Weaving Rewrite-Based Access Control Policies

Anderson Santana de Oliveira[a], Eric Ke Wang[ab], Claude Kirchner[a], Hélène Kirchner[a]

INRIA & LORIA

The University of Hong Kong

FMSE, 2007

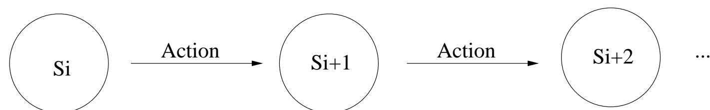
Context

- Access control is a central issue in computer security
- Policies are software artifacts on their own:
  - Policies may be composed of several rule sets
  - Complex conditions on the policy environment
- It is necessary to ensure the compliance of a large system to a given dynamic access control policy
  - Several bugs are related to access control
  - Policy specification and implementation are error prone
  - Policies need to be maintained independently from the application code

Enforcement Mechanism

Execution monitoring

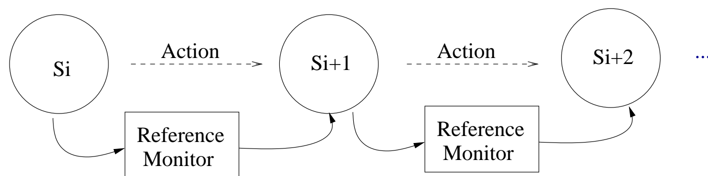
- A reference monitor watches the execution of a target program, and interferes when it is about to violate the security policy
- Monitors can be “inlined” in the application code
- This process can be automated for low level, static, policies



Enforcement Mechanism

Execution monitoring

- A reference monitor watches the execution of a target program, and interferes when it is about to violate the security policy
- Monitors can be “inlined” in the application code
- This process can be automated for low level, static, policies



Objectives

In this work we provide a methodology for building reference monitors for expressive policies

- to adopt a formal model for access control policies using term rewriting with strategies
- to help developers to enforce dynamic policies in a modular and flexible way
- to be able to reuse policies independently specified in different context or applications

Outline

1. Context/Motivation
2. Expressive Rewrite-Based Access Control
3. Weaving Policies
4. Related Work
5. Summary and Future Works

Example - A conference policy

Authors can submit papers during the submission phase.

$$aut(q(author(x), \textbf{submitPaper}, paper(x, z)), \textbf{submission}, cnd) \rightarrow \textbf{permit}$$

$$aut(q(author(x), \textbf{readScores}, p), phase, cnd) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \textbf{notApp}$$

Example - A conference policy

Authors can submit papers during the submission phase.

$$aut(q(author(x), \mathbf{submitPaper}, paper(x, z)), \mathbf{submission}, cnd) \rightarrow \mathbf{permit}$$

$$aut(q(author(x), \mathbf{readScores}, p), phase, cnd) \rightarrow \mathbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \mathbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \mathbf{notApp}$$

Example - A conference policy

Authors may never read scores for a paper.

$$aut(q(author(x), \textbf{submitPaper}, paper(x, z)), \textbf{submission}, cnd) \rightarrow \textbf{permit}$$

$$aut(q(author(x), \textbf{readScores}, p), phase, cnd) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \textbf{notApp}$$

Example - A conference policy

A reviewer may never perform an action over a paper if he is conflicted with that paper.

$$aut(q(author(x), \textbf{submitPaper}, paper(x, z)), \textbf{submission}, cnd) \rightarrow \textbf{permit}$$

$$aut(q(author(x), \textbf{readScores}, p), phase, cnd) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow$$
$$aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \textbf{notApp}$$

Example - A conference policy

A reviewer is conflicted with a paper if he is an author for that paper.

$$aut(q(author(x), \mathbf{submitPaper}, paper(x, z)), \mathbf{submission}, cnd) \rightarrow \mathbf{permit}$$

$$aut(q(author(x), \mathbf{readScores}, p), phase, cnd) \rightarrow \mathbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \mathbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \mathbf{notApp}$$

Example - A conference policy

$$aut(q(author(x), \mathbf{submitPaper}, paper(x, z)), \mathbf{submission}, cnd) \rightarrow \mathbf{permit}$$

$$aut(q(author(x), \mathbf{readScores}, p), phase, cnd) \rightarrow \mathbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \mathbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow$$
$$aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \mathbf{notApp}$$

Rewrite-Based Access Control

- Term rewriting provides a clear semantics to access control and a framework to formulate its formal properties
  - consistency, completeness, termination
  - An uniform semantics for policy combiners relying on rewriting strategies
  - The language allows us to handle a wide range of dynamic policies

D. J. Dougherty, C. Kirchner, H. Kirchner, and A. Santana de Oliveira. **Modular access control via strategic rewriting.** *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2007.

Rewrite-Based Access Control

- Term rewriting provides a clear semantics to access control and a framework to formulate its formal properties
  - consistency, completeness, termination
  - An uniform semantics for policy combiners relying on rewriting strategies
  - The language allows us to handle a wide range of dynamic policies

D. J. Dougherty, C. Kirchner, H. Kirchner, and A. Santana de Oliveira. **Modular access control via strategic rewriting.** *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2007.

Rewrite-Based Access Control

- Term rewriting provides a clear semantics to access control and a framework to formulate its formal properties
  - consistency, completeness, termination
  - An uniform semantics for policy combiners relying on rewriting strategies
  - The language allows us to handle a wide range of dynamic policies

D. J. Dougherty, C. Kirchner, H. Kirchner, and A. Santana de Oliveira. **Modular access control via strategic rewriting.** *ESORICS*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593. Springer, 2007.

Rewriting-Based Policies.

Definition (Security Policy)

A policy is defined by :

1. a signature – for defining structure of the data-structure for the policy environment;
2. a non-empty set of constants – which represent decisions;
3. a set of rewrite rules – to govern the behavior of the policy
4. a set of terms – that represent access requests
5. a rewrite strategy – that controls rule application.

Rewriting-Based Policies.

Definition (Security Policy)

A policy is defined by :

1. a signature – for defining structure of the data-structure for the policy environment;
2. a non-empty set of constants – which represent decisions;
3. a set of rewrite rules – to govern the behavior of the policy
4. a set of terms – that represent access requests
5. a rewrite strategy – that controls rule application.

Rewriting-Based Policies.

Definition (Security Policy)

A policy is defined by :

1. a signature – for defining structure of the data-structure for the policy environment;
2. a non-empty set of constants – which represent decisions;
3. a set of rewrite rules – to govern the behavior of the policy
4. a set of terms – that represent access requests
5. a rewrite strategy – that controls rule application.

Rewriting-Based Policies.

Definition (Security Policy)

A policy is defined by :

1. a signature – for defining structure of the data-structure for the policy environment;
2. a non-empty set of constants – which represent decisions;
3. a set of rewrite rules – to govern the behavior of the policy
4. a set of terms – that represent access requests
5. a rewrite strategy – that controls rule application.

Rewriting-Based Policies.

Definition (Security Policy)

A policy is defined by :

1. a signature – for defining structure of the data-structure for the policy environment;
2. a non-empty set of constants – which represent decisions;
3. a set of rewrite rules – to govern the behavior of the policy
4. a set of terms – that represent access requests
5. a rewrite strategy – that controls rule application.

Example - A conference policy

$$aut(q(author(x), \textbf{submitPaper}, paper(x, z)), \textbf{submission}, cnd) \rightarrow \textbf{permit}$$

$$aut(q(author(x), \textbf{readScores}, p), phase, cnd) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, p), phase, conflict(x, p)) \rightarrow \textbf{deny}$$

$$aut(q(reviewer(x), action, paper(x, z)), phase, cnd) \rightarrow aut(q(reviewer(x), action, paper(x, z)), phase, conflict(x, paper(x, z)))$$

$$aut(a, b, c) \rightarrow \textbf{notApp}$$

- Tom (`http://tom.loria.fr`)
- Extends Java with pattern matching, rewriting, and strategies
  - %**match** allows discriminating among a list of patterns.
  - ‘ (backquote construct) is used to build terms from Java values.
  - %**gom** provides a language to define tree-like data structures
  - %**strategy** groups rules for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. **Tom: Piggybacking rewriting on java.** *RTA 2007*, vol. 4533 of *LNCS*, pages 36–47. Springer.

- Tom (`http://tom.loria.fr`)
- Extends Java with pattern matching, rewriting, and strategies
  - **%match** allows discriminating among a list of patterns.
  - ` (backquote construct) is used to build terms from Java values.
  - **%gom** provides a language to define tree-like data structures
  - **%strategy** groups rules for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles.
Tom: Piggybacking rewriting on java. *RTA 2007*, vol. 4533 of *LNCS*, pages 36–47. Springer.

- Tom (`http://tom.loria.fr`)
- Extends Java with pattern matching, rewriting, and strategies
  - %**match** allows discriminating among a list of patterns.
  - ` (backquote construct) is used to build terms from Java values.
  - %**gom** provides a language to define tree-like data structures
  - %**strategy** groups rules for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. **Tom: Piggybacking rewriting on java.** *RTA 2007*, vol. 4533 of *LNCS*, pages 36–47. Springer.

Tom

Implementing Rewriting in Java

- Tom (`http://tom.loria.fr`)
- Extends Java with pattern matching, rewriting, and strategies
  - %**match** allows discriminating among a list of patterns.
  - ‘ (backquote construct) is used to build terms from Java values.
  - %**gom** provides a language to define tree-like data structures
  - %**strategy** groups rules for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. **Tom: Piggybacking rewriting on java.** *RTA 2007*, vol. 4533 of *LNCS*, pages 36–47. Springer.

- Tom (`http://tom.loria.fr`)
- Extends Java with pattern matching, rewriting, and strategies
  - %**match** allows discriminating among a list of patterns.
  - ‘ (backquote construct) is used to build terms from Java values.
  - %**gom** provides a language to define tree-like data structures
  - %**strategy** groups rules for constructing more complex strategies, e.g. *innermost, outermost, top down, etc.*.

E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles. **Tom: Piggybacking rewriting on java.** *RTA 2007*, vol. 4533 of *LNCS*, pages 36–47. Springer.

Structure of a policy in Tom

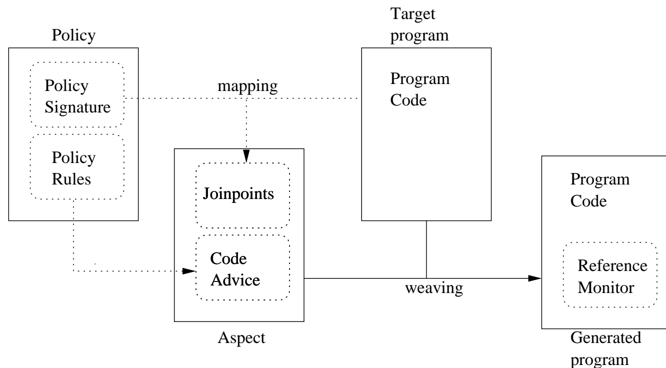
```
public class Policy {

 %gom{
     ...
     Request = q(s:Subject, a:Action, o:Obj)
     Phase = submission()
             | meeting()
             | review()
     Action = submitPaper()| readScores()...
     Decision = permit()
        | deny()
        | aut(r: Request, p:Phase)
     ...
 }
   ...
 %strategy Rules() {
   ...
  aut(q(author(x),submitPaper(),paper(x,y)),
   submission(),cnd) -> {return `permit();}
   ...
 }
  ...
 public static boolean apply(...){
     ...
     Strategy policy =`Innermost(Rules());
     ...
     Decision d = (Decision)policy.visit(q1);
     ...
 }
  ...
}
```

Aspect Oriented Programming

- Separation of “crosscutting” concerns, e.g. access control
- Encapsulation of behaviors that affect multiple classes in aspects
- Basic concepts: *pointcuts* and and *code advice*
- *Weaving*: compilation process that matches the pointcuts against the actual code and inserts the code advice *before, after or around* a joinpoint
- Several implementations exist, a popular one is AspectJ

System Architecture



Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules

Example

```
aspect PolicyAspect {
...
  private int phase;
...
  after(int cp): set(int Conference.currentPhase) && args(cp){
     phase=cp;
  }
...
}
```

Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules

Example

```
aspect PolicyAspect {
...
  private int phase;
...
  after(int cp): set(int Conference.currentPhase) && args(cp){
      phase=cp;
  }
...
}
```

Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules

Example

```
pointcut submitReviewCut(int objId, int revId):
    call(void Conference.submitReview( int, int, int)) && args(objId, revId, *);
```

Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules

Example

```
pointcut submitReviewCut(int objId, int revId):
   call(void Conference.submitReview( int,int,int)) && args(objId,revId,*);
```

Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules: AspectJ compiler

Example

```
before(Paper p):submitPaperCut(p){
  ...
  if( Policy.apply(usr.getId(),...,phase) == Deny){
   System.err.println("Access␣Denied.");
   throw new AccessControlException();
  }
}
```

Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules: AspectJ compiler

Example

```
before(Paper p):submitPaperCut(p){
  ...
  if( Policy.apply(usr.getId(),...,phase) == Deny){
   System.err.println("Access␣Denied.");
   throw new AccessControlException();
  }
}
```

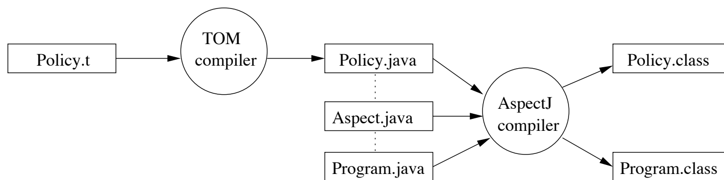
Aspects for Access Control

1. Capture the policy environment
2. Identify access requests
3. Give the control to the reference monitor: advice code
4. Weave the policy rules: AspectJ compiler

Example

```
before(Paper p):submitPaperCut(p){
  ...
  if( Policy.apply(usr.getId(),...,phase) == Deny){
   System.err.println("Access␣Denied.");
   throw new AccessControlException();
  }
}
```

Weaving rules using AspectJ and Tom



Considerations

- Overhead: balanced with modularity and better security
- Proving policy properties increases trust: consistency, completeness, termination
- Policies also need to be protected against malicious or unaware developers

Related works

- Static analyzers – e.g. Java bytecode verifier
- Formalization of run-time *execution monitors* [Schneider, 2000, Ligatti et al., 2005]
- Program rewriters(at compile or load time) [Erlingsson and Schneider, 2000, Evans and Twyman, 1999, Bauer et al., 2005]
  - Other approaches that use AOP for policy enforcement [Song et al., 2005, Cuppens et al., 2006]

Summary and Future Works

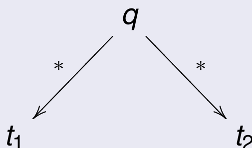
- We presented a methodology to weave expressive rewrite-based policies into existing Java programs
- Developers can formally specify policies and reuse them in a flexible enforcement scheme
- We have implemented a number of examples to test the robustness of the approach
- Future works
  - To automate the generation of pointcuts
  - To provide more fine-tuned analysis for policies

Consistency.

Definition (Consistency)

A security policy is consistent if for every query evaluation, at most one decision is computed.

Confluence



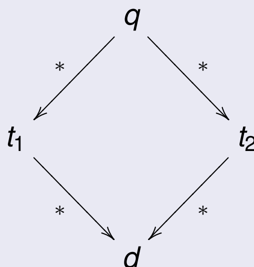
Return

Consistency.

Definition (Consistency)

A security policy is consistent if for every query evaluation, at most one decision is computed.

Confluence



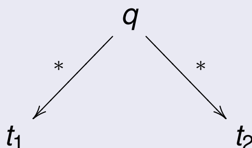
Return

Consistency.

Definition (Consistency)

A security policy is consistent if for every query evaluation, at most one decision is computed.

Confluence



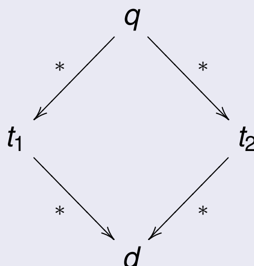
Return

Consistency.

Definition (Consistency)

A security policy is consistent if for every query evaluation, at most one decision is computed.

Confluence



◂ Return

Termination.

Definition (Termination)

A security policy is terminating if every request evaluation is finite.

- Powerful proof techniques: dependency pairs, recursive path orderings, etc
- Several automated tools for proving termination are available: AProVE, TTT, CiME, etc.

Return

Termination.

Definition (Termination)

A security policy is terminating if every request evaluation is finite.

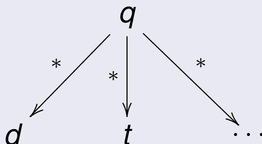
- Powerful proof techniques: dependency pairs, recursive path orderings, etc
- Several automated tools for proving termination are available: AProVE, TTT, CiME, etc.

Return

Completeness.

Definition (Completeness)

A security policy is complete if for every request its evaluation returns an access decision.

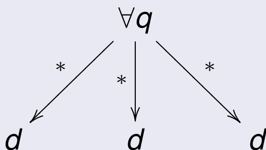


Return

Completeness.

Definition (Completeness)

A security policy is complete if for every request its evaluation returns an access decision.

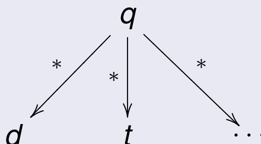


Return

Completeness.

Definition (Completeness)

A security policy is complete if for every request its evaluation returns an access decision.

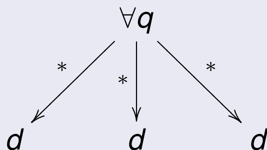


Return

Completeness.

Definition (Completeness)

A security policy is complete if for every request its evaluation returns an access decision.



Return

For Further Information I

Bauer, L., Ligatti, J., and Walker, D. (2005).
Composing security policies with polymer.
In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 305–314, New York, NY, USA. ACM Press.

Cuppens, F., Cuppens-Boulahia, N., and Ramard, T. (2006).
Availability enforcement by obligations and aspects identification.
In *ARES*, pages 229–239. IEEE Computer Society.

Erlingsson, U. and Schneider, F. B. (2000).
Sasi enforcement of security policies: a retrospective.
In *NSPW '99: Proceedings of the 1999 workshop on New security paradigms*, pages 87–95, New York, NY, USA. ACM Press.

Evans, D. and Twyman, A., editors (1999).
Flexible Policy-Directed Code Safety, IEEE Symposium on Security and Privacy, 1999. IEEE Computer Society.

For Further Information II

Ligatti, J., Bauer, L., and Walker, D. (2005).
Enforcing non-safety security policies with program monitors.
In di Vimercati, S. D. C., Syverson, P. F., and Gollmann, D., editors, *ESORICS*, volume 3679 of *Lecture Notes in Computer Science*, pages 355–373. Springer.

Schneider, F. B. (2000).
Enforceable security policies.
ACM Trans. Inf. Syst. Secur., 3(1):30–50.

Song, E., Reddy, R., France, R. B., Ray, I., Georg, G., and Alexander, R. (2005).
Verifiable composition of access control and application features.
In Ferrari, E. and Ahn, G.-J., editors, *SACMAT*, pages 120–129. ACM.