A Graph-based Approach to Specifying Security Policies

James Hoagland

Karl Levitt

Raju Pandey

Computer Security Research Laboratory

Department of Computer Science

University of California, Davis

{hoagland,levitt,pandey}@cs.ucdavis.edu

Outline

- ❏ Introduction
- ❏ System model
- ❏ Graph-based constraint language
- ❏ Composing policies
- ❏ Future work

Security Policies

Security policies:

- ❏ define the security requirements for a system
- ❏ are the manifestations of the security needs of an organization
- ❏ indicate what security-relevant behavior is allowed to occur in certain situations
- ❏ consist of a set of constraints

Approach

Goals of work:

- ❏ an easy way to *formally* specify security policies
  - for enforcing policies in a uniform way
  - to formally reason about policies
- ❏ to be able to specify many policies using this method
  - for greater potential usefulness

Approach:

- ❏ specify policies in a formal language
- ❏ language is based on graphs
  - nodes represent entities
  - edges represent some relationship between entities

System Description

System model:

- ❏ object-oriented approach to describing the security-relevant behavior
- ❏ description consists of a set of classes for the types of entities in the system
- ❏ classes contain:
  - attributes
  - methods

Example system description:

```
class Process {
    clearance: Level
    pid: integer
    spawn()
}
```

```
class File {
    security_level: Level
    read(length:int)
    write(data: string)
}
```


James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

System Instance

The system instance is the state of the system at some moment.

A system instance consists of:

- ❏ a set of class instances (objects) with attribute values
- ❏ a set of method invocations with parameter values

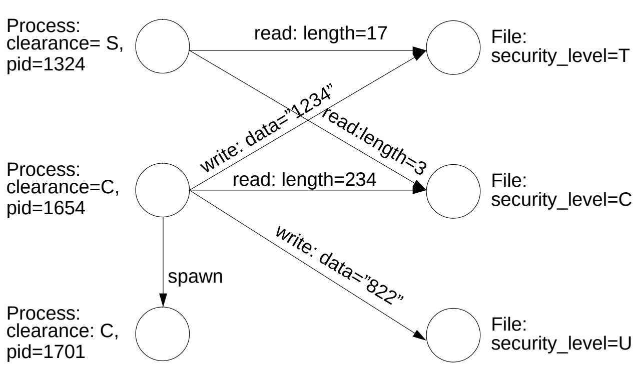
A more formal specification of the system is work in progress.

System instance graph:

- ❏ a way to present a system instance
- ❏ a node for each object
- ❏ an edge for each method invocation
  - from node representing invoking object
  - to node representing invoked object

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

System Instance Graph Example



James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Graph-based Approach

The graph-based approach to specifying security policies

- ❏ policies consist of a set of constraints
- ❏ each constraint is represented by a graph
- ❏ constraints get checked against the system
- ❏ the constraint graphs depict
  - when the policies apply (the *antecedent*)
  - what the requirement is (the *consequent*)

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Antecedent and Consequent Semantics

When applying a policy to a system instance:

1. If the antecedent applies:
2. check the consequent to see if the policy was upheld

For the following:

- let S be a system instance
- let P be a policy in effect on that system
- let $A_p(s)$ be true iff s satisfies the antecedent of p
- let $C_p(s)$ be true iff s satisfies the consequent of p

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Antecedent and Consequent Semantics [2]

- A policy is relevant to a system instance if the antecedent is satisfied.
  - **relevant(P,S)**= $A_p(S)$
- A policy is upheld on a system instance if it is relevant to the instance and if its consequent is satisfied.
  - **upheld(P,S)**= relevant(P,S) $\wedge$ $C_P(S)$ = $A_P(S) \wedge C_P(S)$
- A policy is not violated if it either is not relevant or is upheld.
  - **no_violation(P,S)**= ¬relevant(P,S) $\vee$ upheld(P,S) =

  $\neg A_P(S) \vee (A_P(S) \wedge C_P(S))= (\neg A_P(S) \vee A_p(S)) \wedge (\neg A_P(S) \vee C_P(S))=$

  $A_P(S) \Rightarrow C_P(S)$
- A policy is considered to be violated if it is relevant but its consequent is not satisfied.
  - **violation(P,S)**= relevant(P,S) $\wedge \neg C_P(S) = \neg(\neg A_P(S) \vee C_P(S))$ =

  $\neg(A_P(S) \Rightarrow C_P(S))$ = ¬no_violation(P,S)

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Graph-based Constraint Language

Language has nodes and edges:

- nodes are a pattern for objects of a particular class
- edges are a pattern for method invocations
  - source node is the invoking object
  - destination node is the invoked object

Nodes and edges have annotations:

- antecedent and consequent boolean expressions
- these predicates further restrict what objects and method invocations can match the constraint
- predates can refer to:
  - object attribute values (nodes) or method parameter values (edges)
  - variables (bound like in Prolog, on first use)

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Graph-based Constraint Language [2]

Satisfying the antecedent results in bindings of:

- nodes to system instance objects
- edges to method invocations from the system instance
- variables to values

Formal semantics for evaluating antecedent and consequent expressions is work in progress.

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Example Policy Specification Using Graphs

Example: a process with a certain clearance level can only read a file with lower or equal security level

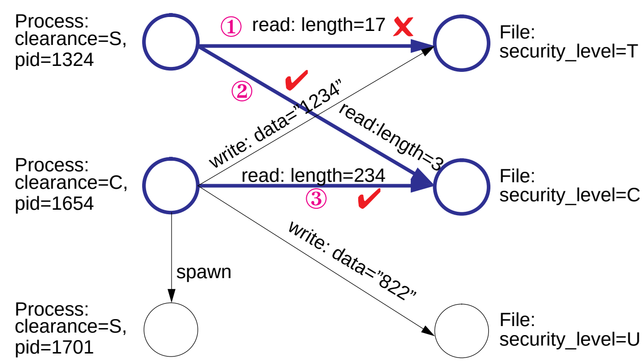
Note:

- **blue parts** are the antecedent or trigger (when the policy applies)
- red parts are the consequent or requirement (what must then be the case)

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Example Policy Applied

The policy applied to the example system graph: 3 applications, 1 violation



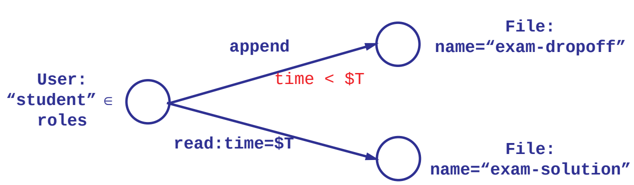
James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Exam Scenario Constraint Graph

An online exam is to be given for a class at a university. Part of the design is:

- completed exams are to be dropped off in a file
- solutions are to be available electronically to students after they turn in their exam, but not before

Policy: If a student appends to the exam dropoff file and reads the exam solution file, then the time of the append must be earlier than the time of the read.

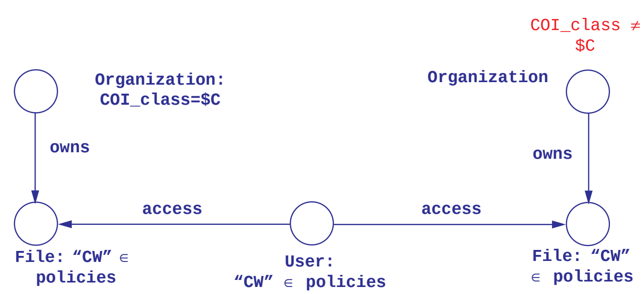


Key: **antecedent is blue**; consequent is red

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Chinese Wall A/C Constraint Graph

Chinese Wall: If a consultant has accessed protected data from two companies, then one company cannot be in the same conflict of interest class as the other.



Key: **antecedent is blue**; consequent is red

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Advantages to this Approach

It is expressive:

- ❑ language is independent of the semantics of the entities and relationships
  - nodes are independent of the specific entity
  - edges can represent any relationship

It is formal:

- ❑ can reason about policies expressed in the language
- ❑ can enforce all policies in the same way

It is separate from the system model

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Composing Policies

Composed policy:

- ❑ the policy consisting of the constraints enforced by two or more policies that are in effect
- ❑ semantics of policy composition:
  - a policy violation if and only if system instance violates any of the set of policies
- ❑ S is a system instance and P is a set of policies:
  - violation(P,S)= $\exists$ p $\in$ P: violation(p,S)
  - no_violation(P,S)= $\forall$ p $\in$ P: no_violation(p,S)

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Policy Contradiction

However, having a set of policies in effect may lead to contradictions.

- ❑ two policies contradict if, for some system instance, one indicates violation and the other indicates no violation
- ❑ for policies expressed in graph language
  - antecedents overlap, and
  - consequents produce opposite results for some of the overlap

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Future Work

Formally develop constraint language

- ❑ define system model formally
- ❑ fully define semantics of the language
- ❑ characterize the language's ability to express policies
  - compare the expressiveness with other methods of formally specifying security policies

Policy violation detection

- ❑ design and implement policy enforcement mechanism for some environment (Java?)

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu

Future Work [2]

Composition of policies

- ❏ investigate different ways to compose policies
  - composition semantics as presented
  - prioritized policies
- ❏ for arbitrary policies specified in the graph constraint language, determine
  - whether two policies are equivalent
  - whether one policy is subsumed by another
  - under what circumstances the policies apply at the same time
  - under what circumstances the policies conflict

James A. Hoagland
Department of Computer Science
University of California, Davis
hoagland@cs.ucdavis.edu