Evaluating Access Control Policies Through Model Checking

Nan Zhang[1], Mark Ryan[1], and Dimitar P. Guelev[2]

[1] School of Computer Science, University of Birmingham, Birmingham UK, B15 2TT
{nxz,mdr}@cs.bham.ac.uk
[2] Section of Logic, Institute of Mathematics and Informatics, Acad. G. Bonchev str.,bl. 8.
1113 Sofia, BULGARIA
gelevdp@math.bas.bg

Abstract. We present a model-checking algorithm which can be used to evaluate access control policies, and a tool which implements it. The evaluation includes not only assessing whether the policies give legitimate users enough permissions to reach their goals, but also checking whether the policies prevent intruders from reaching their malicious goals. Policies of the access control system and goals of agents must be described in the access control description and specification language introduced as RW in our earlier work. The algorithm takes a policy description and a goal as input and performs two modes of checking. In the assessing mode, the algorithm searches for strategies consisting of reading and writing steps which allow the agents to achieve their goals no matter what states the system may be driven into during the execution of the strategies. In the intrusion detection mode, a weaker notion of strategy is used, reflecting the willingness of intruders to guess the value of attributes which they cannot read.

keywords: access control; access control model; model checking; verification; access control policy; access control policy language.

1 Introduction

The importance of access control is growing rapidly in a world where computers are ever-more interconnected. Access control policies are authorisation strategies upon which access control systems are built. The correctness and integrity of access control policies is crucial for an access control system to be effective. Several formalisations have been proposed in the past to understand and describe access control policies. For instance, the main principle of *role-based access control* (RBAC, [1]) is assigning access rights to agents on the grounds of their having certain roles. In another approach known as *mandatory access control* (MAC, [2]) systems enforce access control mechanisms that use clearances and sensitivity labels which can not be overridden by common users without special privileges. Programs can read information at the same or lower access levels, but can write to files at their access level and higher levels only.

The RW (where R and W stand for access by Reading and Writing, respectively) formalism [3] is another example, based on propositional logic. It allows authorisation rules to be defined based on arbitrary conditions so that it can be used for the implementation of other higher level access control mechanisms. Furthermore, it is beginning to

be language and tool supported. A machine-readable language (the RW language [4]) was created to describe policies of access control systems defined in the RW formalism and their properties. A tool was also created. It can take a RW script as input and convert the policy description into XACML [5, 4]. For the property, the tool can verify its validity by a model-checking algorithm. Due to the complexity of access control policies, in many circumstances, it is not easy to determine their correctness manually. Our tool thus makes this task easier.

In a RW script, a property is a query which asks, for a group of agents and a goal, whether the agents, acting within the permissions they have, can achieve the goal. Goals include reading and overwriting data of the system. If the goal is considered to be legitimate, we would be interested to know whether there is a strategy available for the agents so that they can always reach the goal. A positive answer to this question would mean that the access control policies grant users enough permissions for them to carry out their operations and a security hole can be regarded as the achievability of an illegitimate goal.

The question of whether a set of agents has a *strategy* to achieve its goal is an appropriate question if the agents are legitimate users and one wants to know if the system grants them the permissions they need. However, in the case that the agents are malicious intruders, a weaker question is more appropriate. A malicious user may guess the values of attributes it cannot read. Therefore, for malicious users, we ask if there is a *guessing strategy* which they can execute which will take the system from the initial state to the agent's goal state. The question is weaker because when executing a guessing strategy, the agents can guess the knowledge they need along the way. In the case of a normal strategy, they cannot guess, but must find out by sampling them.

The model-checking algorithm mentioned above is proposed to decide the achievability of a goal in a system described in RW and the tool implements the algorithm. Our algorithm and tool can be used to assess the fitness of access control systems.

Structure of the paper Section 2 is a brief formal introduction to RW . The syntax and semantics of RW scripts are briefly explained in Sect. 3. The model-checking algorithm is presented in Sect. 4. Its implementation is discussed in Sect. 5. Related work is discussed in Sect. 6, which is followed by a section of conclusions.

2 The RW Access Control Formalism

2.1 Definition

Let $\mathtt{L}(P)$ be the set of the propositional logic formulas built from the propositional variables in set P. An access control system S is a tuple $\langle A, P, \mathtt{r}, \mathtt{w}\rangle$, where A is a set of *agents*, P is a set of *propositional variables* and the mappings $\mathtt{r}, \mathtt{w} : P \times \mathcal{P}(A) \to \mathtt{L}(P)$ specify the immediate access rights of agent coalitions. States s of S are valuations of the variables from P. Agent $a \in A$ is allowed to read and overwrite variable p iff the current state s satisfies $\mathtt{r}(p, \{a\})$ and $\mathtt{w}(p, \{a\})$, respectively. We assume that rights are exercised by one agent at a time in this paper for the sake of simplicity. Thus the formulas $\mathtt{r}(p, a), \mathtt{w}(p, a) \in \mathtt{L}(P)$ define the conditions for agents to access S as functions on its state.

2.2 Example

Our running example is a simple *Employee Information System* (EIS). It is used to enforce authorisation rules on bonus allocation among the employees of a company. A bonus package with a fixed number of options, such as a-day-off, is available for employees. The director chooses options from the package to give to all employees. He/she can also read the information about the distribution of options. The director can promote an employee to be a manager. Managers can read and set ordinary employees' bonuses, but not those of other managers or the director. An employee can appoint another employee to be his advocate, and have read access to his bonus information – for example, this might be useful if he needs help from a trade union.

To put it in the RW formalism, let $\texttt{Bonus}$ be the set of bonus options, A be the set of employees and thus P include the following propositional variables, for all $b \in \texttt{Bonus}$, $a, a_1, a_2 \in A$:

$\texttt{bonus}(a, b)$	bonus option b is owned by a
$\texttt{manager}(a)$	a is a manager in the department
$\texttt{director}(a)$	a is the director of the department
$\texttt{advocate}(a_1, a_2)$	a_2 is a_1's advocate

The permission mappings $\texttt{r}$ and $\texttt{w}$ can be defined as follows: ("$\rightleftharpoons$" denotes "is defined as".)

$$\texttt{r}(\texttt{bonus}(a,b),x) \rightleftharpoons \left(\begin{array}{l}(x = a \vee \texttt{director}(x))\\ \vee \left(\texttt{manager}(x) \wedge \neg\texttt{manager}(a) \wedge \neg\texttt{director}(a)\right)\\ \vee \texttt{advocate}(a,x)\end{array}\right) \quad 1$$

$$\texttt{w}(\texttt{bonus}(a,b),x) \rightleftharpoons \left(\begin{array}{l}\left(\texttt{manager}(x) \wedge \neg\texttt{manager}(a) \wedge \neg\texttt{director}(a)\right)\\ \vee \texttt{director}(x)\end{array}\right) \quad 2$$

$$\texttt{r}(\texttt{manager}(a),x) \rightleftharpoons \texttt{true} \quad 3$$

$$\texttt{w}(\texttt{manager}(a),x) \rightleftharpoons \left(\texttt{director}(x) \vee \left(x = a \wedge \texttt{manager}(a) \wedge \neg\texttt{director}(a)\right)\right) \quad 4$$

$$\texttt{r}(\texttt{director}(a),x) \rightleftharpoons \texttt{true} \quad 5$$

$$\texttt{r}(\texttt{advocate}(a_1,a_2),x) \rightleftharpoons \texttt{true} \quad 6$$

$$\texttt{w}(\texttt{advocate}(a_1,a_2),x) \rightleftharpoons \left(x = a_1 \vee \left(\texttt{advocate}(a_1,a_2) \wedge x = a_2\right)\right) \quad 7$$

In RW everything should be defined explicitly. However, for the reason of simplicity, in this example, we assume actions which are not explicitly allowed are denied. This rule is also followed by the model checker.

We shall pick several representative rules to explain.

Rule 1 defines who can find out whether a bonus option b belongs to an employee a – the employee himself, the director, a manager, and his advocate. Rule 4 defines who can overwrite an employee a's managership – the director can both promote an employee to be a manager and demote him and an employee who has already been a manager can resign.

```
AccessControlSystem EmployeeInformationSystem
  Class Bonus;
  Predicate bonus(employee: Agent, bonus: Bonus), manager(employee: Agent), director(employee: Agent),
  advocate(appointer: Agent, appointee: Agent);

  bonus(a,b){
        read: (user=a or director(user)) or (manager(user) and ~manager(a) and ~director(a)) or (advocate(a,user));
        write: (manager(user) and ~manager(a) and ~director(a)) or director(user);
  }
  manager(a){
        read: true;
        write: (director(user) or (user=a and manager(a) and ~director(a)));
  }
  director(a){
        read: true;
  }
  advocate(a1,a2){
        read: true;
        write: user=a1 or (user=a2 and advocate(a1,a2));
  }
End
run for 4 Bonus, 8 Agent
check{E a1,a2: Agent, b: Bonus || ~director(a1) and ~director(a2) -> <~(manager(a1) and manager(a2))> or {bonus(a1,b)}}
                                                                                    where actor={a1, a2}
```

Fig. 1. The RW script for the above example.

3 The RW Access Control Description Language

3.1 Overview

Figure 1 shows the RW script for the above EIS example. The script consists of a description part which contains the policies of the system and a specification part which contains a property to be verified. The syntax and semantics of the description part is discussed in [4] using another example.

3.2 Description Part

The description part starts with class definitions. In our example, the class `Bonus` is defined. The class `Agent` is built-in, so one needs not define it explicitly. Next come the definitions of predicates. Each predicate must have at least one parameter. Parameter definitions take the form of `parameter name : parameter type`. The `parameter type` must be one of the defined classes. The following defines `r` (reading) and `w` (writing) mappings. For each parameterised predicate (a parameterised predicate corresponds to a number of variables in P), rules on reading and writing are specified by the formulas following `read :` and `write :` and are enclosed in curly brackets. These rules are defined from the perspective of the acting agent, which is denoted by `user`. Thus the rules define under what condition `user` can read and write the parameterised predicate.

3.3 Specification Part

The keyword `End` separates the description part and the specification part. The specification part starts with the *run-statement* which specifies the numbers of the elements of each class. Four elements are assigned to `Bonus` and eight elements to `Agent` in the example on Fig. 1. These elements are used to build a finite instance of the system to be model-checked. Systems of other sizes are not considered. A similar approach is taken by Alloy 3.0 [6] when the keyword `exact` is used. The *check-statement* defines a property to be verified. The *where-clause* defines the acting agents. It states that the model-checker must establish whether there is a strategy or guessing strategy (depending on the mode) available for non-director employees a_1 and a_2 such that if they can realise they are both managers then somehow they can act together to set a_1's bonus[3]. Although the policies specify a manager cannot set another manager's bonus, it doesn't prevent a_1 from resigning his/her managership and being set bonus by another manager. The result *yes* returned by the model checker shows there is indeed such a possibility. We will come back to this point in Sect. 5.2. Note that we use negation and disjunction to express implication in this case.

A *check-statement* consists of two parts, which are separated by "||". A quantifier prefix is on the left side of "||". "E" prefixes Existential variable definitions, and "A" prefixes universal variable definitions. Quantified variables defined in a same class may represent a same element during the checking. Credentials and a goal definition are on the right side of "||". Credentials and the goal are separated by "$\rightarrow$". Credentials are attributes carried by elements of the classes (usually by agents) during the process of checking. Only rigid predicates – unwritable predicates – can be used as credentials. A credential can be either positive or negative, which means the credential *is* owned by the elements or is *not* owned by the elements. Different credentials can be connected by conjunction only to form a list of credentials and used in the checking. Credentials are used as pre-conditions for the checking.

The goal expression defines the goal that the group of agents intends to achieve. We treat all the variables defined on `Agent` on the left side of "||" which also appear on the right side as the group of acting agents unless it is defined explicitly in the *where-statement* following it. If no agent-variables appear on the right side and no *where-statement* defines acting agents explicitly, we treat all agents in the `Agent` set as the group of acting agents. In other words, agents defined in a *where-statement* takes priority.

The goal is a combination consisting of conjunction and disjunction of three kinds of *atomic goals*. These are *making* goals, *realising* goals and *reading* goals, written using "{ }", "$\langle\ \rangle$" and "[ ]", respectively. For a $\varphi \in \mathtt{L}(P)$, $\{\varphi\}$ is the goal of *making* φ true; $\langle\varphi\rangle$ is the goal of *realising* that φ is true; and $[\varphi]$ is the goal of finding out the truth value of φ, whatever this value is. "Making" goals mean enforcing conditions on the system state by eventually changing it. "Reading" goals are to extract information about the system state. "Realising" goals are auxiliary and are used to allow the construction of conditionals such as $\langle\varphi\rangle$ `and` $\{\alpha\}$ `or` $\langle$`not` $\varphi\rangle$ `and` $\{\beta\}$, which means: achieve either α or β according to whether φ is true or false. A single "realising" goal $\langle\varphi\rangle$ is unlikely to be useful, because φ may simply turn out to be false. See [3] for details.

[3] We use negation and disjunction to express implication

4 The RW Model Checking Algorithm

4.1 Overview of the Algorithm

The Problem. Given an access control system and a goal, we need to determine whether a group of agents can achieve it. The goal is a combination of the atomic goals of finding out the values of some formulas about the state of the system ("reading") and driving the system into a state with a certain property ("making"). Conditions on what has to be achieved can be formulated using the auxiliary primitive goals of "realising" that something holds about the state, as mentioned in the previous section. To achieve the goal, agents can sample and overwrite variables that they are permitted to. Overwriting can be put down as simple assignment statements in the sought strategy, and sampling means that the sampled variable can be used to control conditional statements. Thus the strategy in question can be written in a simple language with assignment, sequential composition and $\mathbf{if-then-else}$. A strategy can guarantee the achievability of the goal because it contains both the outcomes of a "if" statement. A guessing strategy is like a strategy except that it allows the agents to sample a variable even if the policies do not permit them to read the variable. A guessing strategy reflects the possibilities that the agents may be able to acquire the information they need from other sources although the system prohibits them to learn. The verification problem to determine is whether such a strategy or guessing strategy exists. As we have argued in the introduction, this question is meaningful both for intrusion detection and system functionality assessment.

The Solution. Following [3], our algorithm is built around the *knowledge* of the state of the system that the considered group has at each step of implementing its strategy. Obviously there is a set of knowledge states each of which is sufficient for the group to regard its goal as achieved. This is so when the group knows that the formulas in some appropriate combination of the involved making goals are true, enough is known to work out the truth values of the formulas in the reading goals, etc. Each step takes the group from a knowledge state to a possibly richer one. A knowledge state combines knowledge of the initial state of the system and knowledge of its current state. Assignments contribute the knowledge of the current value of the assigned variable, which has been just given to it. This means that learning and changing the system are done simultaneously. To perform an assignment, a writing permission on the variable being assigned is needed. Sampling steps can be done with a reading permission and contribute both the current and the initial value of the sampled variable, unless it has already been overwritten. In the latter case sampling is redundant, because the current value must have become known upon writing it. Overwriting without sampling in advance destroys the prospect to learn the initial value of the variable. Strategies are supposed to take the group from the empty knowledge state[4] to one in which it can deem its goal achieved.

[4] Normally we assume the agents have no knowledge about the system initially, however when credentials are used we assume the agents hold the knowledge about the credentials and the knowledge is used as pre-conditions for the checking.

To describe the group of agents' knowledge on p, we use four knowledge variables. For each $p \in P$, we have

v_{0p} is true if the agents know the initial value of p
t_{0p} is true if the agents know initially p is true
v_p is true if the agents know the current value of p
t_p is true if the agents know currently p is true

When overwriting p to true, v_p and t_p both become true, but v_{0p} and t_{0p} do not change, because it does not increase the agents' knowledge on p's initial value. When overwriting p to false, v_p becomes true; t_p becomes false; both v_{0p} and t_{0p} do not change. When sampling p, where p has not been overwritten, v_{0p} and v_p both become true and t_{0p} and t_p both become false if p turns out to be false, or t_{0p} and t_p both become true if p turns out to be true. Since the contents of t_{0p} and t_p are irrelevant when p is unknown, and the initial value of a variable is known only if the current value is known too, there are indeed only 7, and not 2^4 knowledge states about each variable p. However it is easier to explain our algorithm in terms of v_{0p}, t_{0p}, v_p and t_p as independent variables.

A knowledge state is given by the quadruple (V_0, T_0, V, T), where $V_0 = \{p \in P \mid v_{0p} \texttt{ is true}\}$, $T_0 = \{p \in P \mid t_{0p} \texttt{ is true}\}$, $V = \{p \in P \mid v_p \texttt{ is true}\}$, $T = \{p \in P \mid t_p \texttt{ is true}\}$. we show the effects that the above three kinds of transitions have on knowledge states in Fig. 2.

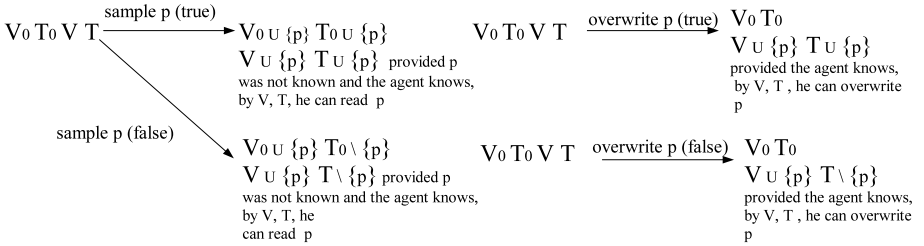

Fig. 2. The transitions.

Therefore, by modelling the accumulation of agents' knowledge, we build a transition system over the access control system in question. Three kinds of transitional relations can be identified – *overwriting-to-true*, *overwriting-to-false* and *sampling*, each of which will carry the knowledge states of agents from one to another until the agents have confidence to deduce the goal is reached from their knowledge states. Once the agents reach the knowledge states from which they can deduce their goal is reached, we regard their goal has been reached. This procedure is illustrated in Fig. 3.

Note the transition relations for overwriting are deterministic; the relation for sampling is not. A strategy should lead the agents to the goal through both possible outcomes of a sampling.

To find out if there is such a strategy our solution is to invert the whole process described above and work backwards. We start from the set of knowledge states where

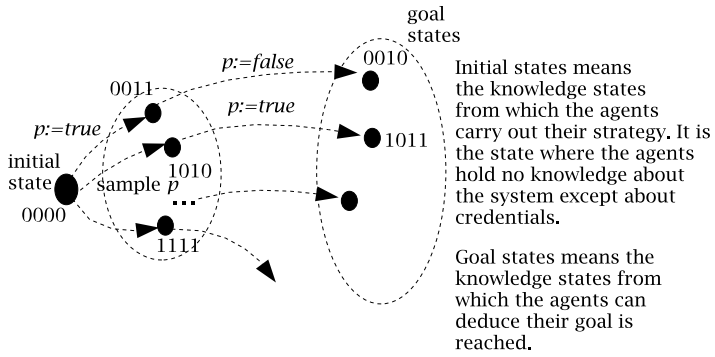

Fig. 3. The process of learning.

the goal can be deemed as achieved. Let $K_{\mathtt{G}}$ denote this set as represented in terms of the variables v_{0p}, t_{0p}, v_p and t_p, $a \in$ the agents, $p \in P$. Given a set of knowledge states Y, we denote

$\mathtt{Pre}^{\exists,a}_{p:=\top}(Y)$ means the set of knowledge states in which a knows it is permitted to overwrite p and which transition into Y by overwriting p to true ($\top$). Its formal definition is: $\{(V_0, T_0, V, T) \mid \exists\ (V_0^{'}, T_0^{'}, V', T') \in Y,\ V_0^{'} = V_0, T_0^{'} = T_0, V' = V \cup \{p\}, T' = T \cup \{p\}, \mathtt{w}(p,a)[\bot/p:\ p \in V \setminus T][\top/p:\ p \in T] = \top\}$.

$\mathtt{Pre}^{\exists,a}_{p:=\bot}(Y)$ means the set of knowledge states in which a knows it is permitted to overwrite p and which transition into Y by overwriting p to false ($\bot$). Its formal definition is: $\{(V_0, T_0, V, T) \mid \exists\ (V_0^{'}, T_0^{'}, V', T') \in Y,\ V_0^{'} = V_0, T_0^{'} = T_0, V' = V \cup \{p\}, T' = T \setminus \{p\}, \mathtt{w}(p,a)[\bot/p:\ p \in V \setminus T][\top/p:\ p \in T] = \top\}$.

$\mathtt{Pre}^{\exists,a}_{p=\top}(Y)$ means the set of knowledge states in which a knows it is permitted to sample p and which transition into Y by sampling p and find out it is true ($\top$). Its formal definition is: $\{(V_0, T_0, V, T) \mid \exists\ (V_0^{'}, T_0^{'}, V', T') \in Y,\ p \notin V_0, p \notin T_0, p \notin V, p \notin T, V_0^{'} = V_0 \cup \{p\}, T_0^{'} = T_0 \cup \{p\}, V' = V \cup \{p\}, T' = T \cup \{p\}, \mathtt{r}(p,a)[\bot/p: p \in V \setminus T][\top/p:\ p \in T] = \top\}$.

$\mathtt{Pre}^{\exists,a}_{p=\bot}(Y)$ means the set of knowledge states in which a knows it is permitted to sample p and which transition into Y by sampling p and find out it is false ($\bot$). Its formal definition is: $\{(V_0, T_0, V, T) \mid \exists\ (V_0^{'}, T_0^{'}, V', T') \in Y,\ p \notin V_0, p \notin T_0, p \notin V, p \notin T, V_0^{'} = V_0 \cup \{p\}, T_0^{'} = T_0 \setminus \{p\}, V' = V \cup \{p\}, T' = T \setminus \{p\}, \mathtt{r}(p,a)[\bot/p: p \in V \setminus T][\top/p:\ p \in T] = \top\}$.

During the course of the algorithm, we maintain pairs (Y, s) consisting of a set Y of knowledge states and a strategy s. The pair (Y, s) denotes the fact that s is a strategy that enables the agents to reach $K_{\mathtt{G}}$ from states in Y. For $K_{\mathtt{G}}$, the s is simply "`skip;`", which means "do nothing".

We start with the pair $(K_{\mathtt{G}}, \mathtt{skip;})$. The core of the algorithm works as follows: given the pair (Y, s), we add the pairs $(\mathtt{Pre}^{\exists,a}_{p:=\top}(Y), (p := \top;\ s))$ and $(\mathtt{Pre}^{\exists,a}_{p:=\bot}(Y), (p$

$:= \bot;\ s)$). For any two pairs (Y_1, s_1) and (Y_2, s_2), we add the pair $(\mathtt{Pre}^{\exists,a}_{p=\top}(Y_1) \cap \mathtt{Pre}^{\exists,a}_{p=\bot}(Y_2), \mathbf{if}\ (p)\ \mathtt{by}\ a\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2)$.

we continue until no new pairs are generated. Now, all the pairs whose set of knowledge states contains the initial knowledge state contain the strategies we are looking for.

To find out guessing strategies instead of strategies, the only thing needs to be changed is to omit the condition $\mathtt{r}(p,a)[\bot/p:\ p \in V \setminus T][\top/p:\ p \in T] = \top$ when computing $\mathtt{Pre}^{\exists,a}_{p=\top}(Y)$ and $\mathtt{Pre}^{\exists,a}_{p=\bot}(Y)$.

4.2 The Algorithm.

The algorithm for extracting strategies is described below in the form of pseudo-code. It assumes as input the initial state $k_{\mathtt{init}}$ and the set of goal knowledge states $K_{\mathtt{G}}$. It outputs at least a strategy for going from $k_{\mathtt{init}}$ to some element of $K_{\mathtt{G}}$. The algorithm works by backwards reachability from $K_{\mathtt{G}}$ to $k_{\mathtt{init}}$. It maintains a set of states it has seen, called `states_seen`, and a data structure associating subsets of `states_seen` with strategies for reaching $K_{\mathtt{G}}$ from them, called `strategies`.

We use A to denote the group of acting agents. The algorithm is:

Input: $K_{\mathtt{G}}$ - set of goal knowledge states $k_{\mathtt{init}}$ - the initial knowledge state
P - set of propositional variables A - set of acting agents (not the set of all agents)
`r`, `w` - reading and writing privilege definitions (will be used when computing the pre-sets, though not explicitly shown in the algorithm)
Output: at least a strategy for going from $k_{\mathtt{init}}$ to some element of $K_{\mathtt{G}}$ if such strategies exist

`strategies` := $\emptyset$;
`states_seen` := $\emptyset$;
put $(K_{\mathtt{G}}, \mathtt{skip};)$ in `strategies`;
repeat until `strategies` does not change{
  choose $(Y_1, s_1) \in$ `strategies`; // for all pairs in `strategies`
  for each $p \in P${
    for each $a \in A${
      $PTY_1 := \mathtt{Pre}^{\exists,a}_{p:=\top}(Y_1)$;
      if $((PTY_1 \neq \emptyset) \wedge (PTY_1 \not\subseteq$ `states_seen`$))${
        `states_seen` := `states_seen` $\cup\ PTY_1$;
        pts_1 := "set p to $\top$ by a;" + s_1;
        `strategies` := `strategies` $\cup\ \{(PTY_1, pts_1)\}$;
        if ($k_{\mathtt{init}} \in PTY_1$)
          output pts_1;
      }
      $PFY_1 := \mathtt{Pre}^{\exists,a}_{p:=\bot}(Y_1)$;
      if $((PFY_1 \neq \emptyset) \wedge (PFY_1 \not\subseteq$ `states_seen`$))${
        `states_seen` := `states_seen` $\cup\ PFY_1$;
        pfs_1 := "set p to $\bot$ by a;" + s_1;

```
            strategies := strategies ∪ {(PFY_1, pfs_1)};
            if (k_init ∈ PFY_1)
                output pfs_1;
        }
    }
}
choose (Y_2, s_2) ∈ strategies;   // for all pairs in strategies
for each p ∈ P{
    for each a ∈ A{
        PSY := Pre^{∃,a}_{p=⊤}(Y_1) ∩ Pre^{∃,a}_{p=⊥}(Y_2);
        if ((PSY ≠ ∅) ∧ (PSY ⊈ states_seen)){
            states_seen := states_seen ∪ PSY;
            strategies := strategies ∪ {(PSY, pss)};
            pss := "if (p) by a then s1 else s2";
            if (k_init ∈ PSY)
                output pss;
        }
    }
}
}
```

4.3 Proof of Correctness

Theorem 1. *The algorithm will eventually terminate.*

Proof. To prove the algorithm will terminate is equivalent to proving that the size of `strategies` will not infinitely grow. The set `strategies` only increases if we encounter states not yet in `states_seen`. As there are only finitely many states, we cannot go on encountering new states for ever.

Lemma 1. *If there exists a strategy s, then there exists a way of resolving the choice in the algorithm such that s is outputted.*

Proof. Suppose s is such a strategy. Assume without loss of generality that s never samples variables it has previously assigned. We recursively annotate the strategy with the knowledge states which arise from executing the strategy at $k_{\texttt{init}}$, according to these rules:

(i) The strategy s is annotated with $(\emptyset, \emptyset, \emptyset, \emptyset)$.
(ii) If "$p := \top; s_1$" is annotated with the state (V_0, T_0, V, T) then s_1 gets annotated with $(V_0, T_0, V \cup \{p\}, T \cup \{p\})$.
(iii) If "$p := \bot; s_1$" is annotated with the state (V_0, T_0, V, T) then s_1 gets annotated with $(V_0, T_0, V \cup \{p\}, T \setminus \{p\})$.
(iiii) If "$\textbf{if } (p) \textbf{ then } s_1 \textbf{ else } s_2$" is annotated with (V_0, T_0, V, T), we annotate s_1 with $(V_0 \cup \{p\}, T_0 \cup \{p\}, V \cup \{p\}, T \cup \{p\})$ and s_2 with $(V_0 \cup \{p\}, T_0 \setminus \{p\}, V \cup \{p\}, T \setminus \{p\})$.

Let Y be the set of states which annotate the leaves of s. Then $Y \subseteq K_{\mathtt{G}}$, by hypothesis. Judicious resolution of the choice operator in the algorithm, corresponding to the strategy s, will result in states which include each annotation being considered by the algorithm, until finally a state including $k_{\mathtt{init}}$ is considered.

Theorem 2. *If there are strategies from $k_{\mathtt{init}}$ to $K_{\mathtt{G}}$ the algorithm finds at least one of them.*

Proof. Following Lemma 1, however the choice operator is resolved, $k_{\mathtt{init}}$ will eventually be included in `states_seen`, and therefore some strategy will be generated.

Lemma 2. *For all $(Y, s) \in$ `strategies`, and for all $y \in Y$, s succeeds on y and the result is in $K_{\mathtt{G}}$.*

Proof. We look at all the ways that (Y, s) can be added to `strategies`. At the beginning, $(K_{\mathtt{G}}, \mathtt{skip};)$ is added in. the correctness of the lemma is self-evident for this case. During the course of the algorithm, pairs are added in one of these three circumstances:

(i) (PTY_1, pts_1) is added, where, $\exists\, a \in A$ and $p \in P$, such that $PTY_1 = \mathtt{Pre}^{\exists, a}_{p:=\top}(Y_1)$, pts_1 = "set p to $\top$ by a;" + s_1, and (Y_1, s_1) is in `strategies`.
We know by the inductive hypothesis for all $y_1 \in Y_1$, s_1 succeeds on y_1 and result is in $K_{\mathtt{G}}$. We also know for all $y \in PTY_1$ that a can do $p := \top$ and that the result of that is in Y_1, because that is the way we get PTY_1 from Y_1. Therefore pts_1 succeeds on all the states in PTY_1 and the result is in $K_{\mathtt{G}}$.

(ii) (PFY_1, pfs_1) is added, where, $\exists\, a \in A$ and $p \in P$, such that $PFY_1 = \mathtt{Pre}^{\exists, a}_{p:=\bot}(Y_1)$, pfs_1 = "set p to $\bot$ by a;" + s_1, and (Y_1, s_1) is in `strategies`.
The argument for the above case applies also to this one.

(iii) (PSY, pss) is added, where, $\exists\, a \in A$ and $p \in P$, such that $PSY = \mathtt{Pre}^{\exists, a}_{p=\top}(Y_1) \cap \mathtt{Pre}^{\exists, a}_{p=\bot}(Y_2)$, pss = "if (p) by a then $s1$ else $s2$" and (Y_1, s_1), and (Y_2, s_2) are both in `strategies`.
We know by the inductive hypothesis for all $y_1 \in Y_1$, s_1 succeeds on y_1 and result is in $K_{\mathtt{G}}$, and $y_2 \in Y_2$, s_2 succeeds on y_2 and result is in $K_{\mathtt{G}}$. We also know for all $y \in PSY$ that a can read p and if it is $\top$, the result of that is in Y_1. However, if it is $\bot$, the result of that is in Y_2. Therefore pss succeeds on all the states in PSY and the result is in $K_{\mathtt{G}}$.

Theorem 3. *If the algorithm outputs the strategy s then s succeeds on $k_{\mathtt{init}}$ and the result is in $K_{\mathtt{G}}$.*

Proof. From Lemma 2 we know that for all $(Y, s) \in$ `strategies` and $y \in Y$, s succeeds on y and the result is in $K_{\mathtt{G}}$. Because if s gets outputted, there must exist a Y, such that $k_{\mathtt{init}} \in Y$ and $(Y, s) \in$ `strategies`. Therefore, it follows that s succeeds on $k_{\mathtt{init}}$ and the result is in $K_{\mathtt{G}}$.

From the implication of theorem 3, we know if there is no strategy s which succeeds on $k_{\mathtt{init}}$ and results in $K_{\mathtt{G}}$, the algorithm will output none.

4.4 Computational Complexity

We use K for the set of all the knowledge states, $|K|$ for the total number of knowledge states, $|P|$ for the number of variables in P, $|A|$ for the number of acting agents. The computation time of the algorithm depends on the number of subsets of K it finds. In the worst case the number of the subsets of K is $|K|$ because we prevent any subset whose elements are already found from being added to `strategies`. Thus the worst case is that subsets of K are just singletons. Because the time spent on computing pre-sets does not depend on $|K|$, the worst-case complexity is $|K| \times (|P| \times |A| + |K| \times |P| \times A) = |K|^2$.

5 Implementation

5.1 Performance

We have implemented the above algorithm in Java. Computations are done in BDDs[5]. The tool can be downloaded from [8]. Its performance is good, despite the state explosion problem. In the EIS example, we assign 4 elements to the `Bonus` set and 8 to `Agent`. The total number of variables in P is 112. For each variable in P we have four knowledge variables to describe the agents' knowledge about it. Thus the total number of variables in BDDs for knowledge states is $112 \times 4 = 448$. During the computation we also need the primed version of variables, for all the variables in P and all knowledge variables. Therefore, the total number of variables we need in BDDs for knowledge states and transition relations together in the EIS example is $112 \times 10 = 1120$. On a computer (Pentium M 1.6G, 512M memory, running Linux, kernel version 2.6.10), it finishes one round of computation, finding one strategy, in about 18 seconds and consumes less than 160MB memory. Whereas the processing power of today's PCs grows very fast, we think our tool is highly usable. For a strategy found by the tool, see Fig. 4

5.2 Abstraction

We have used abstraction to enable the handling of large cases by our tool. One of the bottlenecks in our approach is the computations like $V' = V \cup \{p_i\}$. That computations represent the fact that reading or overwriting p_i only change the agents' knowledge on p_i – it does not change the agents' knowledge on other variables in P. In other words, we keep on tracking the agents' knowledge on all the variables in P, when an action is only performed on p_i. For reasons of efficiency, it would be better not to maintain the agents' knowledge on all variables when actions are performed on p_i.

Therefore we have introduced three abstraction levels in the tool for users to specify when running it. The minimum level, which is level 0, is the level that no abstraction is used, that is, the tool maintains the agents' knowledge on all variables in all computations. It is the most precise level. The maximum level, level 2, is the level when an action is performed on p_i, the tool not only maintains the agents' knowledge on p_i, but also on all the other variables that occur in the goal. In the middle, level 1 is built

[5] The Java BDD package we use can be obtained from [7]

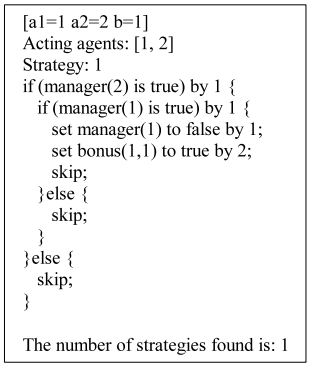

Fig. 4. A strategy found by the model checker. (Note: [a_1=1 a_2=2 b=1] is the assignment, meaning a_1 is assigned the first element in Agent, a_2 is assigned the second element in Agent, and b is assigned the first element in Bonus.)

on level 2. In this level, the tool not only maintains the agents' knowledge on p_i and all the variables in the goal, as level 2 does, but also maintains the agents' knowledge on any other variables in P specified by the user in a configuration file named *abstraction.config*. When working on large systems, this level can be used as counter-example driven refinement abstraction. In this level, when a false strategy is found, one can analyse that which variable has caused this strategy to be found. Thus one can put that variable in *abstraction.config* and run the model checker again. Having kept tracking on this variable, a number of false strategies will be ruled out. The result will be more and more precise.

With these abstraction levels, the tool performs much better. However, the more abstraction we use, from level 0 to level 2, the more precision we lose. If in level 1 or 2, the checking result is $\bot$, then it really means there is no strategy for the agents to reach their goal. But if it is $\top$, it does not guarantee there is a strategy. In fact, the answer is uncertain. By not maintaining the agents' knowledge on all variables, some transitions which actually can not happen may not be ruled out.

6 Related Work

Access control policies analysis has attracted much attention in recent years. Fisler and her colleagues [9] focus on verification and change-impact analysis of role-based access control policies written in XACML. They have a tool called Margrave, which reads XACML, translating them into multi-terminal decision diagrams (MTBDDs) [10] to answer queries. MTBDDs are a more general form of BDDs. Unlike a BDD which only has two terminals, 0 and 1, a MTBDD can have a set of terminals. Because XACML policy evaluation may lead to the result of *permit*, *deny* and *not-applicable*, MTBDDs are more suitable for translating XACML policies than BDDs. Margrave verifies whether a policy preserves a property by taking a query which expresses the property as input

and outputs the answer to the query. It does do by traversing the MTBDD for the policy, using the information provided in the query and seeing which terminal it gets to. Change-impact analysis is also an important aspect of their work. Margrave can take two policies that span a set of changes as input and output a summary of the differences. Two big advantages of the approach from [9] are performance and scalability. According to their experimental data, most verification tasks take no longer than 10 milliseconds (ms), however representing policies take from 70ms to 335ms. Memory consumption is about 4.7Mbytes. Because MTBDDs scale up quite well, the tool might be capable to handle large cases.

However their approach can not detect hidden channels caused by multi-step actions and co-operations.

Consider the policies in Fig. 1 and the strategy found by the tool in Fig. 4. The policy specifies that *no manager can set another manager's bonus.* However, being two managers, a_1 and a_2, they can work together to breach the spirit of this policy, as Fig. 4 shows. First, a_1 resigns its managership. Secondly, a_2 sets a_1's bonus. Although each of the two steps are permitted by the policies, the combining result renders the policies powerless. This kind of hidden channels can not be detected by static analysis, such as [11] and [12], or simply querying a policy. Our approach can reveal such kind of weaknesses in policies because in finding the strategies we consider what coalition of agents can achieve. Model-checking's power of temporal reasoning also helps to reveal possible attacks achieved by multi-step actions.

Schaad and Moffett [13] demonstrate how to use Alloy [6] to check that separation-of-duty constraints may be breached when policies are changed by administrative policies defined in the ARBAC97 model. We have considered the possibility to use Alloy as our modelling formalism and the Alloy analyser [14] as our checking tool too. However, since Alloy has no built-in temporal reasoning, if we use Alloy, we have to hard-code system states and the transition relations explicitly by ourselves. From our experience, we found that this makes models in Alloy too complex and the checking too inefficient. Alloy's lack of temporal reasoning makes it unsuitable for our work.

7 Conclusion

We have discussed the RW access control system description and verification framework. It includes the RW formalism, the RW language and a tool which can both convert a description of access control policies in the RW language into a XACML policy file for implementation and perform verification on the specification in the script. This paper focuses on the verification part.

The model-checking algorithm discussed answers whether a goal can be achieved and figures out how it can be achieved. we have added three abstraction levels to the tool to enable trade-offs between precision and performance. However, even without abstraction, the performance of the tool is good enough to do some reasonably sized cases. With abstraction, the performance is even better. The tool can only check cases of fixed sizes. Nevertheless this is often sufficient. As Daniel Jackson has argued in the case of Alloy; small size checks are still extremely valuable for finding errors [6].

The practical applicability of our framework first depends on the modelling power of the RW formalism. The RW formalism can be used to model various access control systems. For an access control system, what the RW formalism models are attributes of the system and the permission relations which are based on the attributes. The RW formalism captures the essential aspects of a system in a highly abstract way so that unimportant issues may be ignored. That is why RW formalism can be adapted to model a wide range of access control systems.

Our framework can be used to detect errors in policies of existing access control systems. When errors are found, one may figure out how to amend the policies by reading the strategies output by the tool. However, our framework also helps to the design and implementation of an access control system. One may use the tool to verify the proposed policies and then translate them into XACML so that a real access control system can be built on them.

References

1. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-based access control models. IEEE Computer **29** (1996) 38–47
2. Anderson, R.: Security Engineering: A Guide to Building Dependable Distributed Systems. John Wiley & Sons, Inc., U.S.A. (2001)
3. Guelev, D.P., Ryan, M.D., Schobbens, P.Y.: Model-checking access control policies. In: the Seventh Information Security Conference (ISC'04). Lecture Notes in Computer Science, Springer-Verlag (2004)
4. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems in XACML. In: the 2004 ACM Workshop on Formal Methods in Security Engineering, Washington DC, USA, ACM Press (2004) 56–65
5. Godik, S., Moses, T.: eXtensible Access Control Markup Language. OASIS committee. 1.1 edn. (2003) Committee specification.
6. Jackson, D.: Micromodels of Software: Lightweight Modelling and Analysis with Alloy. Software Design Group, MIT Lab for Computer Science. (2002) This document and the tool can be obtained from `http://alloy.mit.edu/`.
7. Whaley, J.: JavaBDD: Java BDD implementation (2004) Information about this implementation can be found at `http://javabdd.sourceforge.net/`.
8. Zhang, N.: Web site for the access control policy evaluator and generator (2005) The tool can be obtained from `http://www.cs.bham.ac.uk/~nxz`.
9. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE'05, St. Louis, Missouri, USA (2005)
10. Clarke, E., Fujita, M., McGeer, P., Yang, J., Zhao, X.: Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In: International Workshop on Logic Synthesis, Tahoe City (1993)
11. Ahmed, T., Tripathi, A.R.: Static verficiation of security requirements in role based CSCW systems. In: SACMAT'03, Como, Italy (2003)
12. Chess, B.: Improving computer security using extended static checking. In: 2002 IEEE Symposium on Security and Privacy, Washington, DC, USA, IEEE Computer Society (2002)
13. Schaad, A., Moffett, J.: A lightweight approach to specification and analysis of role-based access control extensions. In: SACMAT'02, Monterey, California, USA (2002)
14. Jackson, D., Schechter, I., Shlyahter, H.: Alcoa: the Alloy constraint analyzer. In: the 22nd international conference on Software engineering, ACM Press (2000) 730–733