A knowledge-based verification method for dynamic access control policies

Masoud Koleini and Mark Ryan

University of Birmingham,
Birmingham, B15 2TT, UK
{m.koleini,m.d.ryan}@cs.bham.ac.uk

Abstract. We present a new approach for automated knowledge-based verification of access control policies. The verification method not only discovers if a vulnerability exists, but also produces the strategies that can be used by the attacker to exploit the vulnerability. It investigates the information needed by the attacker to achieve the goal and whether he acquires that information when he proceeds through the strategy or not. We provide a policy language for specifying access control rules and the corresponding query language that is suited for expressing the properties we aim to verify. The policy language is expressive enough to handle integrity constraints and policy invariants. Finally, we compare the results and enhancements of the current method - implemented as a policy verification tool called *PoliVer* - over similar works in the context of dynamic access control policy verification.

1 Introduction

Social networks like Facebook and LinkedIn, cloud computing networks like Salesforce and Google docs, conference paper review systems like Easychair and HotCRP are examples of the applications that huge numbers of users deal with every day. In such systems, a group of agents interact with each other to access resources and services. Access control policies in such multi-agent systems are dynamic (state-based) [1–4], meaning that the permissions for an agent depend on the state of the system. As a consequence, permissions for an agent can be changed by the actions of other agents.

For complex systems, reasoning by hand about access control policies is not feasible. Automated verification is a solution and enables policy designers to verify their policies against properties needed. For instance, in Google docs, we need to verify "if Alice shares a document with Bob, it is not possible for Bob to share it with Charlie unless Alice agrees", or in HotCRP, "if Bob is not chair, it is not possible for him to promote himself to be a reviewer of a paper submitted to the conference". If such properties do not hold, it can imply a security hole in the system and needs to be investigated and fixed by policy designers.

Knowledge - the information that an agent or group of agents has gained about the system - plays an important role in exploiting vulnerabilities by the attacker. For instance in Facebook, consider a situation in which Alice is a friend

of Bob, and she has excluded non-friends from seeing her photos and her list of friends. Bob has tagged Alice on some photos of him, which are publicly available. Eve is interested in finding some photos of Alice. If Eve knows that Alice and Bob are friends, then the pseudocode below demonstrates how she can proceed:

```
foreach (photo ∈ Bob.photos)
  if (photo.isAccessibleBy(Eve) and Alice ∈ photo.tags)
    Output photo;
```

Although this vulnerability exists, Eve still needs to find some of Alice's friends to exploit it. The required information may be a prior knowledge, or gained by exploring the system. In both cases, a verification method that investigates how the agents can gain information about the system, share it with other agents and use the information to achieve the goal is valuable in debugging access control policies.

This paper proposes a dynamic access control model supporting knowledge-based verification through reasoning about readability. In this context, an agent knows the value of a proposition[1] (for instance, areFriends(Alice, Bob)) if he has previously read the proposition, or performed an action that has altered its value. This abstraction of knowledge results in a simpler model, which makes the verification efficient, and is powerful enough to model knowledge in access control systems. Using this definition of knowledge, we are able to efficiently verify a property - as a vulnerability - over access control systems, and if the property is satisfied, produce an output which demonstrates how an agent can execute a sequence of actions to achieve the goal, what information he requires to safely proceed through the strategy and what are the risky situations where he needs to guess what action to perform.

As an important feature in this paper, we are interested in finding the system propositions in which the strategy for the attacker to achieve the goal is different according to whether the proposition is true or false. We call those propositions *effective*. The values of effective propositions are needed by the attacker to determine the appropriate strategy. If the attacker does not know the value of the effective proposition, he could still guess the value. In the case of wrong guess, he may be able to backtrack to the guessing state and select the right strategy. However, backtracking has two main disadvantages: firstly, it may not be possible to backtrack or undo the actions already performed and secondly, unauthorized actions may be logged by the system. So, the attacker needs to minimize guessing to get the goal.

The proposed algorithm is able to:

- Verify a property (or equivalently, goal) over an access control system which is characterised by a dynamic policy.
- Provide the strategy together with the information required for an attacker to achieve the goal, if the goal is found to be achievable.

[1] In the context of this paper, a proposition refers to a boolean variable in the system, and a state is a valuation of all system propositions.

– Find out if the attacker can gain the required knowledge while he traverses through the strategy.

Our contribution: We propose a policy language with corresponding verification algorithm that handles integrity constraints - rules that must remain true to preserve integrity of data. The policy language enables users to define action rules and also read permission rules to represent agent knowledge in the system. We provide a verification algorithm (with respect to effective propositions) which is able to find the strategy in a more efficient way than the guessing approach in a similar knowledge based verification framework [1]. The algorithm verifies knowledge by reasoning about readability. This approach approximates knowledge, finds errors efficiently and is easier to automate. Finally, we present case studies for strategy finding and knowledge verification algorithm and compare the performance with similar methods.

The rest of this paper is organized as follows. Related work is discussed in Section 2. Formal definitions of access control policy, access control system and query language are introduced in Section 3. Model-checking strategy is explained in Section 4. Knowledge-based verification of the strategies is presented in Section 5. Experimental results are provided in section 6 and conclusions and future work are explained in Section 7.

Notation 1 *To enhance readability, for the rest of this paper,* letters with no index *such as u and a used as the arguments will represent* variables. indexed letters *such as* u_1 *and* a_1 *will be used for* objects *(instantiated variables).*

2 Related work

Although there is lots of research in the area of stateless access control system verification [5–7], we only mention several important related papers in the context of dynamic policy verification.

One of the first works is the security model of Bell and LaPadula designed in 1976 [8]. This model is a state transition framework for access control policies in a multi-level security structure and is based on security classification of objects and subjects. In general, the model is not fine-grained, not all access control policies can be modelled and also contains several weaknesses [9].

Dougherty et al. [3] define a datalog-based verification of access control policies. They have separated static access control policy from the dynamic behaviour and defined a framework to represent the dynamic behaviour of access control policies. They consider an *environment* consisting of the facts in the system. Performing each action adds some facts or removes some other from the environment. They perform formal analysis for *safety* and *availability* properties on their access control model.

In terms of verifying knowledge, RW [1] is the most similar framework to ours[2]. "Read" and "write" rules in RW define the permissions for read/write

[2] RW is implemented as a tool named AcPeg (Access Control Policy Evaluator and Generator).

access to the system propositions. RW considers agent knowledge propositions in state space. So, an agent can perform an action if he knows he is able to perform it. RW suffers from the restriction that only one proposition can be updated at a time in every write action. Our policy language allows defining actions with bulk variable update. As a practical limitation, the state space in RW grows in a greater context than conventional model-checkers, which makes the verification of complex policies difficult. Our method is more efficient as it abstracts knowledge states and and uses a fast post-processing algorithm for knowledge verification.

SMP [10] is a logic for state-modifying policies based on transaction logic. Although SMP provides an algorithm that finds the optimal sequence of transitions to the goal, it suffers from restricted use of negation in preconditions, which is not the case in our proposed algorithm.

Becker [2] has designed a policy language (DynPAL) that is able to verify safety properties over dynamic access control policies with an unbounded number of objects. The paper proposes two methods for reasoning about *reachability* and *policy invariants*. For reachability, the policy can be translated into PDDL syntax [11] and verified using a planner. Safety properties can be verified using a first order logic theorem prover and by translating the policy and invariance hypothesis into the first order logic validity problem. According to the experimental results [2], the planner may not be successful in finding if a property is an invariant in a reasonable time. Also initial states are not considered in safety property verification.

3 Definitions

3.1 Access control policy

In a multi-agent system, the agents authenticate themselves by using the provided authentication mechanisms, such as login by username and password, and it is assumed that the mechanism is secure and reliable. Each agent is authorized to perform actions, which can change the system state by changing the values of several system variables (in our case, atomic propositions). Performing actions in the system encapsulates three aspects: the agent request for the action, allowance by the system and system transition to another state. In this research, we consider agents performing different actions asynchronously; a realistic approach in computer systems. We present a simple policy language that is expressive enough to model an asynchronous multi-agent access control system.

Syntax definition: Let T be the set of *types* which includes a special type "Agent" for agents and $Pred$ be a set of *predicates* such that each n-ary predicate has a type $t_1 \times \cdots \times t_n \rightarrow \{\top, \bot\}$, for some $t_i \in T$. Let V be a set of variables. Every variable in set V has a type. Consider $\boldsymbol{v}$ as a sequence of distinct variables. If $w \in Pred$, then $w(\boldsymbol{v})$ is called an *atomic formula*. L is a *logical formula* and consists of atomic formulas combined by logical connectives and existential and universal quantifiers. In the following syntax, *id* represents the identifier for the rules, and u is a variable of type Agent.

The syntax of access control policy language is as follows:

$$L ::= \top \mid \bot \mid w(\boldsymbol{v}) \mid L \vee L \mid L \wedge L \mid L \rightarrow L \mid \neg L \mid \forall v : t\ [L] \mid \exists v : t\ [L]$$
$$W ::= +w(\boldsymbol{v}) \mid\ -w(\boldsymbol{v}) \mid \forall v : t.\ W$$
$$W_s ::= W \mid W_s, W$$
$$\text{ActionRule} ::= id(\boldsymbol{v}) : \{W_s\} \leftarrow L$$
$$\text{ReadRule} ::= id(u, \boldsymbol{v}) : w(\boldsymbol{v}) \leftarrow L$$

Given a logical formula L, we define $fv(L) \subseteq V$ to be the set of all variables in atomic formulas in L, which occur as free variables in L. We extend fv to the set $\{W_s\}$ in the natural way.

An *action rule* has the form "$\alpha(\boldsymbol{v}) : E \leftarrow L$" such that logical formula L represents the condition under which the action is permitted to be performed. The set of signed atomic formulas E represents the effect of the action. $+w(\boldsymbol{v})$ in the effect means executing the action will set the value of $w(\boldsymbol{v})$ to true and $-w(\boldsymbol{v})$ means setting the value to false. In the case of $\forall v : t.W$ in the effect, the action updates the signed atom in W for all possible values of v. $\alpha(\boldsymbol{v})$ represents the name of the action rule. We can refer to the whole action rule as $\alpha(\boldsymbol{v})$.

We also stipulate for each action rule $\alpha(\boldsymbol{v}) : E \leftarrow L$ where $\boldsymbol{v} = (v_1, \ldots, v_n)$:

- v_1 is of type Agent and presents the agent performing the action.
- $fv(E) \cup fv(L) \subseteq \boldsymbol{v}$.
- $\{+w(\boldsymbol{x}), -w(\boldsymbol{x})\} \not\subseteq E$ where $\boldsymbol{x}$ is a sequence of variables.

A *read permission rule* has the form "$\rho(u, \boldsymbol{v}) : w(\boldsymbol{v}) \leftarrow L$" such that the logical formula L represents the condition under which the atomic proposition $w(\boldsymbol{v})$ is permitted to be read and $\rho(u, \boldsymbol{v})$ represents the name of the read permission rule. We can refer to the whole read permission rule as $\rho(u, \boldsymbol{v})$.

We also stipulate for each read permission rule $\rho(u, \boldsymbol{v}) : w(\boldsymbol{v}) \leftarrow L$ where $\boldsymbol{v} = (v_1, \ldots, v_n)$:

- u is of type Agent and presents the agent reading $w(\boldsymbol{v})$.
- $fv(L) \subseteq \{u, v_1, \ldots, v_n\}$.

Definition 1. *(Access control policy). An* access control policy *is a tuple* $(T, Pred, A, R)$ *where* T *is the set of* types*,* $Pred$ *is the set of* predicates*,* A *is the set of* action rules *and* R *is the set of* read permission rules*.*

Example 1. A conference paper review system policy contains the following properties for unassigning a reviewer from a paper:

- A chair is permitted to unassign the reviewers.
- If a reviewer is removed, all the corresponding subreviewers (subRev) should be removed from the system at the same time.

The unassignment action can be formalized as follows:

$$\text{delRev}(u, p, a) : \{\text{-rev}(p, a), \forall b : \text{Agent}.\ \text{-subRev}(p, a, b)\} \leftarrow \text{chair}(u) \wedge \text{rev}(p, a)$$

Example 1 shows how updating several variables synchronously can preserve integrity constraints. The RW framework is unable to handle such integrity constraint as it can only update one proposition at a time.

3.2 Access control system

Access control policy is a framework representing authorizations, actions and their effect in a system. Access control systems can be presented by a policy, a set of objects and corresponding substitutions.

We define Σ as a *finite set of objects* such that each object in Σ has a type. $\Sigma_t \subseteq \Sigma$ is the set of objects of type t. If V is the *set of variables*, then a *substitution* σ is a function $V \to \Sigma$ that respects types.

Definition 2. *(Atomic propositions). The set of* atomic propositions P *is defined as the set of predicates instantiated with the objects in* Σ*:*

$$P = \{w(\boldsymbol{v})\sigma \mid w \in Pred, \boldsymbol{v} \in V^* \text{ and } \sigma \text{ is a substitution}\}$$

A *system state* is a valuation of atomic propositions in P. A state s can be defined as a function $P \to \{\top, \bot\}$. We use $s[p \mapsto m]$ to denote the state that is like s except that it maps the proposition p to value m.

Instantiation of the rules: When a substitution applies to an action rule in the policy, it will extend to the variables in arguments, effects and logical formula in the natural way. If $a : e \leftarrow f$ is the instantiation of $\alpha(\boldsymbol{v}) : E \leftarrow L$ under the substitution σ, then $a = \alpha(\boldsymbol{v})\sigma$, $e = E\sigma$ and $f = L\sigma$.

This is the same for applying a substitution to the read permission rules in the policy. If $r : p \leftarrow f$ is the instantiation of $\rho(u, \boldsymbol{v}) : w(\boldsymbol{v}) \leftarrow L$ under the substitution σ, then $r = \rho(u, \boldsymbol{v})\sigma$, $p = w(\boldsymbol{v})\sigma$ and $f = L\sigma$.

Definition 3. *(Action, read permission). An* action *is an instantiation of a policy action rule. A* read permission *is an instantiated read permission rule.*

Since the number of objects is finite, each quantified logical formula will be expanded to a finite number of conjunctions (for $\forall$ quantifier) or disjunctions (for $\exists$ quantifier) of logical formulas during the instantiation phase. The universal quantifiers in the effect of actions will be expanded into a finite number of signed atomic propositions.

Definition 4. *(Access control system). An* access control system *is an access control policy instantiated with the objects in* Σ*.*

Definition 5. *(Action effect). Let* $a : e \leftarrow f$ *be an action in the access control system. Action* a *is* permitted to be performed *in state* s *if* f *evaluates to true in* s*. We also define:*

$$\mathit{effect}_+(a) = \{p \mid +p \in e\} \qquad \mathit{effect}_-(a) = \{p \mid -p \in e\}$$
$$\mathit{effect}(a) = \mathit{effect}_+(a) \cup \mathit{effect}_-(a)$$

3.3 Query language

Verification of the policy deals with the *reachability problem*, one of the most common properties arising in temporal logic verification. A state s is *reachable* if it can be reached in a finite number of transitions from the initial states.

In multi-agent access control systems, the transitions are made by the agents performing actions.

The query language determines the *initial states* and the *specification*. The syntax of the *policy query* is:

$$L ::= \top \mid \bot \mid w(\boldsymbol{v}) \mid \langle w(\boldsymbol{v})\rangle \mid L \vee L \mid L \wedge L \mid L \rightarrow L \mid \neg L \mid \forall v : t\ [L] \mid \exists v : t\ [L]$$
$$W ::= w(\boldsymbol{v}) \mid w(\boldsymbol{v})* \mid w(\boldsymbol{v})! \mid w(\boldsymbol{v})*! \mid \neg W$$
$$W_s ::= \text{null} \mid W_s, W$$
$$G ::= C : (L) \mid C : (L \text{ THEN } G)$$
$$\text{Query} ::= \{W_s\} \rightarrow G$$

where $w(\boldsymbol{v})$ is an atomic formula and C is a set of variables of type Agent.

In the above definition, G is called a nested goal if it contains the keyword THEN, otherwise it is called a simple goal. C is a *coalition of agents* interacting together to achieve the goal in the system. Also the agents in a coalition share the knowledge gained by reading system propositions or performing actions. The specification $\langle w(\boldsymbol{v})\rangle$ means $w(\boldsymbol{v})$ is readable by at least one of the agents in the coalition. *Initial states* are the states that satisfy the literals in $\{W_s\}$. Every literal W is optionally tagged with * when the value of atomic formula is fixed during verification, and/or tagged with ! when the value is initially known by at least one of the agents in the outermost coalition.

Example 2. One of the properties for a proper conference paper review system policy is that the reviewers (rev) of a paper should not be able to read other submitted reviews (submittedR) before they submit their own reviews. Consider the following query:

$$\{\text{chair}(c)*!, \neg\text{author}(p,a)*, \text{submittedR}(p,b), \text{rev}(p,a), \neg\text{submittedR}(p,a)\} \rightarrow$$
$$\{a\} : (\langle\text{review}(p,b)\rangle \wedge \neg\text{submittedR}(p,a) \text{ THEN } \{a,c\} : (\text{submittedR}(p,a)))$$

The query says "starting from the initial states provided, is there any reachable state that agent a can promote himself in such a way that he will be able to read the review of the agent b for paper p while he has not submitted his own review and after that, agent a and c collaborate together so that agent a can submit his review of paper p?". If the specification is satisfiable, then there exists a security hole in the policy and should be fixed by policy designers. In the above query, the value of chair(c) and author(p,a) is fixed and chair(c) is known to be true by the agent a at the beginning.

Instantiation of the policy query: An *instantiated query* or simply *query* is the policy query instantiated with a substitution. The query $i \rightarrow g$ is the instantiation of policy query $I \rightarrow G$ with substitution σ if $i = I\sigma$ and $g = G\sigma$.

For the query $i \rightarrow g$, we say g is *satisfiable* in an access control system if there exists a conditional sequence of actions called *strategy* (defined below) that makes the agents in the coalitions achieve the goal beginning from the initial states. The strategy is presented formally by the following syntax:

$$\textit{strategy} ::= \text{null} \mid a; \textit{strategy} \mid \text{if}(p)\ \{\textit{strategy}\}\ \text{else}\ \{\textit{strategy}\}$$

In the above syntax, p is an atomic proposition and a is an action. If a strategy contains a condition over the proposition p, it means the value of p determines the next required action to achieve the goal. p is known as an *effective proposition* in our methodology.

Definition 6. *(Transition relation). Let $s_1, s_2 \in S$ where S is the set of states, and ξ be a strategy. We use $s_1 \rightarrow_\xi s_2$ to denote "strategy ξ can be run in state s_1 and result in s_2", which is defined inductively as follows:*

- $s \rightarrow_{null} s$.
- $s \rightarrow_{a;\xi_1} s'$ *if*
  - *a is permitted to be performed in state s and*
  - *$s'' \rightarrow_{\xi_1} s'$ where s'' is the result of performing a in s.*
- $s \rightarrow_{if(p)\{\xi_1\}\ else\ \{\xi_2\}} s'$ *if:*
  - *If $s(p) = \top$ then $s \rightarrow_{\xi_1} s'$ else $s \rightarrow_{\xi_2} s'$.*

A set of states st_2 is reachable *from set of states st_1 through strategy ξ ($st_1 \rightarrow_\xi st_2$) if for all $s_1 \in st_1$ there exists $s_2 \in st_2$ such that $s_1 \rightarrow_\xi s_2$.*

Definition 7. *(State formula). If S is the set of states and $st \subseteq S$ then:*

- *f_{st} is a formula satisfying exactly the states in st: $s \in st \leftrightarrow s \models f_{st}$.*
- *st_f is the set of states satisfying f: $s \in st_f \leftrightarrow s \models f$.*

4 Model-checking and strategy synthesis

Our method uses backward search to find a strategy. The algorithm begins from the goal states st_g and finds all the states with transition to the current state, called pre-states. The algorithm continues finding pre-states over all found states until it gets all the initial states (success) or no new state could be found (fail).

The model-checking problem in this research is not a simple reachability question. As illustrated in Figure 1, the strategy is successful only if it works for all the outcomes of reading or guessing a proposition in the model. Thus, reading/guessing behaviour produces the need for a universal quantifier, while actions are existentially quantified. The resulting requirement has an alternation of universal and existential quantifiers of arbitrary length, and this cannot be expressed using standard temporal logics such as CTL, LTL or ATL.

Notation 2 *Assume f is a propositional formula. Then $p \in prop(f)$ if proposition p occurs in all formulas equivalent to f.*

Definition 8. *(Transition system). If action a is defined as $a : e \leftarrow f$ and st is a set of states, $PRE_a^\exists(st)$ is the set of states in which action a is permitted to perform and performing the action will make a transition to one of the states in st by changing the values of the propositions in the effect of the action. Let Lit^* be the set of literals that are tagged by $*$ in the query. Then:*

$$PRE_a^\exists(st) = \left\{ s \in S \mid s \models f, \forall l \in Lit^* : s \models l, s[p \mapsto \top \mid +p \in e][p \mapsto \bot \mid -p \in e] \in st \right\}$$

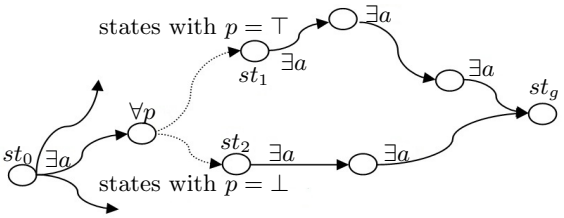

Fig. 1. Strategy finding method. Ovals represent sets of states. Solid lines show the existence of an action that makes a transition between two sets of states. Dashed lines are universally quantified over the outcome of reading or guessing the value of proposition p.

4.1 Finding effective propositions

Definition 9. *(Effective proposition). Atomic proposition p is* effective *with respect to st_0 as the set of initial states and st_g as the set of goal states if there exist a set of states st and strategies ξ_0, ξ_1 and ξ_2 such that $\xi_1 \neq \xi_2$ and:*

- $st_0 \rightarrow_{\xi_0} st$,
- $st \cap \{s \mid s(p) = \top\} \rightarrow_{\xi_1} st_g$,
- $st \cap \{s \mid s(p) = \bot\} \rightarrow_{\xi_2} st_g$ *and*
- $st \cap \{s \mid s(p) = \top\} \neq \emptyset$, $st \cap \{s \mid s(p) = \bot\} \neq \emptyset$.

Effective propositions are important for the following reason:

The value of proposition p is not specified in the query and is not known by the agents at the beginning. The agents need to know the value of p to select the appropriate strategy to achieve the goal. In the states of st, if the agent (or coalition of agents) knows the value of p, he will perform the next action without taking any risk. Otherwise, he needs to guess the value of p. This situation is risky and in the case of a wrong decision and may not be repeatable.

The algorithm provided in this paper is capable of finding effective propositions while searching for strategies, and then, is able to verify the knowledge of the agents about effective propositions in the decision states.

Proposition 1. *Let st_1, st_2 and st_g be sets of states and ξ_1 and ξ_2 be strategies such that $st_1 \rightarrow_{\xi_1} st_g$ and $st_2 \rightarrow_{\xi_2} st_g$. Suppose $p \in prop(f_{st_1}) \cap prop(f_{st_2})$, $st_d = st_{f_{st_1[\top/p]}} \cap st_{f_{st_2[\bot/p]}}$ and $s \in st_d$. Then if $s(p) = \top$, we conclude that $s \rightarrow_{\xi_1} st_g$, otherwise $s \rightarrow_{\xi_2} st_g$ will be concluded.*

Proof. A complete proof using structural induction is provided in [12].

Let st_g in proposition 1 be the set of goal states, st_d the set of states found according to the proposition 1 and st_0 the set of initial states. If there exist a strategy ξ_0 such that $st_0 \rightarrow_{\xi_0} st_d$, then by definition 9, the atomic proposition p is an effective proposition and therefore $st_d \rightarrow_{\text{if}(p)\ \{\xi_1\}\ \text{else}\ \{\xi_2\}} st_g$. The states in st_d are called *decision states*.

Example 3. Let $(T, Pred, A, R)$ be a simple policy for changing password in a system, where:

$T = \{\text{Agent}\}$

$Pred = \{\text{permission}(a : \text{Agent}), \text{trick}(a : \text{Agent}), \text{passChanged}(a : \text{Agent})\}$

$A = \{\text{setTrick}(a) : \{+\text{trick}(a)\} \leftarrow \neg\text{permission}(a),$

$\text{changePass}(a) : \{+\text{passChanged}(a)\} \leftarrow \text{permission}(a) \vee \text{trick}(a)\}$

We have excluded read permission rules, as they are not required in this particular example. In the above policy, the administrator of the system has defined a permission for changing password. The permission declares that one of the propositions permission(a) or trick(a) is needed for changing password. permission(a) is write protected for the agents and no action is defined for changing it. If an agent does not have permission to change his password, he can set trick(a) to true first and then, he will be able to change the password. This can be seen as a mistake in the policy.

Consider that we have just one object of type Agent in the system ($\Sigma_{Agent} = \{a_1\}$) and we want to verify the query $\{\} \rightarrow \{a\} : (\text{passChanged}(a))$. The only possible instantiation of the query is when a is assigned to a_1. As the initial condition is empty, the set of initial states contain all the system states ($st_0 = S$). The following procedures show how the strategy can be found:

$$f_{st_g} = \text{passChanged}(a_1)$$

We can find one set of states as the pre-state of st_g:

$$f_{PRE^{\exists}_{\text{changePass}(a_1)}(st_g)} = f_{st_1} = \text{permission}(a_1) \vee \text{trick}(a_1)$$

$$st_1 \rightarrow_{\text{changePass}(a_1)} st_g$$

f_{st_g} and f_{st_1} don't share any proposition and hence, there is no effective proposition occurring in both of them together. For the set st_1, we can find one pre-set:

$$f_{PRE^{\exists}_{\text{setTrick}(a_1)}(st_1)} = f_{st_2} = \neg\text{permission}(a_1)$$

$$st_2 \rightarrow_{\text{setTrick}(a_1);\text{changePass}(a_1)} st_g$$

The next step is to look for effective propositions occurring in f_{st_1} and f_{st_2}. For $p = \text{permission}(a_1)$ we have:

$$f_{st_1}[\top/p] = \top, f_{st_2}[\bot/p] = \top, f_{st_1}[\top/p] \wedge f_{st_2}[\bot/p] = \top$$

$$st_3 = st_\top = S \qquad st_3 \rightarrow_\xi st_g$$

$\xi = \text{if}(\text{permission}(a_1))\{\text{changePass}(a_1)\}$ else $\{\text{setTrick}(a_1); \text{changePass}(a_1)\}$

Since $st_0 \subseteq st_3$, the goal is reachable and we output the strategy.

Backward search transition filtering: If an action changes a proposition, the value of the proposition will be known for the rest of the strategy. So in backward search algorithm, we filter out the transitions that alter effective propositions before their corresponding decision states are reached.

4.2 Pseudocode for finding strategy

Consider P as the set of atomic propositions, A_C the set of all the actions that the agents in coalition C can perform, st_0 the set of initial states and st_g the set of goal states for simple goal g. K_C contains the propositions known by the agents in coalition C at the beginning (tagged with ! in the query). The triple (st, ξ, efv) is called *state strategy* which keeps the set of states st found during backward search, the strategy ξ to reach the goal from st and the set of effective propositions efv occurring in ξ. The pseudocode for the strategy finding algorithm is as follows:

input: P, A_C, st_0, st_g, K_C
output: strategy
state_strategies:=$\{(st_g, \text{null}, \emptyset)\}$
states_seen:=$\emptyset$
old_strategies:=$\emptyset$

while old_strategies≠state_strategies **do**
  old_strategies:=state_strategies
  for all $(st_1, \xi_1, efv_1) \in state\_strategies$ **do**
    for all $a \in A_C$ **do**
      if $effect(a) \cap efv_1 = \emptyset$ **then**
        $PRE := PRE_a^{\exists}(st_1)$
        if $PRE \neq \emptyset$ and $PRE \not\subseteq states\_seen$ **then**
          $states\_seen := states\_seen \cup PRE$
          $\xi :=$ "a;" $+ \xi_1$
          $state\_strategies := state\_strategies \cup \{(PRE, \xi, efv_1)\}$
          if $st_0 \subseteq PRE$ **then**
            output ξ
          end if
        end if
      end if
    end for

    for all $(st_2, \xi_2, efv_2) \in state\_strategies$ **do**
      for all $p \in P \backslash K_C$ **do**
        if $p \in prop(f_{st_1}) \cap prop(f_{st_2})$ **then**
          $PRE := st_{f_{st_1[\top/p]}} \cap st_{f_{st_2[\bot/p]}}$
          if $PRE \neq \emptyset$ and $PRE \not\subseteq states\_seen$ **then**
            $states\_seen := states\_seen \cup PRE$
            $\xi :=$ "if(p)" $+ \xi_1 +$ "else" $+ \xi_2$
            $state\_strategies := state\_strategies \cup \{(PRE, \xi, efv_1 \cup efv_2 \cup \{p\})\}$
            if $st_0 \subseteq PRE$ **then**
              output ξ
            end if
          end if

37: **end if**
38: **end for**
39: **end for**
40: **end for**
41: **end while**

The outermost while loop checks the fixed point of the algorithm, where no more state (or equivalently, state strategy) could be found in backward search. Inside the while loop, the algorithm traverses the state strategy set that contains $(st_g, \text{null}, \emptyset)$ at the beginning. For each state strategy (st, ξ, efv), it finds all the possible pre-states for st and appends the corresponding state strategies to the set. It also finds effective propositions and decision states by performing pairwise analysis between all the members of the state strategy set based on the proposition 1. The strategy will be returned if the initial states are found in backward search. The proof for the termination, soundness (If the algorithm outputs a strategy, it can be run over st_0 and results in st_g) and completeness (If some strategy exists from st_0 to st_g, then the algorithm will find one) is provided in [12].

Verification of the nested goals: To verify a nested goal, we begin from the inner-most goal. By backward search, all backward reachable states will be found and their intersection with the states for the outer goal will construct the new set of goal states. For the outer-most goal, we look for the initial states between backward reachable states. If we find them, we output the strategy. Otherwise, the nested goal is unreachable.

5 Knowledge vs. guessing in strategy

Agents in a coalition know the value of a proposition if: they have read the value before, or they have performed an action that has affected that proposition[3]. If a strategy is found, we are able to verify the knowledge of the agents over the strategy and specifically for effective propositions, using read permissions defined in the policy. Read permissions don't lead to any transition or action, and are used just to detect if an agent or coalition of agents can find out the way to the goal with complete or partial knowledge of the system. The knowledge is shared between the agents in a coalition.

To find agent knowledge over effective propositions, we begin from the initial states, run the strategy and verify the ability of the coalition to read the effective propositions. If at least one of the agents in the coalition can read an effective proposition before or at the corresponding decision states, then the coalition can find the path without taking any risk. In the lack of knowledge, agents should guess the value in order to find the next required action along the strategy.

[3] In this research, we do not consider reasoning about knowledge like the one in interpreted systems. This approach makes the concept of knowledge weaker, but more efficient to verify.

Pseudocode for knowledge verification over the strategy: Let g be a simple goal and (st_0, ξ, efv) be the state strategy where st_0 is the set of initial states, $st_0 \rightarrow_\xi st_g$ and efv is the set of effective propositions occurring in ξ. If C is the coalition of agents and K_C the knowledge of the coalition at the beginning, then the recursive function "KnowledgeAlgo" returns an annotated strategy with a string "Guess:" added to the beginning of every "if" statement in ξ, where the coalition does not know the value of the proposition inside if statement.

input: st_0, ξ, efv, C, K_C
output: Annotated strategy ξ'

function KnowledgeAlgo(st, ξ, efv, C, K_C)
  if ξ=null **then**
    return null
  end if
  for all $p \in efv, u_1 \in$ C **do**
    for all read permissions $\rho(u_1, \boldsymbol{o}) : p \leftarrow f$ **do**
      if $st \models f$ **then**
        $K_C := K_C \cup \{p\}$
      end if
    end for
  end for
  if $\xi = a; \xi_1$ **then**
    st' :=result of performing action a in st
    return "a;"+
        KnowledgeAlgo($st', \xi_1, efv, C, K_C \cup effect(a)$)
  else if $\xi =$ if(p){ξ_1} else {ξ_2} **then**
    if $p \in K_C$ **then**
      str :=""
    else
      str :="Guess: "
    end if
    return str+ "if(p){"+
      KnowledgeAlgo($st_{f_{st} \wedge p}, \xi_1, efv \backslash \{p\}, C, K_C$) + "} else {"+
      KnowledgeAlgo($st_{f_{st} \wedge \neg p}, \xi_2, efv \backslash \{p\}, C, K_C$) +"}"
  end if
end function

Knowledge verification for nested goals: To handle knowledge verification over the strategies found by nested goal verification, we begin from the outermost goal. We traverse over the strategy until the goal states are reached. For the next goal, all the accumulated knowledge will be transferred to the new coalition if there exists at least one common agent between the two coalitions. The algorithm proceeds until the strategy is fully traversed.

	RW(Algo-1)		PoliVer algorithm	
Query	Time	Memory	Time	Memory
Query 4.2	2.05	18.18	0.27	3.4
Query 4.3	0.46	9.01	0.162	6.68
Query 4.4	6.45	59.95	0.52	6.61
Query 6.4	9.10	102.35	0.8	12.92
Query 6.8	20.44	222.02	0.488	7.30

Fig. 2. A comparison of query verification time (in second) and runtime memory usage (in MB) between RW and PoliVer.

6 Experimental results

We implemented the algorithms as a policy verification tool called *PoliVer* by modifying the AcPeg model-checker, which is an open source tool written in Java. First, we changed the parser in order to define actions and read permissions in the policy as in section 3.1. The query language also changed to support queries of the form defined in section 3.3. Second, we implemented the strategy finding algorithm in the core of AcPeg and then, applied knowledge verification algorithm over strategies found.

One of the outcomes of the implementation was the considerable reduction of binary decision diagram (BDD) variable size compared to RW. In RW, there are 7 knowledge states per proposition and therefore, an access control system with n propositions contains 7^n different states. Our simplification of knowledge-state variables results in 2^n states. The post-processing time for knowledge verification over found strategies is negligible compared to the whole process of strategy finding, while produces more expressive results.

We encoded authorization policies for a conference review system (CRS), employee information system (EIS) and student information system (SIS) in [1] into our policy language. We compared the performance in terms of verification time and memory usage for the queries: Query 4.2 for CRS with 7 objects (3 papers and 4 agents) that looks for strategies which an agent can promote himself to become a reviewer of a paper, Query 4.3 for CRS which is a nested query that asks if a reviewer can submit his review for a paper while he has read the review of someone else before, Query 4.4 with 4 objects for CRS with five-level nested queries that checks if an agents can be assigned as a pcmember by the chair and then resign his membership, Query 6.4 with 18 objects for EIS which evaluates if two managers can collaborate to set a bonus for one of them and Query 6.8 for SIS with 10 objects that asks if a lecturer can assign two students as the demonstrator of each other.

Figure 2 shows a considerable reduction in time and memory usage by the proposed algorithm compared to Algo-1 in RW (Algo-1 has slightly better performance and similar memory usage compared to Algo-0). As a disadvantage for both systems, the verification time and state space grow exponentially when more objects are added. But this situation in our algorithm is much better than RW. Our experimental results demonstrates the correctness of our claim in

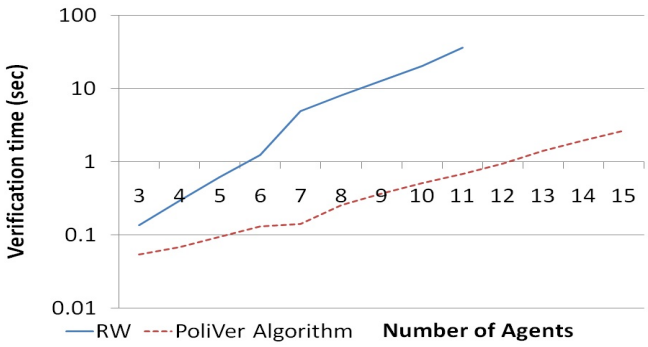

Fig. 3. Verification time vs. number of agents for RW and PoliVer (Query 6.8)

practice by comparing the verification time of Query 6.8 for different number of agents. Figure 3 sketches the verification time for both algorithms for different number of agents in logarithmic scale. The verification time in RW increases as 2.5^n where n is the number of agents added, while the time increases as 1.4^n in our algorithm. Note that this case study does not show the worst case behaviour when the number of agents increases[4].

7 Conclusion and future work

Our language and tool is optimised for analysing the access control policies of web-based collaborative systems such Facebook, LinkedIn and Easychair. These systems are likely to become more and more critical in the future, so analysing them is important. More specifically, in this work:

- We have developed a policy language and verification algorithm, which is also implemented as a tool. The algorithm produces evidence (in the form of a strategy) when the system satisfies a property.
- We remove the requirement to reason explicitly about knowledge, approximating it with the simpler requirement to reason about readability as it is sufficient in many cases. Compared to RW that has 7^n states, we have only 2^n states in our approach (where n is the number of propositions). Also, complicated properties can be evaluated over the policy by the query language provided.
- We detect the vulnerabilities in the policy that enable an attacker to discover the strategy to achieve the goal, when some required information is not accessible. We introduce the concept of effective propositions to detect such vulnerabilities.
- A set of propositions can be updated in one action. In the RW framework, each write action can update only one proposition at a time.

[4] The tool, case studies and technical reports are accessible at: `http://www.cs.bham.ac.uk/~mdr/research/projects/11-AccessControl/poliver/`

Future work: For the next step, we intend to cover large applications like Facebook and Easychair in our case studies.

Acknowledgements: We would like to thank Microsoft Research as Masoud Koleini is supported by a Microsoft PhD scholarship. We also thank Moritz Becker and Tien Tuan Anh Dinh for their useful comments and Joshua Phillips for the performance and quality testing of the release version of PoliVer.

References

1. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems through model checking. J. Comput. Secur. **16**(1) (2008) 1–61
2. Becker, M.Y.: Specification and analysis of dynamic authorisation policies. In: CSF '09: Proceedings of the 2009 22nd IEEE Computer Security Foundations Symposium, Washington, DC, USA, IEEE Computer Society (2009) 203–217
3. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: of Lecture Notes in Computer Science. Volume 4130., Springer (2006) 632–646
4. Naldurg, P., Campbell, R.H.: Dynamic access control: preserving safety and trust for network defense operations. In: SACMAT '03: Proceedings of the eighth ACM symposium on access control models and technologies, New York, NY, USA, ACM (2003) 231–237
5. Fisler, K., Krishnamurthi, S., Meyerovich, L.A., Tschantz, M.C.: Verification and change-impact analysis of access-control policies. In: ICSE '05: Proceedings of the 27th international conference on software engineering, New York, NY, USA, ACM (2005) 196–205
6. Becker, M.Y., Gordon, A.D., Fournet, C.: SecPAL: Design and semantics of a decentralised authorisation language. Technical report, Microsoft Research, Cambridge (Sep. 2006)
7. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management framework. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (May 2002) 114–130
8. Bell, D., LaPadula., L.J.: Secure computer systems: Mathematical foundations and model. Technical report, The Mitre Corporation (1976)
9. Bell, D.E.: Looking back at the bell-la padula model. In: ACSAC '05: Proceedings of the 21st Annual Computer Security Applications Conference, Washington, DC, USA, IEEE Computer Society (2005) 337–351
10. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. ACM Trans. Inf. Syst. Secur. **13**(3) (2010) 1–28
11. Fox, M., Long, D.: Pddl2.1: An extension to pddl for expressing temporal planning domains. Journal of Artificial Intelligence Research **20** (2003) 61–124
12. Koleini, M., Ryan, M.: A knowledge-based verification method for dynamic access control policies. Technical report, University of Birmingham, School of Computer Science, Available at: `http://www.cs.bham.ac.uk/~mdr/research/projects/11-AccessControl/poliver/` (2010)