# Evaluating access control policies through model-checking

Nan Zhang

Mark Ryan

Dimitar Guelev

School of Computer Science, University of Birmingham

Section of Logic, Institute of Mathematics and Informatics, Bulgaria

# The motivation of this work

- Security challenges caused by ever-increasing interconnectivity of computers, phones, PDAs, via wireless, *ad-hoc*, and p2p networks
- Access control is part of the toolset used to meet the challenges
- The reliability of an access control system depends on the fitness of the policy it adopts
- This work focuses on the analysis of access control policies
  - in which permissions are themselves data subject to permissions

# Motivation - continued


Our analysis detects security holes in access control policies caused by:

- Interactions of rules: It is not enough to know that whether a single rule behaves correctly, but all rules, working together, behave correctly.
- Co-operation between agents: It is not enough to know what an agent can do by himself, but what a set of agents can achieve through co-operations.
- Multi-step actions: It is not enough to know what an agent can do in a single step of action, but what she can achieve through a sequence of actions.
  - Especially valid when agents can change permissions to give themselves or others privileges.

# An example

Consider a **conference paper review system**.
Agents – authors, reviewers, PC members,…
Papers –

1. The chair of the PC assigns papers to PC members for reviewing.
2. PC member a can read PC member b's review for a paper p provided p is not assigned to a.
3. If two PC members, a and b, are both assigned paper p for reviewing, a can read b's review for p if a has already submitted its own review for p. This is also true for b.
4. Having been assigned a paper p, PC member a can give up being reviewer for p before the reviewing is finished.

p – a paper
a, b – two PC members
c – the chair

Scenario:
1, c assigns p to both a and b for reviewing.
(permitted by rule 1)

2, Before a submits his review for p, he resigns being reviewer for p.
(permitted by rule 4)

3, a reads b's review for p.
(permitted by rule 2)

4, c assigns p to a for reviewing again.
(permitted by rule 1)

# Our solution -- model-checking

The advantages of using model-checking:

- As always, *analysis* aids *understanding*
- Temporal reasoning is suitable for exploring possible consequences of multi-step actions

We consider coalitions of agents instead of a single agent when the checking is performed. Therefore, potential attacks caused by co-operations between agents may also be uncovered.

# The *RW* framework

- The RW language – our input language, used to describe access control systems, and the properties to be verified
- The RW model-checking algorithm – the algorithm decides whether a model satisfies a property
- AcPeg – our model-checker, which implements the algorithm.
  - It also converts the policies defined in RW to policies in XACML, from which Java classes implementing the AC can be derived.

# The *RW* formalism

Access control systems S are tuples of the form (Σ, P, r, w):

Σ – a set of agents

P – a set of propositional variables, representing data, relations between data and even permissions of the system; L(P) denotes the set of propositional formulas built from the variables in P

r, w – mappings of type: P X Σ → L(P).
r(p,a) is a formula over P which expresses the circumstances under which a can read p.

States of S are valuations of the variables in P. Agent a is allowed to read (overwrite) variable p in state s iff s satisfies r(p,a) (w(p,a)).

# Part of the *RW* program for the example

```
AccessControlSystem Conference

    Class  Paper;

    Predicate pcmember(agent: Agent), chair(agent: Agent)!,
       reviewer(paper: Paper, agent: Agent), submitted(paper:
       Paper, agent: Agent), review(paper: Paper, agent: Agent);


    chair(a){   read: true;  }

    pcmember(a){   read : true;
       write : (chair(user) | (pcmember(a) & a=user)); }

    review(p, a){
       read : submitted(p, a) & pcmember(user) &
       ((reviewer(p, user) -> submitted(p, user)) | user=a);
       write : user=a & reviewer(p,a) & ~submitted(p, user);}

    . . .

End
```

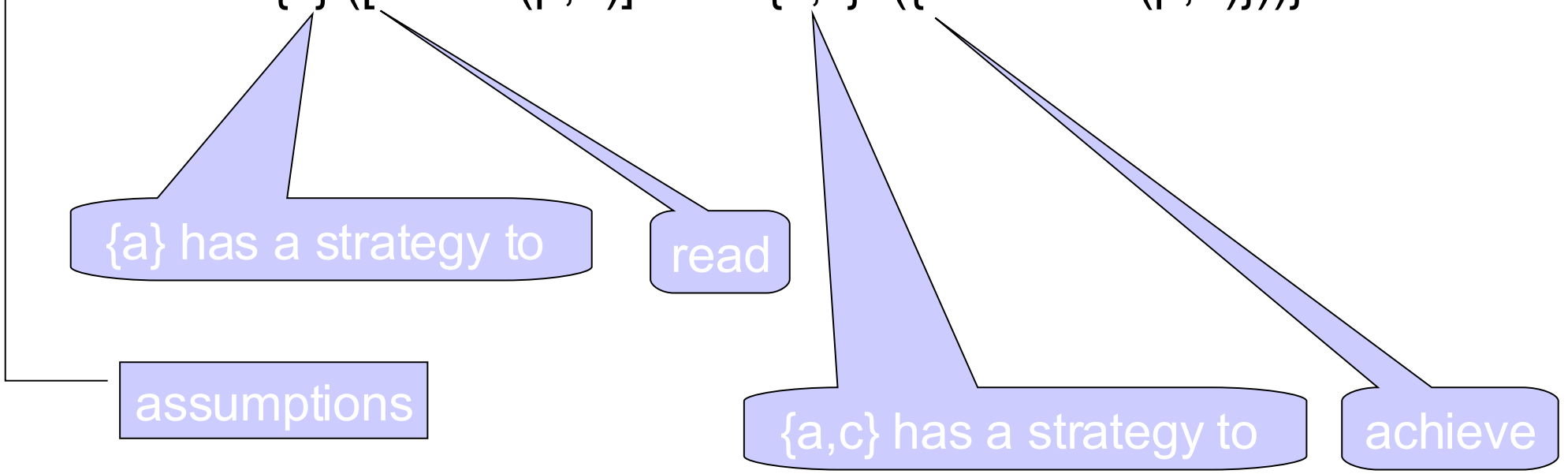
w(review(p,a),user)

# The *RW* program for the example

```
AccessControlSystem Conference

    Class  Paper;

    Predicate pcmember(agent: Agent), chair(agent: Agent)!,
       reviewer(paper: Paper, agent: Agent), submitted(paper: Paper,
       agent: Agent), review(paper: Paper, agent: Agent);

    chair(a){   read: true; }

    pcmember(a){   read : true;    write : (chair(user) | (pcmember(a) &
       a=user)); }

    reviewer(p, a){   read : pcmember(user);

      write : (chair(user) & pcmember(a)) | ((pcmember(user) & user=a &
       reviewer(p,user)));}

    submitted(p, a){   read : pcmember(user);

      write : (user=a) & reviewer(p, user) & ~submitted(p, user);}

    review(p, a){   read : pcmember(user) & submitted(p, a) & ((reviewer(p,
       user) -> submitted(p, user)) | user=a);

      write : user=a & reviewer(p,a) & ~submitted(p, user);}

End
```

# A property written in the *RW language*

```
run for 4 Paper, 8 Agent

 check {E disj a,b,c: Agent, p : Paper ||
        chair(c)*! and submitted(p,b)*! and ~submitted(p,a)! and
    pcmember(a)*! and reviewer(p,a)!
    --> {a}:([review(p,b)] AND {a,c}: ({submitted(p,a)}))}
```



# A property written in the *RW language*

```
run for 4 Paper, 8 Agent

  check {E disj a,b,c: Agent, p : Paper ||
          chair(c)*! and submitted(p,b)*! and ~submitted(p,a)! and
     pcmember(a)*! and reviewer(p,a)!
     --> {a}:([review(p,b)] AND {a,c}: ({submitted(p,a)}))}
```

Meaning: ***assume*** that

c is known to be the chair constantly;
b is known to have already submitted his review for p;

a is known to have not submitted her review for p initially; a is known to be a PC member constantly;

a is known to be a reviewer for p initially;

Then: ***are there strategies*** available for {a} and {a,c} such that such that; a can first read the result of b's review for p and then a and c can work together to enable a to submit her review for p.

# A property written in the *RW language*

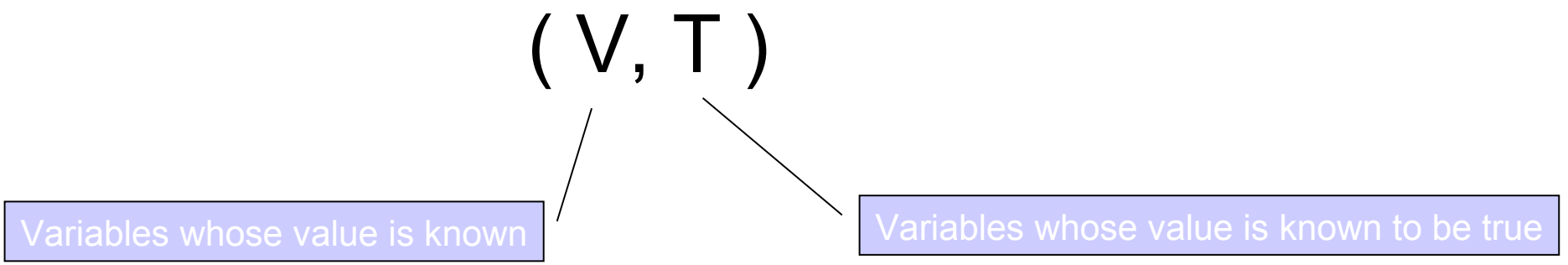
```
run for 4 Paper, 8 Agent

 check {E disj a,b,c: Agent, p : Paper ||
        chair(c)*! and submitted(p,b)*! and ~submitted(p,a)! and
    pcmember(a)*! and reviewer(p,a)!
    --> {a}:([review(p,b)] AND {a,c}: ({submitted(p,a)}))}
```

Meaning: ***a could read b's review of p, before submitting her own review of p.***

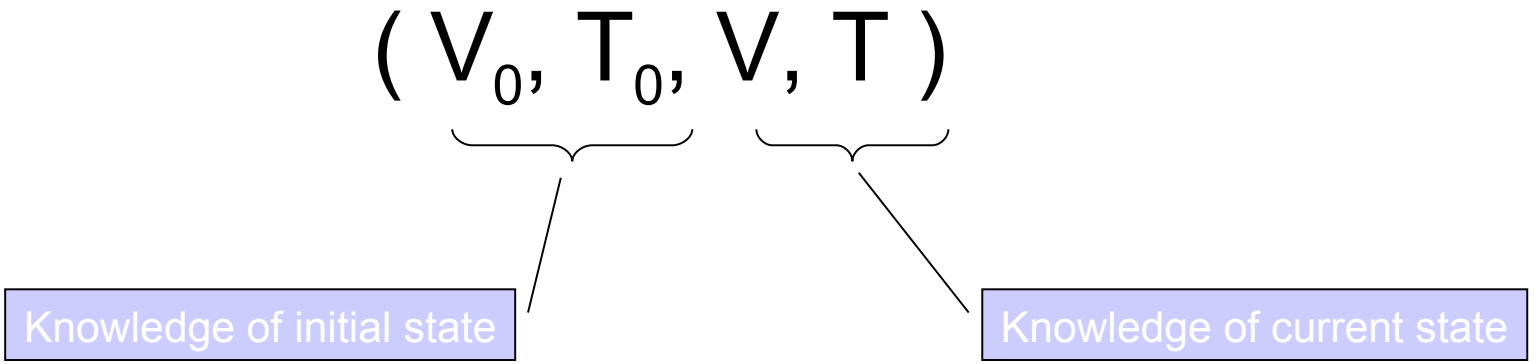
**Note: *"small model hypothesis" similar to Alloy***

# The *RW* model-checking: knowledge states

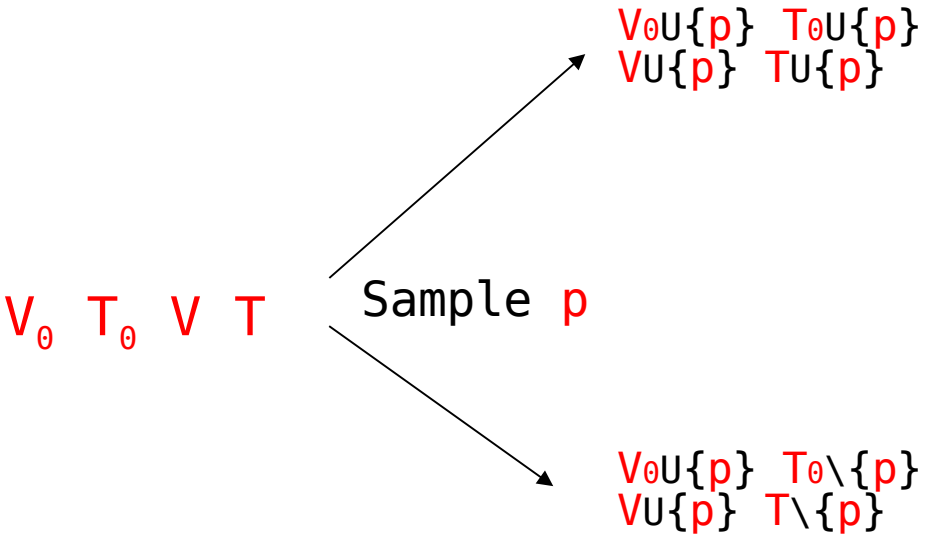
- The algorithm searches for a strategy by modelling the accumulation of the agents' knowledge about the state of the system.
  - An agent's knowledge of the system is given by




We reason with states like



# *RW* model-checking: the transition system

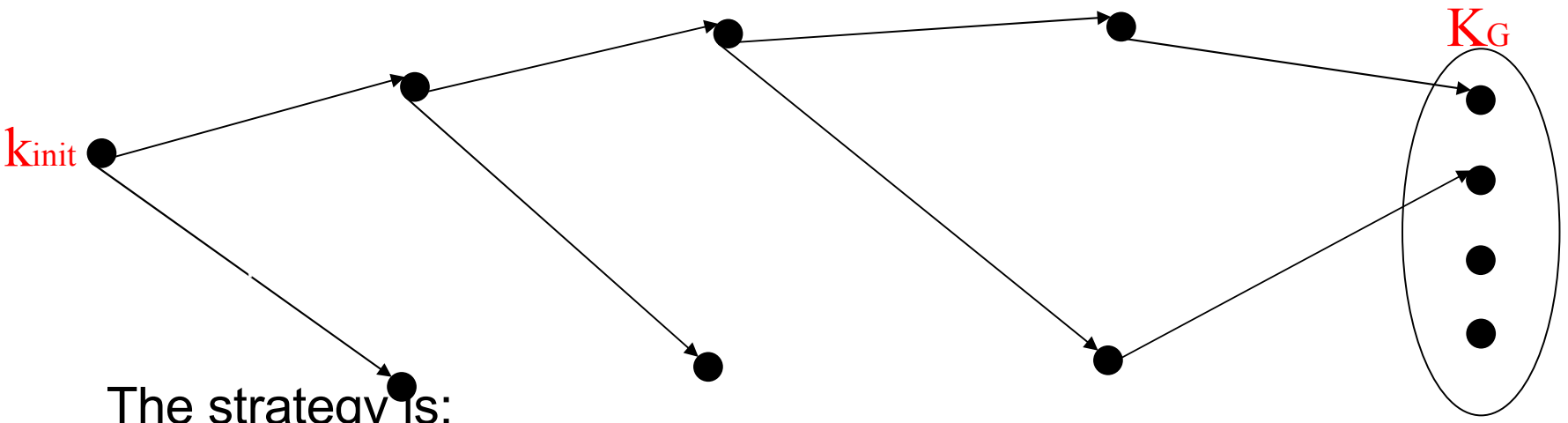



V0 T0 V T —overwrite p to true→ V0 T0 VU{p} TU{p}

V0 T0 V T —overwrite p to false→ V0 T0 VU{p} T\{p}

# The *RW* model-checking: backwards reachability

- Through sampling and overwriting the agents' knowledge state evolves from $k_{init}$ to states in $K_G$ if there is at least a strategy




The strategy is:

1, Overwrite p to true;  2, Overwrite q to true;

3, if (r is true) {

overwrite n to true;

}else {

overwrite m to false;}

- To find out such a strategy, the algorithm computes backwards reachability from $K_G$.

# The strategies found for the above example

***check {E disj a,b,c: Agent, p : Paper || chair(c)*! and submittedreview(p,b)*! and ~submittedreview(p,a)! and pcmember(a)*! and reviewer(p,a)! -> {a}:([review(p,b)] AND {a,c}: ({submittedreview(p,a)}))}***

```
[a=1 b=2 c=3 p=1]
Strategy: 1.1

Coalition: [a]
a: submitted(p,a) := true;
a: if (review(p,b)) {
   a: skip;
}else {
   a: skip;
}

Coalition: [a, c]
a:skip;
```

```
[a=1 b=2 c=3 p=1]
Strategy: 4.1

Coalition: [a]
a: set reviewer(p,a) := false;
a: if (review(p,b)) {
   a: skip;
}else {
   a: skip;
}

Coalition: [a, c]
c: reviewer(p,a) := true;
a: submitted(p,a) := true;
```

# Abstraction

- The implementation uses CEGAR: *Counter-example abstraction refinement*
- Start with an abstraction of the system
  - doesn't track all the variables that remain constant in a given transition
    - they are candidates for being considered irrelevant
- Run the model check
  - If "satisfied": great
  - If "not satisfied": check if the counterexample is valid. If it isn't valid, refine the abstraction and run the check again.

# Performance

On a laptop running Linux (kernel 2.6.10), 1.6G Pentium M, 512M RAM. Uses a BDD implementation with Java interface.

| Assignment | \|P\| | Memory usage (MB) | Time spent (ms) |
|---|---|---|---|
| Paper=2, Agent = 4 | 32 | 155 | 882 |
| Paper=3, Agent = 5 | 55 | 156 | 2776 |
| Paper=4, Agent = 6 | 84 | 159 | 6992 |
| Paper=4, Agent = 8 | 112 | 162 | 54570 |
| Paper=5, Agent = 10 | 170 | 169 | 141679 |
| Paper=6, Agent = 10 | 200 | 171 | 210172 |

# Conclusions

- The RW language – flexible and convenient
- The RW model-checking algorithm – based on transitions of knowledge states
- AcPeg – our model-checker.
- Performance results

# Current work

- Improve abstraction.

The end

Thank you

# Amending the policy

- ***The reason is: reading permissions to reviews are not restricted to reviewers***

- ***The amendment (if the reviewer has only p for reviewing):***

```
review(p, a){
    read : pcmember(user) & submittedreview(p, a) &
      ((reviewer(p, user) -> submittedreview(p, user)) |
      user=a) & (E p1: Paper [reviewer(p1,user)]);
    write : ... ;}
```

- ***The checking result: only strategy 1.1 is found***