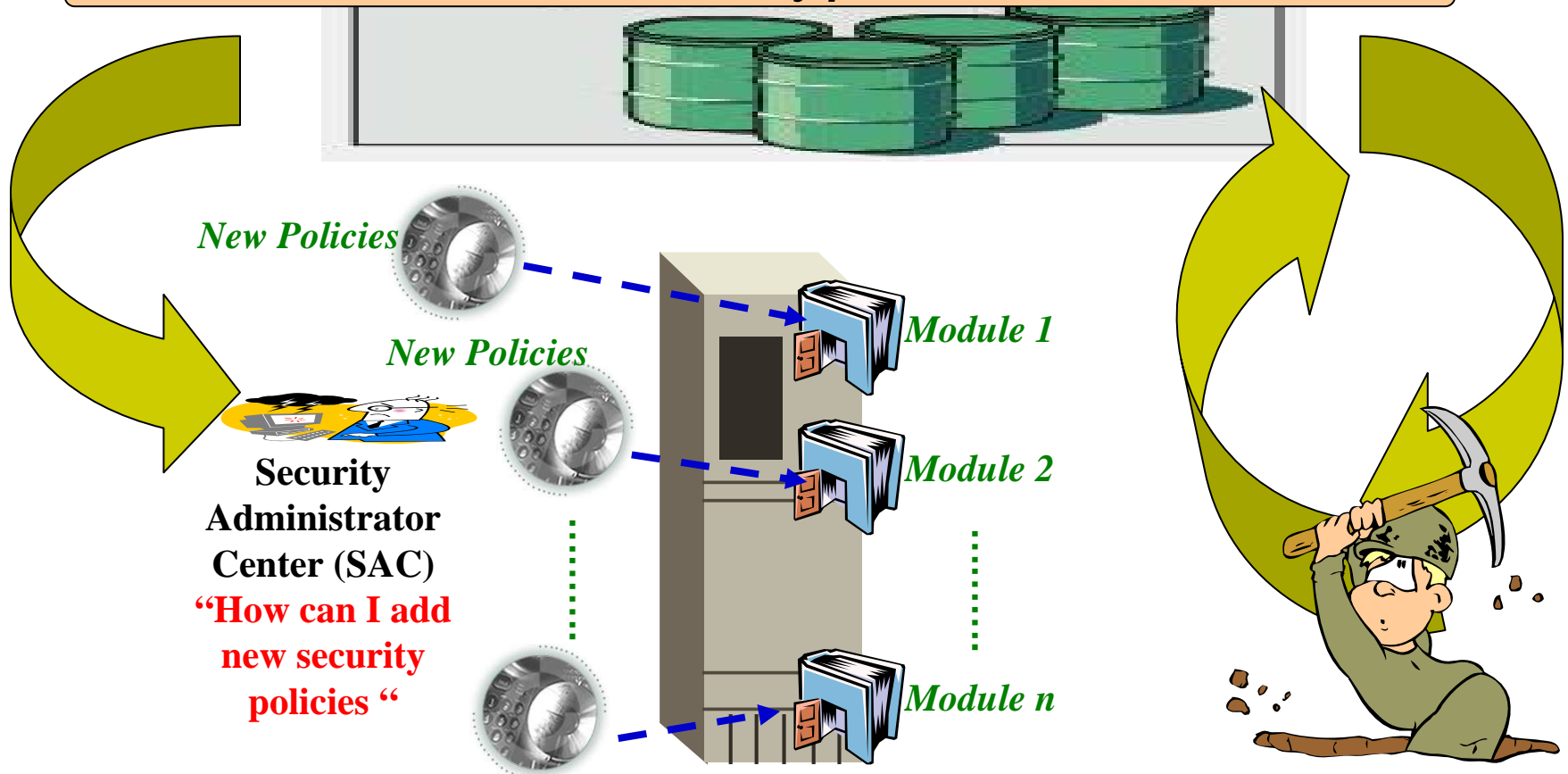
- Background & Motivations
- Our Schema
- Discussion
- Related work
- Conclusion & Future work
- Questions





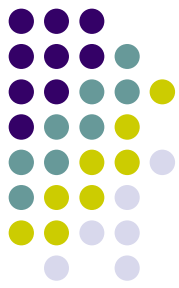
# What is the Problem ?

**Various Security policies !!!**



**Security Administrator Center (SAC)**  
**“How can I add new security policies “**

**Insert them into modules one by one?**



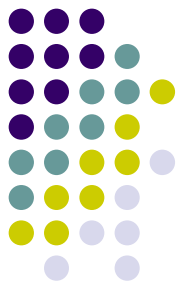
# Motivations-related problems

## How to enforce dynamic policies?

- How to formalize the security policies?
- How to weave these polices efficiently and effectively?

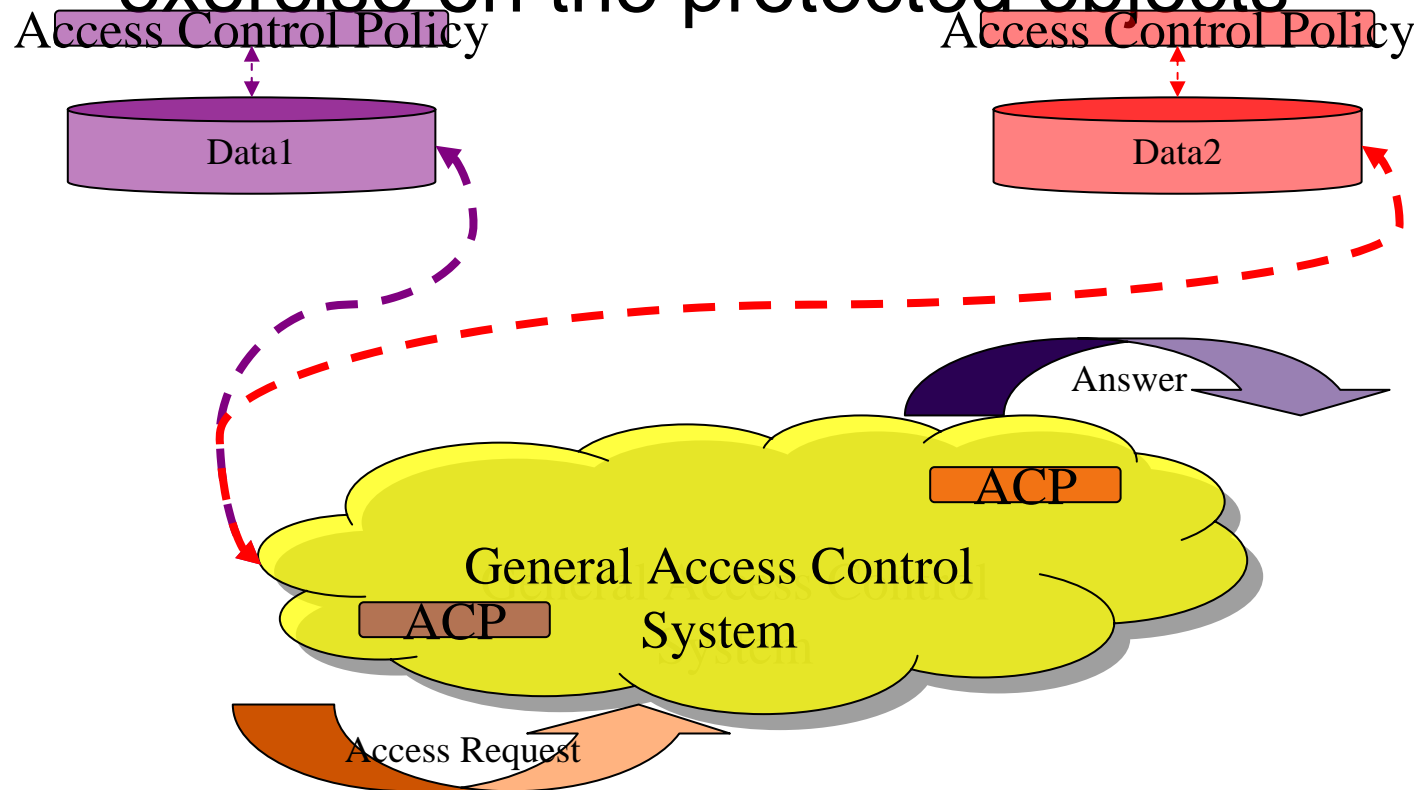
Current status: define policies in ad hoc, platform-dependent ways

- **Can't reuse policies on programs for different platforms especially for dynamic policies**
- **Hard to write, understand and reason about policies**
- **Can not guarantee the reliability of polices ( Terminating and Complete)**



# Access Control (AC) Policies

- An access control policy determines the operations and rights that subjects can exercise on the protected objects





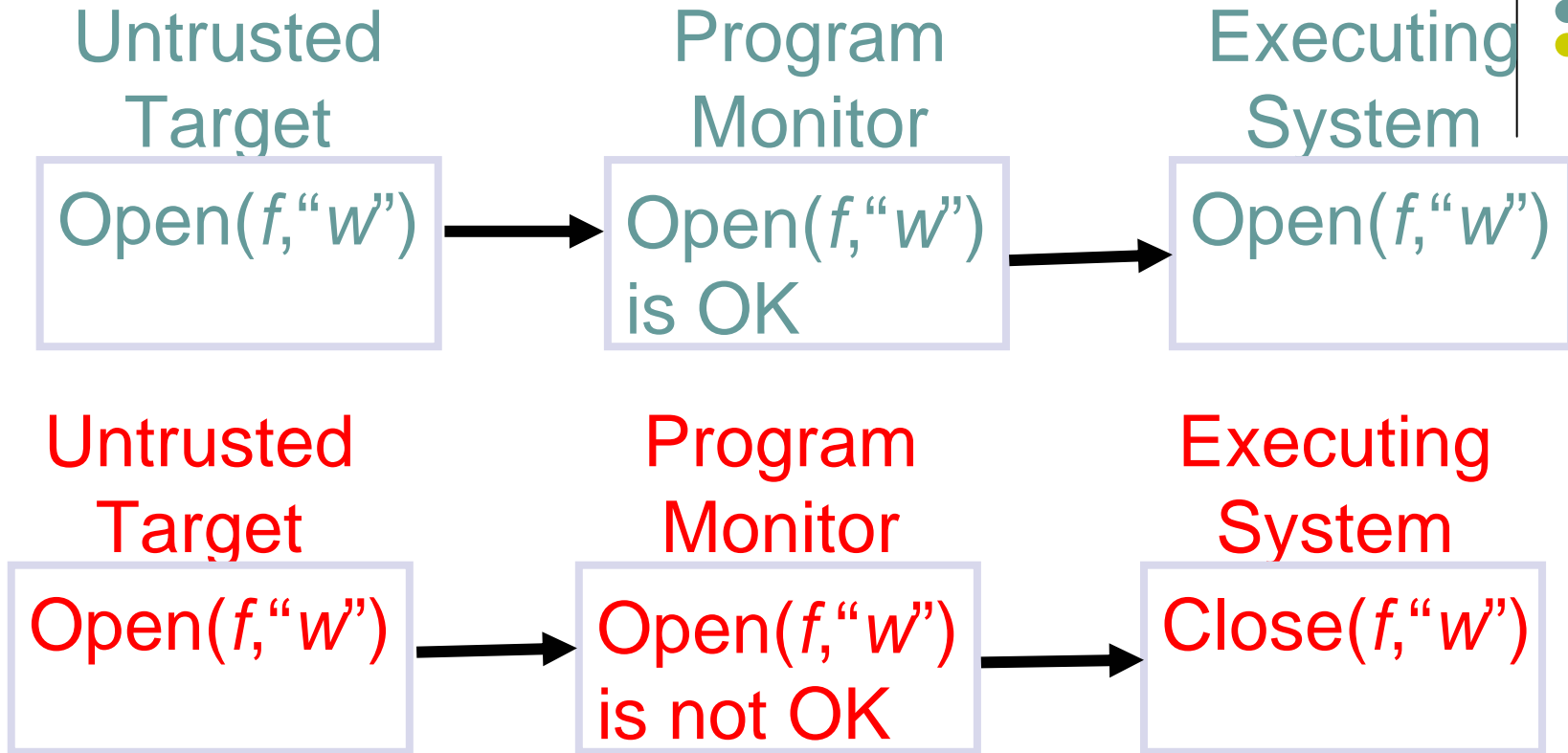
# Enforcing Access Control Policy

**One solution is reference monitoring**

**The other solution is program rewriter**



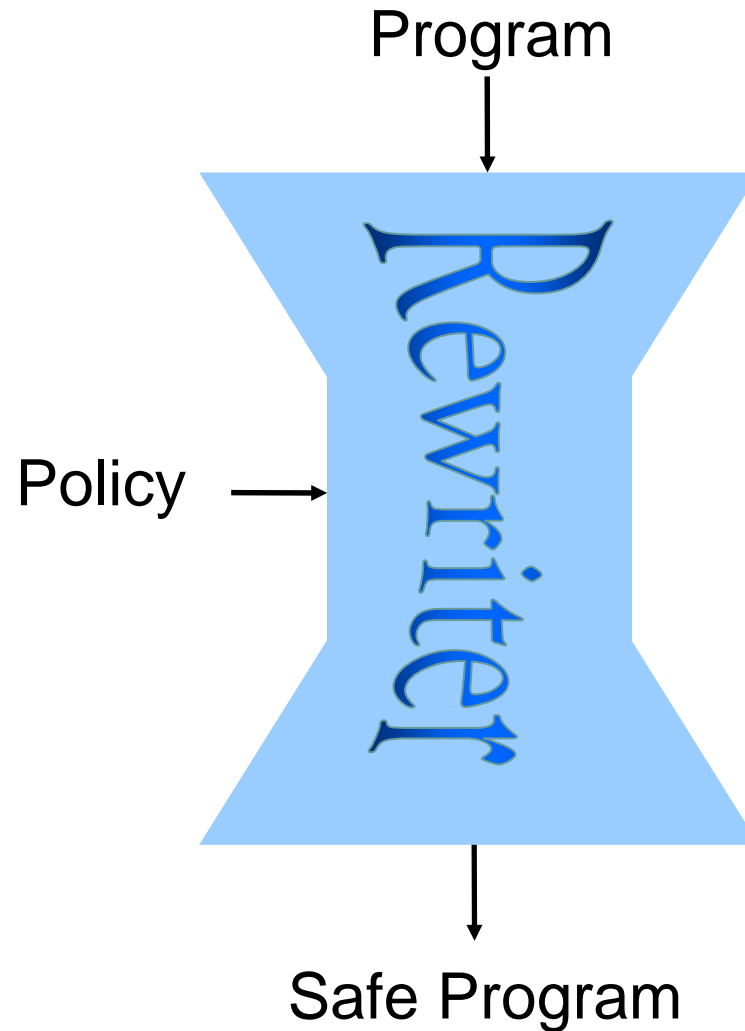
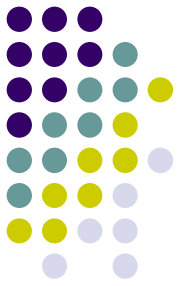
# Run-time Reference Monitors



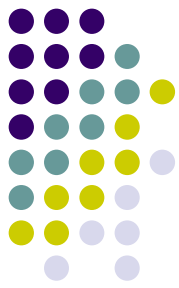
Monitors enforce policies by:

- Interposing between untrusted code and the system executing the untrusted code
- Making sure only legal code is executed

# Program Rewriter

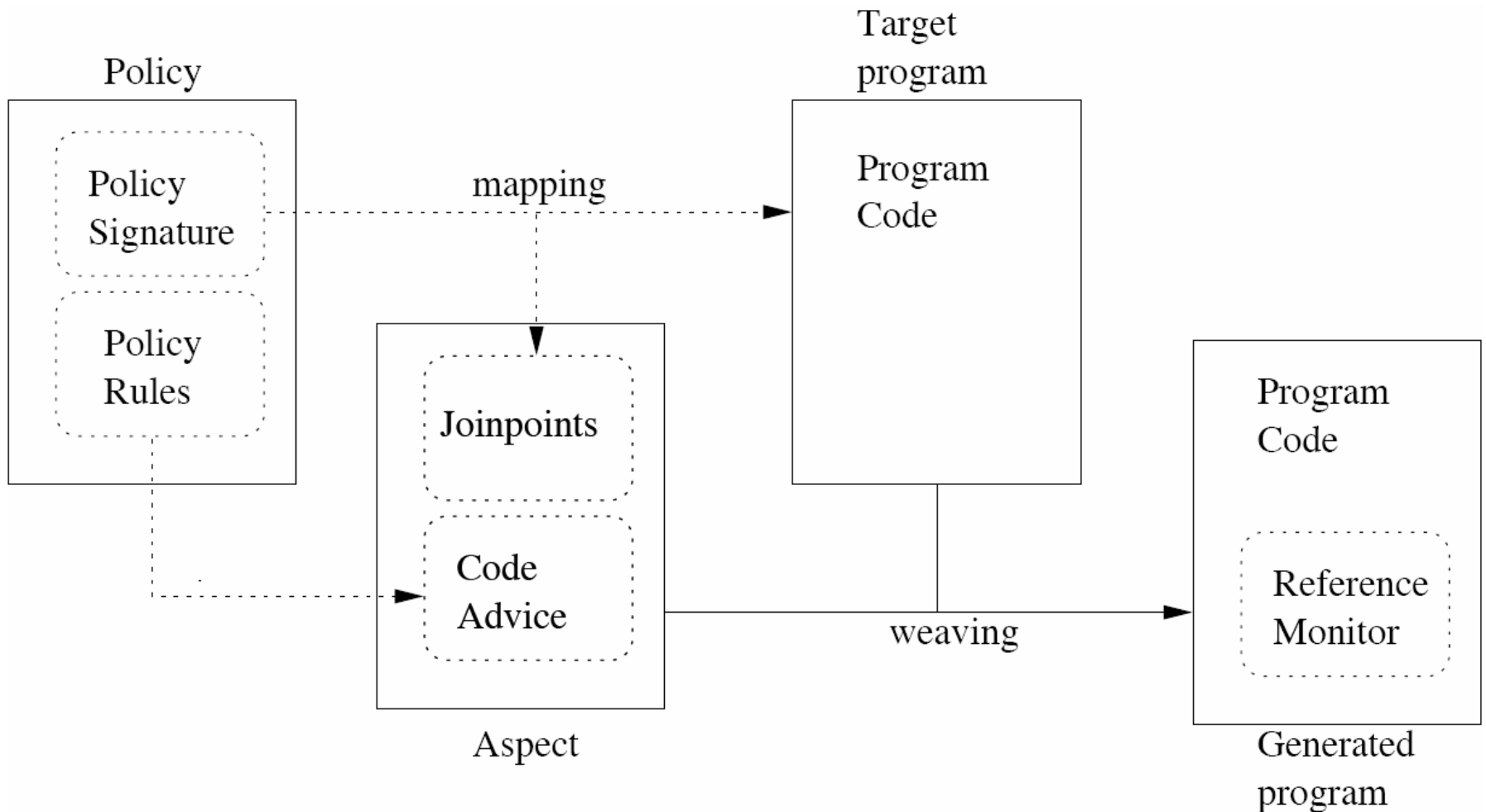


# Can we combine these two approaches with assuring reliable policies ?

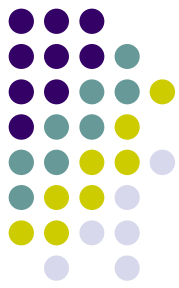


- Run-time Reference Monitors
- Program Rewriters
- Formalize access control policies (guarantee the policies are terminating and complete)

# The answer is “Yes”



# Enter Aspect-Oriented Programming (AOP)



- Why do we adopt AOP???

→ AOP is a good combination example of reference monitor and program rewriter

→ AOP enables **Separation Of Concerns**  
(cross-cutting concerns)



# Cross-cutting concerns

Symptoms:

**Code tangling:**

when a module or code section

is managing several concerns

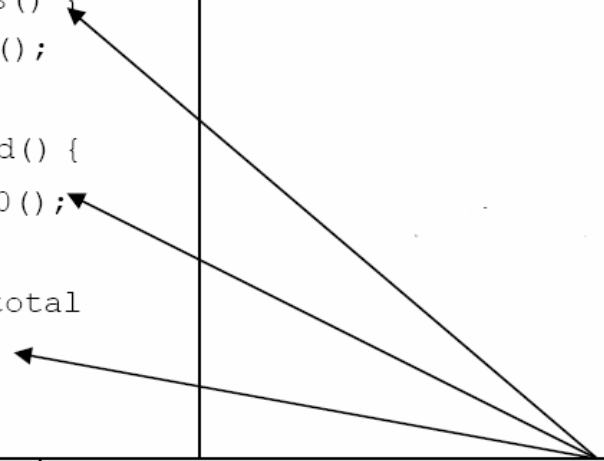
simultaneously

**Code scattering:**

when a concern is spread over many modules and is not well localized and modularized

```
package java.io;
    public class File implements
    java.io.Serializable {
        private String path;
        ...
        public boolean exists() {
            return exists0();
        }
        public boolean canRead() {
            return canRead0();
        }
        ...// 16 methods in total
    }
```

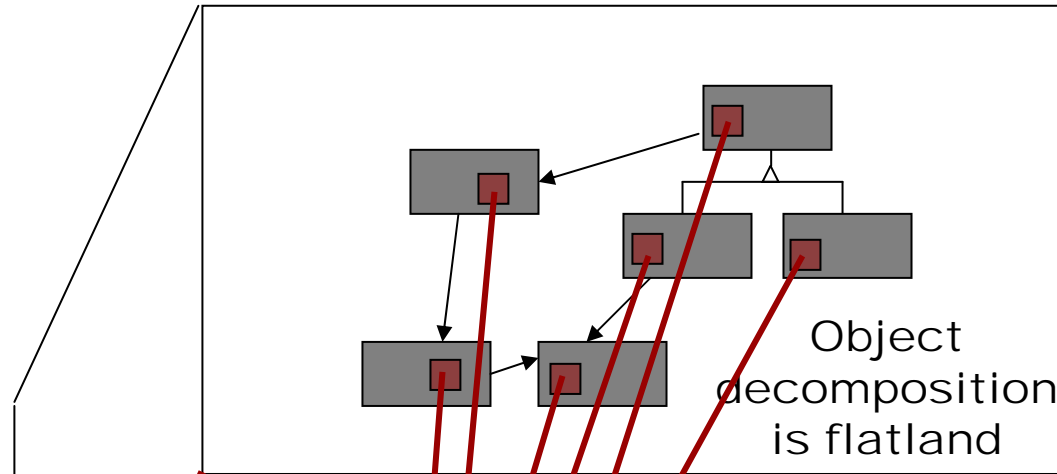
```
//security service call
    SecurityManager security =
        System.getSecurityManager();
    if (security != null) {
        security.checkRead(path); }
```



# Adds a new dimension to software dev.



From presentation by Frank Sauer  
Copyright Technical Resource Connection Inc.



Object decomposition is flatland

A cross-cutting concern is scattered because it is realized in the third dimension

*Aspect*

Aspects are orthogonal to the primary decomposition

*concerns*



# Aspects

- **One Aspect** is the unit of modularity in AOP
- Similar to the *Class* construct in OOP
- Implemented as regular class in Java

```
aspect PolicyAspect {  
  
    private int phase;  
    private User usr;  
    private int paperId;  
  
    pointcut readScoresCut():  
        call(int Paper.readScores(int));  
  
    ....  
  
    before():readScoresCut(){  
        if(!Policy.apply(usr.getId(),paperId,  
            usr.getRole(),phase,true,"readScores")){  
            System.out.println("Access Denied.");  
            System.exit(1);  
        }  
    }  
}
```

**Point-cut:** define a joint-point which point in the execution of a program where an aspect can intervene

**Advice:** piece of action whenever the point cut is executed

# AspectJ



- AspectJ – AOP for Java™
- **Actually, AspectJ is a good weaving tool for inlining reference monitor and program rewriting**

# TOM



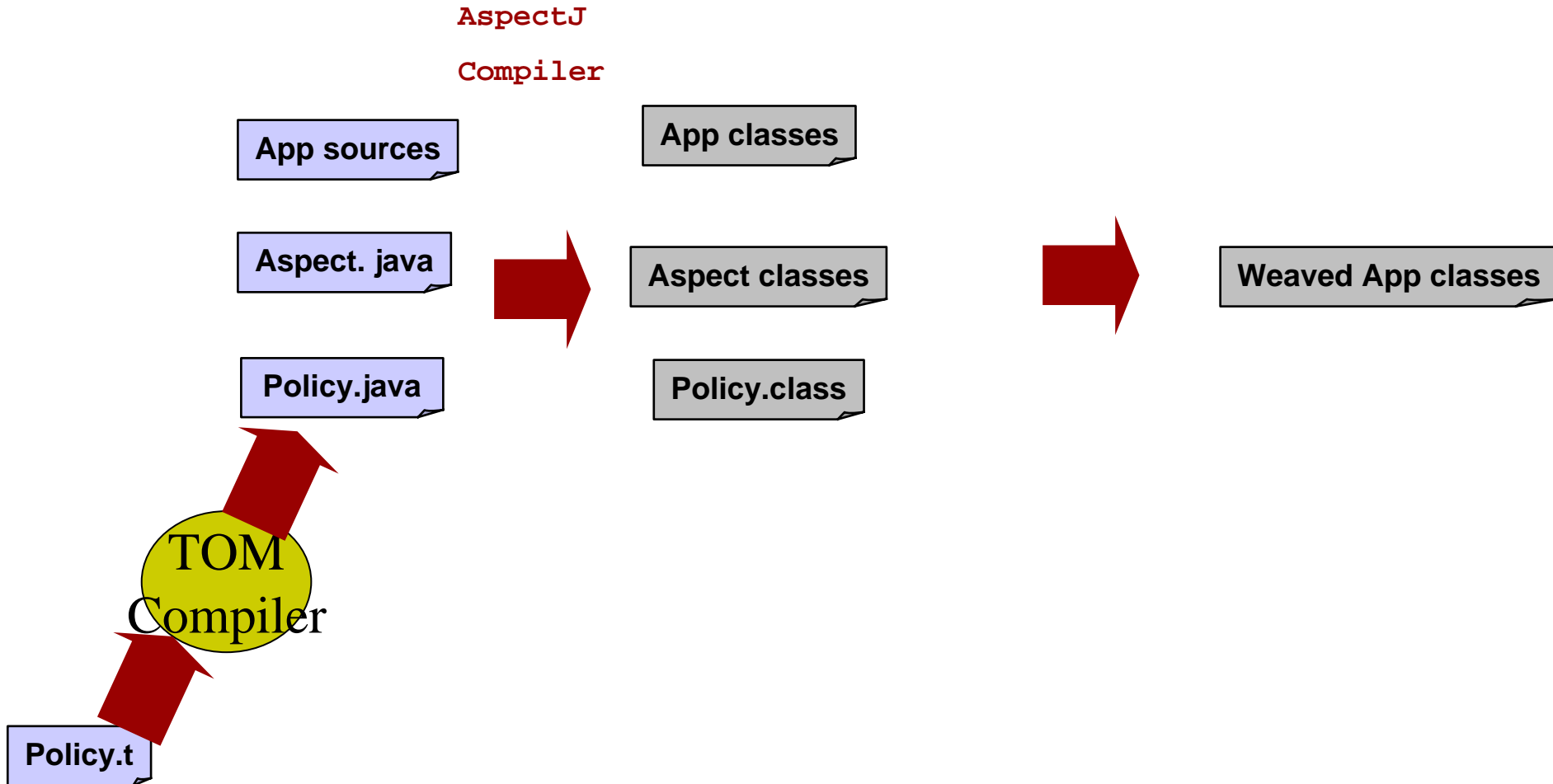
- Tom is a good tool for **formalizing policies** which is able to guarantee the reliability of policies
- Tom is a pattern matching compiler developed at [INRIA](#).
- Tom is a language extension which adds new matching primitives to languages like C and Java.



Since AspectJ functions like **reference monitor** and **program rewriter**, Tom functions as **formalizer of policies**,

**can we connect AspectJ with TOM ?**

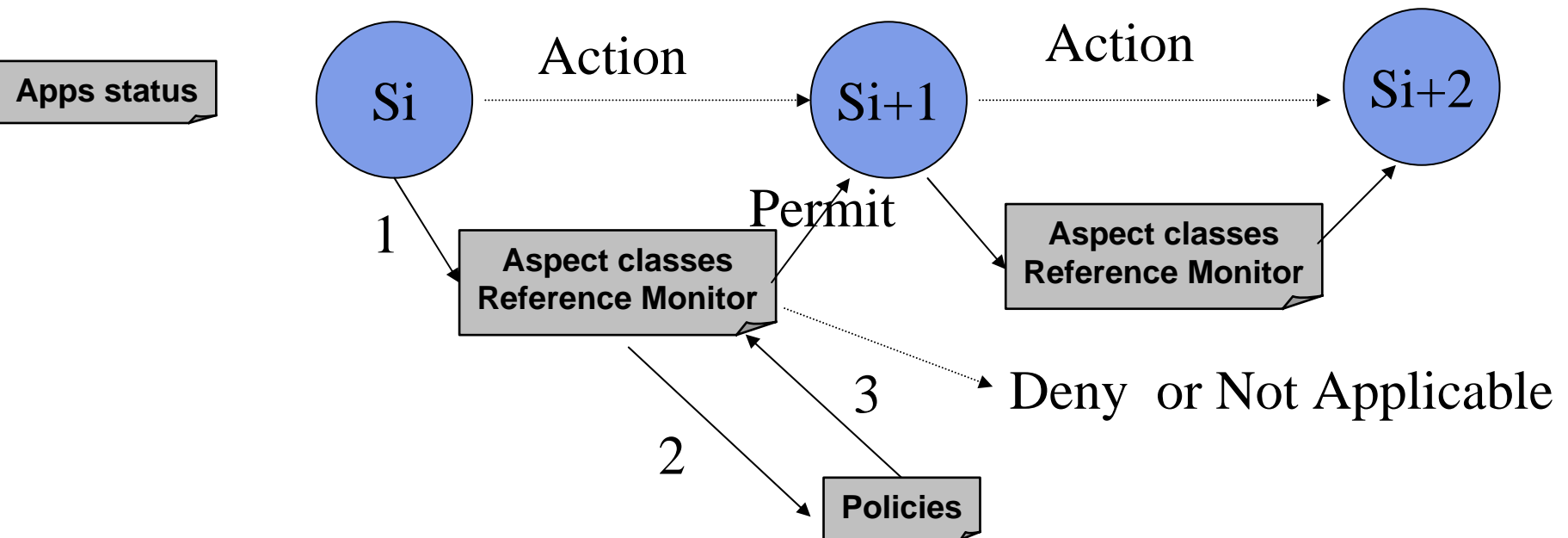
# Our Solutions

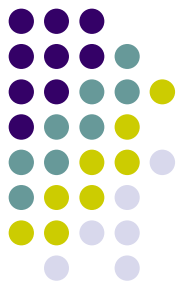




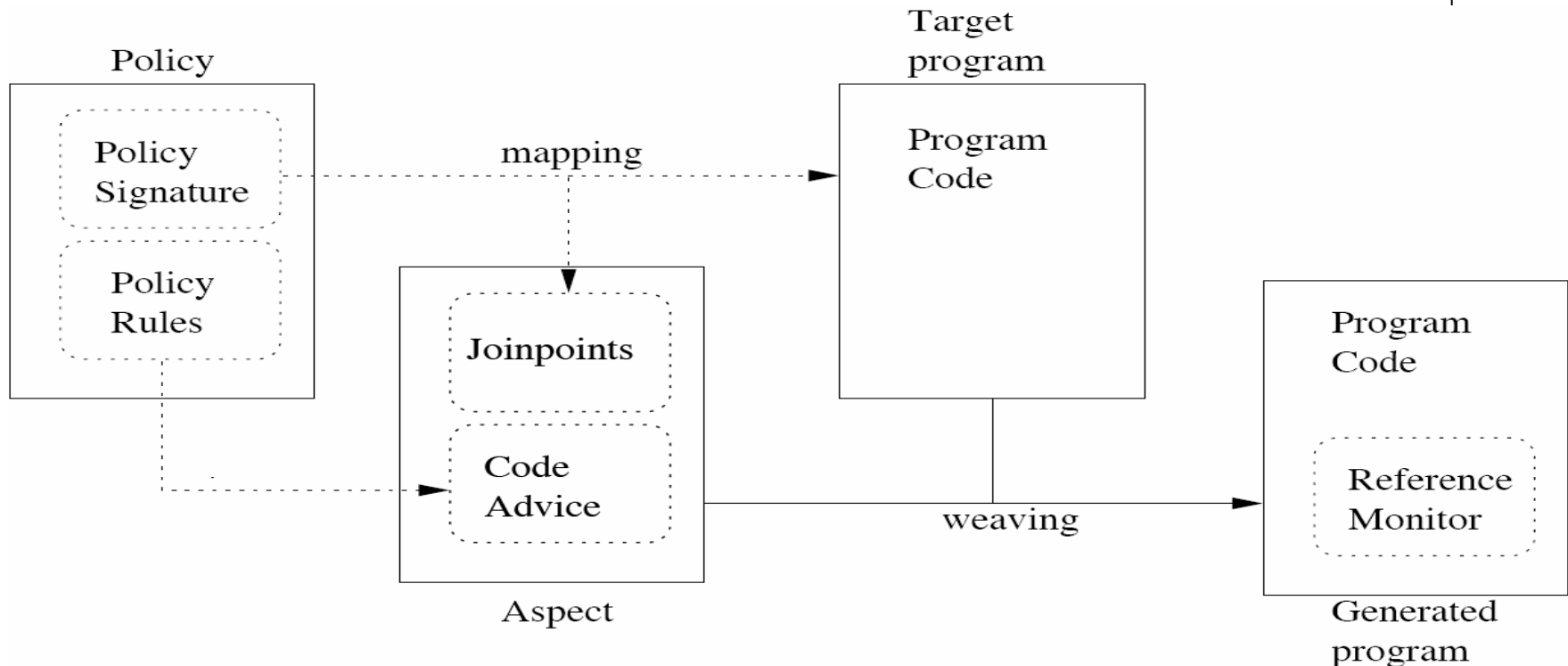
# Execution graph

1. Reference monitor monitors the status and actions of executing target application.
2. Monitor triggers one process to ask the opinion of Policy
3. The policy return the value of “Permit” or “Deny” or “Not Applicable”.





# The Architecture and Steps



- 1. Use Tom to design and formalize policy**
- 2. Create Aspect for defining joint points and code advices**
- 3. Use AspectJ to compile the set of target program, policy and aspect.**

# Example

Step 1 →  
Design and formalize the policy rules by Tom, using Tom to compile \*.t file into \*.java file.

```
public class Policy(  
%gom(  
    ...  
    Decision = permit()  
    | deny()  
    | notApplicable()  
    | aut(r: Request, p:Phase, cnd:Condition)  
Request = q(s:Subject, a:Action, o: Obj)  
Phase = submission() | meeting() | review()  
Action = submitPaper()  
    | readScores()  
    | submitReview()  
    | assignPaper()  
    | addReviewer()  
    ...  
%strategy Rules(){  
    ...  
    aut( q(author(x), submitPaper(), paper(x, y)),  
phase, z) -> {return `deny();}  
    ...  
}  
public static boolean apply(...){  
    ...  
    Strategy policy = `Rules();  
    Decision d = (Decision) policy.visit(`aut(q(s, a, o), p, cnd));  
    ...  
}  
}  
    ...  
}
```

# Strategy Rules Examples of a Conference System



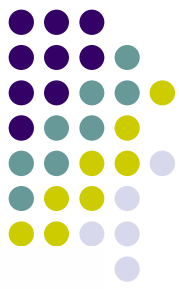
```
aut ( q(author(x), submitPaper(), paper(x, y)),  
      submission(), z) -> {return `permit();}
```

```
aut ( q(author(x), submitPaper(), paper(x, y)),  
phase, z) -> {return `deny();}
```

```
aut (q(author(x), readScores(), paper(x, y)),  
     phase, cond) -> {return `deny();}
```

```
aut (q(reviewer(x), action, p), phase,  
     conflict(x, p)) -> { return `deny(); }
```

```
aut (q(reviewer(x), submitReview(), paper(y, z)),  
     review(), assigned(x, paper(y, z)))  
-> {return `permit();}
```

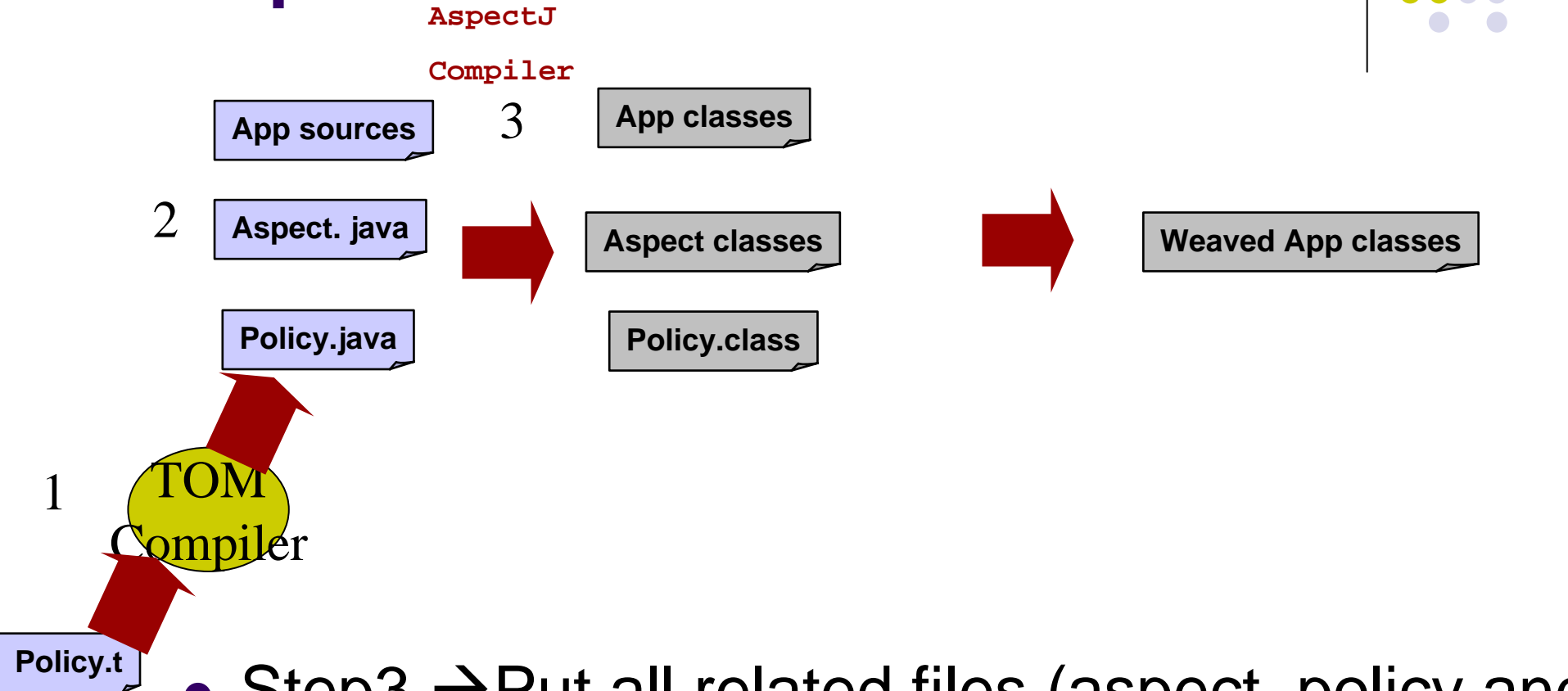


# Example-cont.

- Step2 → Create Aspect by AspectJ

```
aspect PolicyAspect {  
    ...  
    pointcut submitPaperCut (Paper pa) :  
        call (void Conference.submitPaper (Paper))  
        && args (pa) ;  
    ...  
    before (Paper p) : submitPaperCut (p) {  
        paperId=p.getId () ;  
        if (!Policy.apply (usr.getId () , paperId,  
            usr.getRole () , phase , true , "submitPaper" ) ) {  
            System.out.println ("Access Denied." ) ;  
            System.exit (1) ;  
        }  
    }  
}
```

# Final Weaving by AspectJ compiler



- Step3 → Put all related files (aspect, policy and target program into one directory)
- And compile them in ajc(AspectJ compiler)



# Analysis

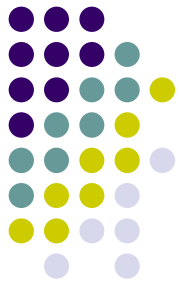
- Employ TOM to formalize policies
- Employ AspectJ as a weaving tool for inlining reference monitors
- Aspects act as a **bridge** between **target applications** and **policies** which are written and compiled in TOM

# Proofs and security: towards better trust



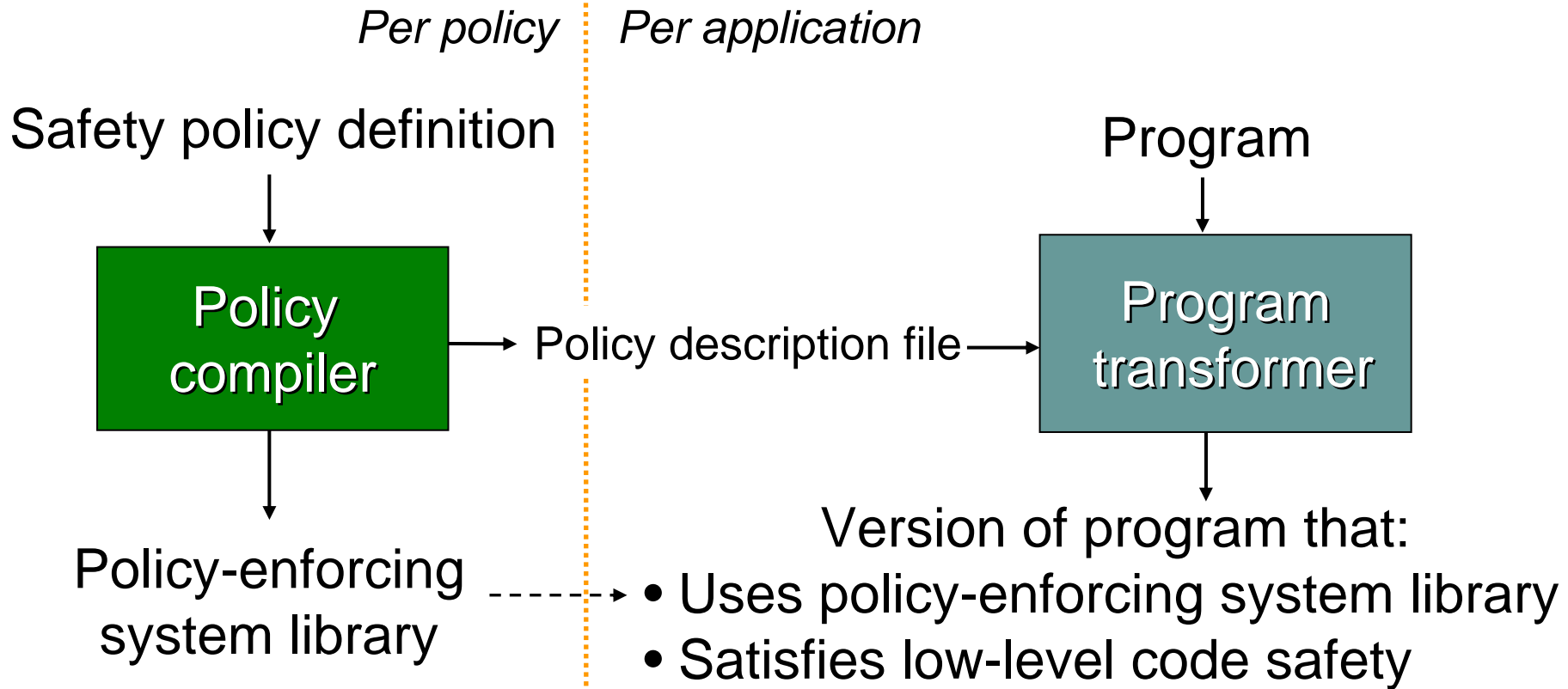
- Security issues
  - Policy reliability (terminating, completeness)
  - Code protection
- Performance
  - AspectJ compilation
  - TOM compilation

# Related work



- Some works are mainly for execution monitors[1,2]
- Some works are for program rewriters[3,4,5]
- Some works are for enforcing policy by AOP[6,7]

# One main previous project --Naccio Architecture (MIT Lab)



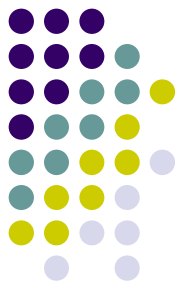
Main difference between it and ours work is the power of describing policy



# Conclusion & Future work

- Our contribution: we proposed a systematic methodology to construct inlined reference monitors for dynamic access control policies.
- And we showed that developers are able to enforce dynamic policies in our modular and flexible way by the connection of AOP tools and TOM.
- Future work: provide more fine tuned analysis tools for policies , and perform a systematic analysis of the application to mechanize the weaving process

# Reference



1. F.B.Schneider. Enforceable security policies. *ACM Trans. Inf. Syst.Secur.*, 3(1):30-50,2000
2. J.Ligatti, L.Bauer and D.Walker. Enforcing non-safety security policies with program monitors. In S.D.C di Vimercati, *ESORICs, lecture notes in Computer Science*, pages 355-373. Springer, 2005
3. U.Erlingsson and F.B.Schneider. Sasi enforcement of security policies: a retrospective. In *NSPW'99*.
4. D.Evans and A.Twyman, Flexible Policy-Directed Code Safety, *IEEE Computer Society*, 1999
5. K.Hamlen. Security Policy Enforcement By automated Program-rewriting. Phd Thesis, Cornell University, 2006.
6. F.Cuppens, N.Cuppens-Boulahia, Availability enforcement by obligations and aspects identification. In *ARES*, 2006
7. E.Song, R.Reddy, R.B.France. Verifiable composition of access control and application features. *SACMAT*, ACM 2005
8. ....

# Questions

- Thank you very much!

