X-Policy: Knowledge-based Verification Tool for Dynamic Access Control Policies

Hasan Qunoo and Mark Ryan

School of Computer Science, University of Birmingham
Birmingham, United Kingdom
{H.Qunoo, M.D.Ryan}@cs.bham.ac.uk

Abstract

Verifying the correctness of large, complex and dynamic access control policies by hand is insufficient and error-prone. We present X-policy, a knowledge-based verification tool that can analyse the system's vulnerabilities where the attackers can act as a coalition of users, use the system, share knowledge and collaborate with each other to achieve the attack. We present a policy language that is able to express dynamic access control policies and a corresponding query language. We model the EasyChair conference management system and we analyse in details three security properties of EasyChair using our model. Finally, we compare our results with similar tools and we discuss the results and the advantages of our tool and approach.

Keywords: *X-policy, policy language, query language, EasyChair conference management system*

1. Introduction

Web-based collaborative systems like social networking websites, conference management systems, and application processing systems are all examples of central systems that give users the ability to create and control access to their data. Access to data in these systems is dynamic; it depends on the state of the system and its configuration. Users, with the right permissions and in the right system state, can acquire information about the system state or execute complex and compound actions that causes the system state to evolve into another state.

For instance, large conference management systems like EasyChair , iChair, HotCRP are widely used to manage academic conferences. The size and the complexity of the system policy makes it difficult to analyse its security and correctness properties by hand. In Conference Management System, for example, one might want to verify properties like "a single user cannot review the same paper twice". or "an author cannot review her own paper". It is also desirable that in the case where the system policy fails the property, the tool outputs a counterexample strategy to help the policy designers to identify and fix the problem.

It is also evidently important in the case of multi-agent collaborative systems to reason about user's knowledge about the state of the system. It allows us to model the user's allowed behaviour. As the access to the system is dynamic and depends on its state, the attacker needs to gather information about the system to be able to evaluate whather or not she can perform a certain action. Such a method allows the attacker to avoid being logged and/or flagged by the system monitor/administrator in the case of requesting prohibited information. It also allows us to reason about situations in which attackers form a coalition where they share knowledge about the system. Alternatively, the user can *guess* the value of these variables. The result strategy is called *guessing strategy*, first introduced by RW [7]. Using *guessing strategy* analysis can produce strategies that derive the system to undesirable states where the user cannot backtrack from or unintentionally destroy a certain piece of information while carrying on the strategy.

We propose *X-Policy* , a knowledge-based verification tool for dynamic access control policies. *X-Policy* 's modelling language with its corresponding query language and verification algorithm are based on concepts of RW [7], but extends it with the ability to express and analyse compound actions. *X-Policy* 's modelling language allows us to describe access control system as two sets of rules: *read permission rules* and *action execution rules*. A *read permission rule* allows us to specify the permission conditions that a user needs to satisfy to be able to read the value of a certain system variable. An *action execution rule* allow us to specify the system operation and its execution permissions where the user can change one or more of the system variables as a compound action (all or none). We assume that the user has knowledge of a system variable if she or any other member of her coalition has previously read that variable. Users can share information with the coalition as they can communicate with each other using an outside channel. We also assume that the user will maintain that knowledge of the variable if the user has performed an action that changed the value of that variable, and that the coalition has an exclusive access to the system and has a complete knowledge of the system policy while executing the attack (strategy). While such assumptions give the attackers what seems as an unfair advantage, it is certainly desired that the system be tested in the most rigorous way. It also covers scenarios where attackers automate their attack by simulating access requests to the system. Such attacks are common in web-based collaborative systems and very important to model. *X-Policy* allows the user to input a system policy specification and a property (query). *X-Policy* then verifies whether or not the system satisfies the properties. If the system fails, it outputs a strategy that shows how the attacker or the coalition of attackers can achieve the goal.

One of the most important features of *X-Policy* is its ability to express compound actions that update multiple variables. This is crucial for modelling collaborative systems. For example, when a user responds to an invitation to take a certain role, the system updates the status of that invitation and then update the user's role according to her answer. Similarly, once a user account is deleted from the system all the user's roles will be consequently deactivated.

1.1 Our Contribution

This paper contributs the following:

- Modeling language that expresses dynamic access control policies with compound actions that update multiple variables.
- Knowledge-based verification algorithm that verifies properties over an access control policy that has com-pound actions.
- An automated tool, called *X-Policy*, which implements the algorithm.
- A case study in which we focus on analysing three security properties of ***EC*** model [6], which is based on EasyChair conference management system.

1.2 Paper Structure

We introduce our modeling language in Section 2. X-Policy knowledge-based model checking algorithm is discussed in Section 3. We discuss the tool implementation and EC model and properties in Section 4. We compare our tool with similar tools/approaches and we discuss the results and the advantages of our tool and approach in Section 5. Related work is explained in Section 6. Conclusion and Future work is discussed in Section 7.

2. X-Policy Modeling Language

2.1 Access Control System

An access control system S is defined as two sets of rules: *read permission rules and action execution rules.*

Let T be a set of types, which includes a special type `Agent` for agents, also let *Pred* be a finite set of predicates.

Each *n*-ary predicate has a signature, $t_1 \times \ldots \times t_n \rightarrow \{\mathrm{T}, \perp\}$, where $t_i \in T$. For example, in the case of a conference review system, *T* can include Paper which we can define using type definition statement of the form:

Type Paper;

and *Pred* can include the predicate

$$\text{Author} : \text{Agent} \times \text{Paper} \rightarrow \{\mathrm{T}, \perp\}.$$

Note that each system will require a different list of predicates. We assume a set of variables *V*, each with a type. If $p \in Pred\ and\ \vec{x} = x_1, \cdots, x_q$ is a sequence of variables of the appropriate type, then $p(\vec{x})$ is an *atomic formula*.

2.1.1 Read Permission Rules

These definitions allow us to define whether or not a user has permission to access the truth value of an *atomic formula* and is of the form

$$p(\vec{y})\{read : formula;\}$$

where $p \in Pred$ and the variables in $\vec{x}$ occur in $\vec{x}$. In CMS example, a read permission rule can be: [numbers=left,label=foo,mathescape=true] Reviewer(p,a) read : PCmember(user) Author(p, user); This expresses the rule that whether or not an agent is a reviewer of a paper is readable by all the pcmembers except the author of that paper. Note that user is a special agent that represent the user requesting access.

2.1.2 Action Execution Rules

An action execution rule allows the user to change the truth values of an atomic formula and is of the form

Action Actionname $(\overleftarrow{x})$:- {writebody } {*formula*}

where writebody is an expression formed from the following BNF:

writebody ::= assignment | for ($v : t$) { writebody }
| writebody writebody

where v is a variable of the type t and an assignment is of the form $p(\overleftarrow{y}) := \top$; or $p(\overleftarrow{y}) := \bot$;. We allow an *atomic formula* $p(\overleftarrow{y})$ to occur at most once at the left of ":=" in an action to avoid ambiguity in computing the action effect. The assignment statements within the same action can be written in any order. All free variables in an assignment must be declared either in a surrounding for-statement or in Actionname statement. Intuitively, a for-statement in an action is a 'macro' that is interpreted as multiple *assignment* statements. *formula* defines the conditions for an agent u $\in$ Agent to execute an action $act \in$ Actions where Actions is a finite set of all the actions in the defined system. *formula* is a logical formula which is defined using atomic formulae and logical connectors: $\neg$ (negation), $\wedge$ (conjunction), $\vee$ (disjunction), $\rightarrow$ (implication), $\exists$ and $\forall$ (existential and universal quantification over variables of the appropriate type). The variables that occur in *formula* are required to be either in $\overleftarrow{x}$ or u. The *formula* defines the conditions for agents to execute these actions as functions on its state.

Conversely, an action execution rule, in a CMS, can be: [numbers=left,label=foo,mathescape=true] Action DeletePCmember(a:Agent) PCmember(a) :=false; Chair(a):=false; for(p:Paper) Reviewer(p,a):=false; Chair(user); This expresses the action of deleting a PCmember and all his reviewing assignments. Only Chair can execute this action.

2.1.3 Access Control Model

A system S described in the description part, as in Section 2.1, is only a template. To perform model-checking, a concrete instance based on the template needs to be constructed. This task is done through a run-statement. The syntax of the run-statement is:

```
RunStatement    ::=     run for NumberTypePair
                        ("," NumberTypePair ) *
NumberTypePair  ::=     d TypeName
```

where d is an integer. When a run-statement is executed, the tool creates a model M which assigns each defined type t a fixed number of elements σ_t as specified in the run-statement. We define $\sigma = \sigma_{t_1} \cup \ldots \cup \sigma_{t_n}$ as the set of all the individuals defined by M. We assume $\sigma_{t_1} \cap \sigma_{t_2} = \varnothing$ whenever t_1 and t_2 are distinct. These elements are then used to instantiate the relations defined by the parametrised predicates and actions. Once these two steps are finished, a concrete model with fixed size is established, on which model-checking is performed. If $p \in Pred$, $act \in Actions$, and $\overleftarrow{\alpha}$ is a sequence of individuals of the appropriate type then $p(\overleftarrow{\alpha})$ is a ground atomic formula and $act(\overleftarrow{\alpha})$ is an instantiated action. The finite set of ground atomic formulae is P and $Actions^*$ is the finite set of instantiated actions. Models of other sizes are not considered by the checking. Although large models may contain errors that small models cannot display, small models are still extremely useful for finding errors as in Alloy [5] and RW [7].

2.2.1 For-loops

We describe the semantics of for-loops in the context of a model M , with $\sigma_t = \{v_1, \ldots, v_k\}$ the set of individuals in M of the type t . Let $act \in$ Actions. We then transform each for-statement to its equivalent multiple *assignment* statements. For example the following for-statement:

$$\text{for}\ (v:t)\ \{p(\overleftarrow{\gamma_1}, v, \overleftarrow{\gamma_2}) := \bot;\}$$

is in the write action $act(\overleftarrow{x})$ where $\overleftarrow{\gamma_1}$ and $\overleftarrow{\gamma_2}$ are subsequences of other parameters. This for-statement is transformed to:

$$p(\overleftarrow{\gamma_1}, v_1, \overleftarrow{\gamma_2}) := \bot;\ \ldots\ p(\overleftarrow{\gamma_1}, v_k, \overleftarrow{\gamma_2}) := \bot\,;$$

We apply this process repeatedly until we have no for-statement in our action. The same process will applied to all actions.

2.2.2 Effect of Actions

Let act be a loop-free action in Actions and $\overleftarrow{\alpha}$ a sequence of individuals of the appropriate type for act . We define the result of running the instantiated action $act(\overleftarrow{\alpha})$. We apply the functions: effect$^+$() and effect$^-$() which compute the positive and the negative effect of the instantiated loop-free action $act(\overleftarrow{\alpha})$ as following:

$$\text{effect}^+(act(\overleftarrow{\alpha})) = \{\ p(\overleftarrow{\beta})\ |\ p(\overleftarrow{\beta}) := \mathrm{T}\ \text{occurs in}\ act(\overleftarrow{\alpha})\}$$
$$\text{effect}^-(act(\overleftarrow{\alpha})) = \{\ p(\overleftarrow{\beta})\ |\ p(\overleftarrow{\beta}) := \bot\ \text{occurs in}\ act(\overleftarrow{\alpha})\}$$

where all the values of $\overleftarrow{\beta}$ are members of σ . effect$^+(act(\overleftarrow{\alpha}))$ and effect$^-(act(\overleftarrow{\alpha}))$ return two mutually exclusive sets for all $act(\overleftarrow{\alpha}) \in Actions^*$

Example

In the ***EC*** model, if the following line

run for 3 Paper, 4 Agent

is put after the system definitions. The algorithm will assign three elements to the set $\text{P}Paper\,|$ and four elements to the set $Agent\,|$ when the statement gets executed. As a result, the set $Paper$ becomes $\{p_1, p_2, p_3\}$, and the set $Agent$ becomes $\{a_1, a_2, a_3, a_4\}$. Then, all the defined predicates are instantiated. For example, the predicate $Author(paper : Paper, agent : Agent)$ is instantiated to twelve ground atomic formalas: from $Author(p_1, a_1)$ to $Author(p_3, a_4)$. Finally, all the defined actions are instantiated the same way. For example, the action $DeletePCmember(agent : Agent)$ is instantiated to four actions : from $DeletePCmember(a_1)$ to $DeletePCmember(a_4)$. Subsequently, effece$^-$ ($DeletePCmember(a_1)$) is the set:

{ $PCmember(a_1)$, $Chair(a_1)$, $Reviewer(p_1,a_1)$, $Reviewer(p_2,a_1)$, $Reviewer(p_3,a_1)$ }
while
effect$^+$ ($DeletePCmember(a_1)$) $=\varnothing$.

2.3 Query

Following the run-statement, a query can be specified based on [7]. A query, taking the form of

$$\text{check } \{ \; L \;\; \| \, I \rightarrow C_1 : (G_1 \; THEN \,|\, C_2 : (G_2 \ldots THEN \,|\, C_n : (G_n) \ldots)) \}$$

where L, defines a number of quantified variables used in I, G_i and C_i; I (optional) is a list of conditions based on which the goal, defined by G_i, is to be achieved; and C_i defines a coalition of agents who work together, intending to achieve the goal. Its meaning is: Are there strategies or guessing strategies available for agents in C_1, C_2, …, and C_n, such that, if conditions in I are true, the agents in C_1 can achieve a state in which the goal G_1 is satisfied, and then (in that state) the agents in C_2 can achieve the goal G_2, …, and finally the agents in C_n can achieve the goal G_n. What this nested goal describes is a sequencing of actions performed by the agents in C_1, C_2, …, and C_n. However, this sequencing is achieved only if the conditions defined in I do not enable agents in C_i make other subgoals true while on the way of achieving G_i. The check statement goes through an instantiation process where the quantified variable defined in L and used in I, G_i and C_i are replaced with the appropriate individuals from σ to conform with model similar to the way we handle actions and atomic formulae in the previous example.

Example

The following one level nested query

```
  check   E   dist   a,b,c:   Agent,   p:   Paper   ||   Chair(c)*!
Author(p,a)*!  and  Submitted-review(p,b,c)*!  and   Submitted-
review(p,a,b)!  and  PCmember(a)*!  and   Reviewer(p,a)!  and
Subreviewer(p,b,a)*!    and       Subreviewer(p,c,a)*!    and
Subreviewer(p,a,a)*!      ->      a:([Review(p,b,c)]      THEN
a,c:(Submittedreview(p,a,b)))
```

represents the property that a reviewer a should not read the review of another reviewer before she submits her own review for that paper. The variables followed by '!' means that these variables in I are known to the agents in C_i, in this case agents a and c. One the other hand variables followed by '*!' means that the coalition know the value of that variable but must not change that value during the strategy. Similarly, we can see that G_1=[Review(p,b,c)] is a *reading goal* which means agent a knows that he can *read* the variable Review(p,b,c) while G_2 means that agent a and c can reach a state where they know that they can *write* Submittedreview(p,a,b) to true as it is a *making goal* which we distinguish by the brackets {}.

2.4 Strategy

A strategy is a sequence of read or write actions where in each step, there is an agent in σ_C who can execute an action.

$$\begin{aligned}\text{Strategy::=}\quad & \text{null} \mid \text{a: } act(\overleftarrow{\alpha})\text{ ; } \mid \text{Strategy Strategy}\\ & \mid \text{if(} p(\overleftarrow{\alpha})\text{)\{Strategy\} else \{Strategy\}}\end{aligned}$$

where $a \in \sigma_C$, $p \in Pred$ and $act \in$ Actions and α is a sequence of individuals where its members are in σ . Note that only one agent can act at a time as the agents perform actions asynchronously which is a realistic assumption in computer systems.

3. X-Policy Model Checking

Given a query Q, such as the one defined in 2.3, the task of the *X-Policy* model-checking algorithm is to figure out whether or not the strategies queried by Q exist, and if they exist, output at least one of them.

3.1 The transition system

The algorithm is built around the knowledge of the state of the system S that the considered coalition C has at each step of implementing its strategy. This is achieved by extending the idea of [7] to handle complex executable actions instead of simple assignments. Obviously there is a set of knowledge states each of which is sufficient for C to regard its goal as achieved. This is so when C knows that the formulae in some appropriate combination of the involved making goals are true, or that enough is known to work out the truth values of the formulae in the reading goals. We denote the set of the knowledge states from which C can deduce that its goal is achieved by $K_{G|}$. Each step of a strategy takes C from a knowledge state to a possibly richer one until a state in $K_{G|}$ is reached. A knowledge state combines knowledge of the initial state of the system, that is, the state of the system at the beginning of executing a strategy, and knowledge of its current state. Executing actions contribute the knowledge of the current values of the assigned variables, which has been just given to them. This means that learning and changing the system are done simultaneously. Sampling steps contribute C's knowledge on both the current and the initial value of the sampled variable. Overwriting without sampling in advance destroys the prospect to learn the initial values of the variables. Strategies are supposed to take C from its initial knowledge state k_{init} to one in K_G from which its goal is deemed as achieved.

To describe C's knowledge on $p(\overleftarrow{\alpha})$, we use four knowledge variables. For each $p(\overleftarrow{\alpha}) \in P$, we have

$v_{0p(\overleftarrow{\alpha})}$	*is true if C knows the initial value of $p(\overleftarrow{\alpha})$*
$t_{0p(\overleftarrow{\alpha})}$	*is true if C knows initially $p(\overleftarrow{\alpha})$ is true.*
$v_{p(\overleftarrow{\alpha})}$	*is true if C knows the current value of $p(\overleftarrow{\alpha})$.*
$t_{p(\overleftarrow{\alpha})}$	*is true if C knows currently $p(\overleftarrow{\alpha})$ is true.*

When overwriting $p(\overleftarrow{\alpha})$ to true, $v_{p(\overleftarrow{\alpha})}$ and $t_{p(\overleftarrow{\alpha})}$ both become true, but $v_{0p(\overleftarrow{\alpha})}$ and $t_{0p(\overleftarrow{\alpha})}$ do not change, because overwriting does not contribute C's knowledge on $p(\overleftarrow{\alpha})$'s initial value. When overwriting $p(\overleftarrow{\alpha})$ to false, $v_{p(\overleftarrow{\alpha})}$ becomes true; $t_{p(\overleftarrow{\alpha})}$ becomes false; both $v_{0p(\overleftarrow{\alpha})}$ and $t_{0p(\overleftarrow{\alpha})}$ do not change. When sampling $p(\overleftarrow{\alpha})$, $v_{0p(\overleftarrow{\alpha})}$ and $v_{p(\overleftarrow{\alpha})}$ both become true and $t_{0p(\overleftarrow{\alpha})}$ and $t_{p(\overleftarrow{\alpha})}$ both become false if $p(\overleftarrow{\alpha})$ turns out to be false, or $t_{0p(\overleftarrow{\alpha})}$ and $t_{p(\overleftarrow{\alpha})}$ both become true if $p(\overleftarrow{\alpha})$ turns out to be true. Since the contents of $t_{0p(\overleftarrow{\alpha})}$ and $t_{p(\overleftarrow{\alpha})}$ are irrelevant when $p(\overleftarrow{\alpha})$ 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 $v_{0p(\overleftarrow{\alpha})}$, $t_{0p(\overleftarrow{\alpha})}$, $v_{p(\overleftarrow{\alpha})}$ and $t_{p(\overleftarrow{\alpha})}$ as independent variables.

A knowledge state is given by the quadruple (V_0, T_0, V, T), where

$$V_0 = \{p(\overleftarrow{\alpha}) \in P \mid v_{0p(\overleftarrow{\alpha})} \textit{ is } \mathrm{T}\}, T_0 = \{p(\overleftarrow{\alpha}) \in P \mid t_{0p(\overleftarrow{\alpha})} \textit{ is } \mathrm{T}\},$$

$$V = \{p(\overleftarrow{\alpha}) \in P \mid v_{p(\overleftarrow{\alpha})} \textit{ is } \mathrm{T}\}, \text{ and } T = \{p(\overleftarrow{\alpha}) \in P \mid t_{p(\overleftarrow{\alpha})} \textit{ is } \mathrm{T}\}.$$

3.2 Representation of k_{init}

k_{init} is the state where C knows nothing about the state of S, except on the values of the variables marked by '!' in the conditions defined by I. This means that for all members of P which also occur in I and are marked by '!', C knows their values. However, for all the other members of P, C does not know their values initially. Now for each variable $p(\overleftarrow{\alpha}) \in P$, we have four knowledge variables $v_{0p(\overleftarrow{\alpha})}$, $t_{0p(\overleftarrow{\alpha})}$, $v_{p(\overleftarrow{\alpha})}$ and $t_{p(\overleftarrow{\alpha})}$ to describe C's initial and current knowledge about it. We use a boolean expression composed of the knowledge variables to represent $k_{\mathrm{P}init|}$. This boolean expression is then represented by a BDD in the course of the *X-Policy* model-checking.

We divide members of P into three mutually-exclusive subsets. P^+ is the set for members of P which only occur positively in I and are marked by '!' and by '*!'. P^- is the set for members of P which only occur negatively in I and are marked by '!' and by '*!'. $P^{\mathrm{Po|}}$ is the set for all the other members of P. Now for each $p(\overleftarrow{\alpha}) \in P^+$, we use $(v_{0p(\overleftarrow{\alpha})} \wedge t_{0p(\overleftarrow{\alpha})} \wedge v_{p(\overleftarrow{\alpha})} \wedge t_{p(\overleftarrow{\alpha})})$ to represent C's initial knowledge about $p(\overleftarrow{\alpha})$. In the case that $p(\overleftarrow{\alpha}) \in P^-$, we use $(v_{0p(\overleftarrow{\alpha})} \wedge \neg t_{0p(\overleftarrow{\alpha})} \wedge v_{p(\overleftarrow{\alpha})} \wedge \neg t_{p(\overleftarrow{\alpha})})$ to represent C's initial knowledge about it; and finally, if $p \in P^0$, we use $(\neg v_{0p(\overleftarrow{\alpha})} \wedge \neg t_{0p(\overleftarrow{\alpha})} \wedge \neg v_{p(\overleftarrow{\alpha})} \wedge \neg t_{p(\overleftarrow{\alpha})})$ to represent C's initial knowledge about it. Therefore, the representation of $k_{\mathrm{P}init|}$ by the knowledge variables is the conjunction of all the representations of the above forms for members of P.

3.3 Representation of K_G

Given a formula G describing the goal we want to produce the knowledge representation of the goal states. G is a conjunction and disjunction combination of reading goals $[l]$ and making goals $\{l\}$, where l in each case is a boolean combination of members of P.

We want to represent G as a set of knowledge states, being those in which the goal is known to be true. A set of knowledge states can be represented by a formula over the propositions $\{v_{0p(\bar{\alpha})}, t_{0p(\bar{\alpha})}, v_{p(\bar{\alpha})}, t_{p(\bar{\alpha})} \mid p(\overleftarrow{\alpha}) \in P\}$. Note that $v_{0p(\bar{\alpha})}$ is true in a state if $p \in V_0$, $t_{0p(\bar{\alpha})}$ is true if $p(\overleftarrow{\alpha}) \in T_0$, and similarly for $v_p(\overleftarrow{\alpha})$ and $t_{p(\bar{\alpha})}$.

Suppose an agent's knowledge of the state of the system is represented by V, T. Then a formula over $\{v_{0p(\bar{\alpha})}, t_{0p(\bar{\alpha})}, v_p(\overleftarrow{\alpha}), t_p(\overleftarrow{\alpha}) \mid p(\overleftarrow{\alpha}) \in P\}$ expressing the agents ability to determine that l is true may be constructed as follows:

- if $v_{p(\bar{\alpha})}$ is true, then substitute $t_{p(\bar{\alpha})}$ for $p(\overleftarrow{\alpha})$ in l. This covers the case that the agent knows the value of $p(\overleftarrow{\alpha})$.
- if $v_{p(\bar{\alpha})}$ is false, then replace l with a version in which T is substituted for $p(\overleftarrow{\alpha})$ and another in which $\perp$ is substituted for $p(\overleftarrow{\alpha})$. This covers the case that the agent does not know $p(\overleftarrow{\alpha})$.

Thus, the formula expressing the agent's ability to determine that l is true is

$$(l[t_{p(\bar{\alpha})}/p(\overleftarrow{\alpha})] \wedge v_{p(\bar{\alpha})}) \vee (l[\mathrm{T}/p(\overleftarrow{\alpha})] \wedge l[\perp/p(\overleftarrow{\alpha})] \wedge \neg v_{p(\bar{\alpha})})$$

In the following definition, we generalise this formula to consider the effect of all the $p(\overleftarrow{\alpha}) \in P$.

Definition Let V, T be the knowledge held by an agent, and l a formula over the propositions P. The propositions $v_{p(\bar{\alpha})}$ and $t_{p(\bar{\alpha})}$ signify $p(\overleftarrow{\alpha}) \in V$ and $p(\overleftarrow{\alpha}) \in T$, respectively. The $\gamma_{V,T} l$ is represented in the formula:

$$\begin{aligned}(\bigwedge_{S \subseteq P} l[(v_{p(\bar{\alpha})} \to p(\overleftarrow{\alpha}))/p(\overleftarrow{\alpha}) \mid p(\overleftarrow{\alpha}) \in S]\\ [(v_{p(\bar{\alpha})} \wedge p(\overleftarrow{\alpha}))/p(\overleftarrow{\alpha}) \mid p(\overleftarrow{\alpha}) \in P \setminus S])\\ [t_{p(\bar{\alpha})}/p(\overleftarrow{\alpha}) \mid p(\overleftarrow{\alpha}) \in P]\end{aligned}$$

$\gamma_{V,T} l$ expresses the agent's ability to determine that l is true. The $S \subseteq P$ produces all the possible combinations of T and $\perp$ to substitute $p(\overleftarrow{\alpha})$s such that $v_{p(\bar{\alpha})}$ is false. Note that $v_{p(\bar{\alpha})} \to p(\overleftarrow{\alpha})$ is T when $v_{p(\bar{\alpha})}$ is false, and $p(\overleftarrow{\alpha})$ otherwise; similarly, $v_{p(\bar{\alpha})} \wedge p(\overleftarrow{\alpha})$ is $\perp$

if $v_{p(\overleftarrow{\alpha})}$ is false and $p(\overleftarrow{\alpha})$ otherwise. Hence, S just enumerates all the vectors of T s and $\perp$ s for the $p(\overleftarrow{\alpha})$ s and $v_{p(\overleftarrow{\alpha})} \rightarrow p(\overleftarrow{\alpha})$ and $v_{p(\overleftarrow{\alpha})} \wedge p(\overleftarrow{\alpha})$ are used to restrict the effect of the substitution only to $p(\overleftarrow{\alpha})$ s such that $v_{p(\overleftarrow{\alpha})}$ is false.

3.3.1 Substitution of Reading Goals

The knowledge states in which the reading goal $[l]$ is known to be achieved are those in which the knowledge held is sufficient to evaluate l in V_0, T_0. In order to do that, the agent needs to be able to determine that l is true, or that it is false. Thus, the appropriate formula over $\{v_{0p(\overleftarrow{\alpha})}, t_{0p(\overleftarrow{\alpha})}, v_{p(\overleftarrow{\alpha})}, t_{p(\overleftarrow{\alpha})} \mid p(\overleftarrow{\alpha}) \in P\}$ is $\gamma_{V_0,T_0} l \vee \gamma_{V_0,T_0}(\neg l)$.

3.3.2 Substitution of Making Goals

The knowledge states in which the making goal $\{l\}$ is known to be achieved are those in which the knowledge held is sufficient to evaluate that l is true in V, T. Thus, the appropriate formula over $\{v_{0p(\overleftarrow{\alpha})}, t_{0p(\overleftarrow{\alpha})}, v_{p(\overleftarrow{\alpha})}, t_{p(\overleftarrow{\alpha})} \mid p(\overleftarrow{\alpha}) \in P\}$ is $\gamma_{V,T} l$.

The principles for the substitution is essentially the same as the one used in *RW* [7], a precursor of *X-Policy*.

3.4 Backwards Reachability Computation

3.4.1 Computing Sets of States

This is a crucial step and show how we deal with compound actions in our reachability analysis. To find strategies the algorithm starts from K_G, searching for sets of the knowledge states which transition into K_G by reading a variable $p(\overleftarrow{\alpha}) \in P$; sets of the knowledge states which transition into $K_{\mathrm{P}G|}$ by executing a write action $act(\overleftarrow{\alpha}) \in Actions^*$; Then for each newly found set, the algorithm continues to find other sets of the knowledge states which transition into the new set through either of the two kinds of transition relations. During this process, if $k_{\mathrm{P}init|}$ is found in a set of knowledge states, the goal is considered as reachable by following the operations represented by the transition relations which connect the set in which $k_{\mathrm{P}init|}$ is found to $K_{\mathrm{P}G|}$. The operations along the path are deemed as the steps of a strategy by the algorithm.

We also use $\gamma_{V,T}(X(act(\overleftarrow{\alpha}), a)) = \mathrm{T}$ and $\gamma_{V,T} r(p(\overleftarrow{\alpha}), a) = \mathrm{T}$ to denote the conditions under which a knows with Knowledge V, T that she is permitted to execute $act(\overleftarrow{\alpha})$ and read $p(\overleftarrow{\alpha})$ respectively. The mapping $X(act(\overleftarrow{\alpha}), a)$ and $r(p(\overleftarrow{\alpha}), a)$ are boolean expressions composed of members of P as defined in the system policy. $X(act(\overleftarrow{\alpha}), a)$ defines the condition under which a is permitted to execute $act(\overleftarrow{\alpha})$. $r(p(\overleftarrow{\alpha}), a)$ defines the condition under which a is permitted to read $p(\overleftarrow{\alpha})$. For instance, to represent a's current knowledge on $X(act(\overleftarrow{\alpha}),a)$, we need to use the knowledge variables in V and T to replace every occurrence of variables in $X(act(\overleftarrow{\alpha}),a)$, because variables in V and T describes C 's knowledge (a and C share the same knowledge) about the current state of the system. The same applies to $r(p(\overleftarrow{\alpha}),a)$.

In order to formally describe the process of solving the reachability problem, we shall first define the concept of pre-sets. For any $a \in C$, $p(\overleftarrow{\alpha}) \in P$ and and $act(\overleftarrow{\alpha}) \in Actions^*$, where a given set of knowledge states Y.

- $Pre|^{\exists,a}_{act(\overleftarrow{\alpha})}(Y)$ is the set of the knowledge states in which a knows she is permitted to execute $act(\overleftarrow{\alpha})$ and which transition into Y by executing the action $act(\overleftarrow{\alpha})$. Its formal definition is: $\{(V_0,T_0,V,T) \mid (V_0,T_0,V \cup \text{effect}^+(act(\overleftarrow{\alpha})) \cup \text{effect}^-(act(\overleftarrow{\alpha}))$, $T \cup \text{effect}^+(act(\overleftarrow{\alpha})) \setminus \text{effect}^-(act(\overleftarrow{\alpha})) \in Y$, $\gamma_{V,T} X(act(\overleftarrow{\alpha}),a) = \mathrm{T}\}$.

- $Pre^{\exists,a}_{p(\overleftarrow{\alpha})=\mathrm{T}}(Y)$ is the set of the knowledge states in which a knows she is permitted to read the value of the variable $p(\overleftarrow{\alpha})$ and which transition into Y by returning the $\{(V_0,T_0,V,T) \mid p(\overleftarrow{\alpha}) \notin V_0, p(\overleftarrow{\alpha}) \notin V, (V_0 \cup \{p(\overleftarrow{\alpha})\}, T_0 \cup \{p(\overleftarrow{\alpha})\}, V \cup \{p(\overleftarrow{\alpha})\}, T \cup \{p(\overleftarrow{\alpha})\}) \in Y, \gamma_{V,T} r(p(\overleftarrow{\alpha}),a) = \mathrm{T}\}$ value of $p(\overleftarrow{\alpha})$ and find out it is true. Its formal definition is:.

- $Pre|^{\exists,a}_{p(\overleftarrow{\alpha})=\perp}(Y)$ is the set of the knowledge states in which a knows she is permitted to read the value of the variable $p(\overleftarrow{\alpha})$ and which transition into Y by returning the value of $p(\overleftarrow{\alpha})$ and find out it is false ($\perp$). Its formal definition is: $\{(V_0,T_0,V,T) \mid p(\overleftarrow{\alpha}) \notin V_0, p(\overleftarrow{\alpha}) \notin V, (V_0 \cup \{p(\overleftarrow{\alpha})\}, T_0 \setminus \{p(\overleftarrow{\alpha})\}, V \cup \{p(\overleftarrow{\alpha})\}, T \setminus \{p(\overleftarrow{\alpha})\}) \in Y, \gamma_{V,T} r(p(\overleftarrow{\alpha}),a) = \mathrm{T}\}$.

3.4.2 Generating Strategies

During the course of the computation, the algorithm maintains 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 C to reach $K_{G|}$ from each state in Y. For $Y = K_{G|}$, the s is simply '$skip;$', which means 'do nothing'.

The algorithm starts with the pair $(K_{G|}, skip;)$. The core of the algorithm works as follows: Given the pair (Y,s), it adds the pairs $(Pre^{\exists,a}_{act(\overleftarrow{\alpha})}(Y),(a : act(\overleftarrow{\alpha});s))$. For any two pairs (Y_1,s_1) and (Y_2,s_2) where $Pre^{\exists,a}_{\mathrm{p}(\overleftarrow{\alpha})=\mathrm{T}}(Y_1) \cap Pre^{\exists,a}_{\mathrm{p}(\overleftarrow{\alpha})=\perp}(Y_2) = Y$, it adds the pair $(Pre|^{\exists,a}_{\mathrm{p}(\overleftarrow{\alpha})=\mathrm{T}}(Y_1) \cap Pre|^{\exists,a}_{\mathrm{p}(\overleftarrow{\alpha})=\perp}(Y_2), \textbf{if } (\mathrm{a} : \mathrm{p}(\overleftarrow{\alpha})) \textbf{ then } s_1 \textbf{ else } s_2)$.

The algorithm continues until no new pairs are generated. Now, all the pairs whose set of knowledge states contains k_{init} contain the strategies we are looking for.

3.4.3 Pseudo-code for the Algorithm

The algorithm for extracting strategies is described below in the form of pseudo-code. It assumes as input the initial state k_{init} and the set of goal knowledge states K_G. It outputs at least a strategy for going from k_{init} to some state in K_G. The algorithm works by backwards reachability from K_G to k_{init}. It maintains a set of states it has seen, called $states\_seen$, and a data structure storing the pairs found so far, called $strategies$. $strategies$:= $\emptyset$; $states\_seen$:= $\emptyset$; put $(K_G, skip;)$ in $strategies$; repeat until $strategies$ does not change choose $(Y_1, s_1) \in strategies$; for each $act(\overleftarrow{\alpha}) \in Actions^*$ for each $a \in C$ PXY := $Pre|^{\exists,a}_{act(\overleftarrow{\alpha})}(Y_1)$; if ((PXY $\neq \emptyset$) $\wedge$ (PXY $\not\subseteq states\_Pseen$)) $states\_seen$:= $states\_seen \cup$ PXY; pts_1 := a : act$(\overleftarrow{\alpha})$ + s_1; $strategies$:= $strategies \cup (\mathrm{PXY}, pts_1)$; if ($k_{init} \in$ PXY) output pts_1;   choose $(Y_2, s_2) \in strategies$; for each variable $\mathrm{p}(\overleftarrow{\alpha}) \in P$ for each $a \in C$ PRY := $Pre^{\exists,a}_{\mathrm{p}(\overleftarrow{\alpha})=\mathrm{T}}(Y_1)$ $\cap$ $Pre^{\exists,a}_{\mathrm{p}(\overleftarrow{\alpha})=\perp}(Y_2)$; if ((PRY $\neq \emptyset$) $\wedge$ (PRY $\not\subseteq states\_seen$)) $states\_seen$:= $states\_seen \cup$ PRY ; pss := "if (a : p$(\overleftarrow{\alpha})$) then $s1$ else $s2$"; $strategies$:= $strategies \cup (\mathrm{PRY}, pss)$; if ($k_{Pinit|} \in$ PRY) output pss;

4. Tool: Case Study - Analysis of *EC* Security Properties

In this Section, we will present three security properties for ***EC*** in details. We have first discovered[6] these issues while using EasyChair . In each case, we show an attack strategy to achieve an undesirable state. These attacks are derived using ***EC*** and *X-Policy* . In the following, we report the results of each attack and make some suggestions on how to fix the system. We describe the *X-Policy* queries that represent these properties. For space restriction, readability and presentation purposes, we describe an optimal strategy that explains the attack for each property as the tool-generated strategies can be complex. These attacks have also succeeded on EasyChair as of 1st of September 2009. For our ***EC*** model, we create the following configuration:

1. The system has five agents: Alice , Bob , Eve , Carol and Marvin . The system has two submitted papers: p1 and p2 .
2. Alice is the Chair of PC. Bob and Carol are PC members. Paper p1 is submitted by the author Marvin while p2 is submitted by the author Eve . The configuration and the attacks' detailed derivation of the model ***EC*** are detailed in [6].

4.1 Property 1: A single subreviewer should not be able to determine the outcome of a paper reviewing process by writing two reviews of the same paper.

This attack against ***EC*** involves 4 agents: Alice , Bob , Carol , and Eve . We explain the attack strategy executed by these agents as follows:

1. Alice acts as chair. She executes the actions: AddReviewerAssignment(p1, Bob) to assign Bob to review the paper p1 . She also executes AddReviewerAssignment(p1, Carol) to assign Carol to review the paper p1 .
2. Bob and Carol both assign Eve as their sub-reviewer for paper p1 by executing the actions RequestReviewing(p1,Bob,Ev) and RequestReviewing (p1 ,Carol , Eve) respectively.
3. Eve accepts the two paper subreviewing requests and sends Bob and Carol two similar reviews using AcceptReviewingRequest(p1 , Carol , Eve) and AcceptReviewingRequest(p1 , Bob , Eve) .
4. Bob and Carol receive Eve 's reviews and submit them to the system using AddReview(p1 , Bob , Eve) and AddReview(p1 , Carol , Eve) .

X-Policy query that represent this property is

```
run for 2 Paper, 5 Agent
check E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
Chair(Alice)! and PCmember(Bob)! and PCmember(Carol)! and Author(p1,Marvin)!
and Author(p2,Eve)! and PCmember-review-editing-enabled()! and View-
submission-by-chair-permitted()! and Chair-review-menu-enabled()! and
PCmember-review-menu-enabled()! and Submission-anonymous()! and Review-
Assignment-enabled()! -> Alice,Carol,Bob:( Submitted-review(p1,Bob,Eve) THEN
Alice,Carol,Bob:Submitted-review(p1,Carol, Eve))
```

EC obviously fails this property and allows Eve to submit two reviews for the same paper. One possible fix for this attack is as follows. Every time an agent a invites another agent b to subreview a paper, EasyChair should check whether agent b has been invited by another agent to subreview the same paper.

4.2 Property 2: A paper author should not review her own paper.

As before, we explain the attack strategy executed by the agents Alice , Bob and Eve :

1. Alice acts as Chair and assigns Bob , who is a PC member, to review the paper p2 submitted by Eve by executing the action AddReviewerAssignment(p2, Bob)
2. Bob executes the action RequestReviewing(p2 , Bob , Eve) to assign Eve as his sub-reviewer as she is a good researcher in the field.
3. Eve accepts the request using AcceptReviewingRequest(p2 , Bob , Eve) .
4. Bob submits the review using AddReview(p2 , Bob , Eve) .

We represent this property in *X-Policy* as the following check statement

```
run for 2 Paper, 5 Agent check E dist p1,p2:Paper ,Alice, Carol, Bob,
Marvin, Eve: Agent || Chair(Alice)! and PCmember(Bob)! and PCmember(Carol)!
and Author(p1,Marvin)! and Author(p2,Eve)! and PCmember-review-editing-
enabled()! and View-submission-by-chair-permitted()! and Chair-review-menu-
enabled()! and PCmember-review-menu-enabled()! and Submission-anonymous()!
and Review-Assignment-enabled()!-> Alice,Carol,Bob: Submitted-
review(p2,Bob,Eve)
```

EC fails the property and allows Eve to review her own paper. Note that the names of the authors and other reviewers are not known to the PC members. One possible fix for this attack is that every time an agent a invites another agent b to subreview a paper, EasyChair should check if b is actually an author of that paper.

4.3 Property 3: Users should be accountable for their actions.

This property is violated in several ways, all of which involve the use of "log in as another pc member" a feature provided for PC chair. For example, the system should not allow the chair to submit a review for a paper as another PC member without making it clear that it is actually the chair who has submitted the review and not the PC member. The following attack scenario involves Alice and Bob :

1. Alice is the chair. She executes AddReviewerAssignment(p1 , Bob) to assign Bob to review the paper p1 .
2. Bob submits his review using AddReview(p1 , Bob , Bob) .
3. Alice reads Bob 's review of paper p1 by executing ShowReview (p1 , Bob , Bob) .
4. Alice submits a review for the paper p1 as if she is Carol who is a very famous and sought after academic by executing AddReview(p1 , Carol , Carol) .

We represent this property in *X-Policy* as the following check statement

```
 run for 2 Paper, 5 Agent
 check E dist p1,p2:Paper ,Alice, Carol, Bob, Marvin, Eve: Agent ||
Chair(Alice)! and PCmember(Bob)! and PCmember(Carol)! and Author(p1,Marvin)!
and Author(p2,Eve)! and PCmember-review-editing-enabled()! and View-
submission-by-chair-permitted()! and Chair-review-menu-enabled()! and
PCmember-review-menu-enabled()! and Submission-anonymous()! and Review-
Assignment-enabled()! -> Alice,Bob: (Submitted-review(p1,Carol,Carol) THEN
Alice,Bob,:Submitted-review(p1,Carol,Carol))
```

Note that Carol in not part of the coalition. EasyChair fails this property and allows the chair to read another reviewer's review for a paper and then submits a review for that paper as another PC member without being detected by the other PC members or the other chairs. This attack is possible because the system does not register the name of the user who updated the review. It will appear to others as if Carol has submitted the review herself. One possible fix for this attack is for *AddReview*() to have an additional parameter. Alice would then need to execute the action AddReview(p , a , b , c) where agent a is the chair acting on behalf of b who is the PCmember submitting the review written by agent c . The predicate Submitted-review( ) also has to be changed accordingly.

5. Evaluation of the method/tool

Table 1. Query Evaluation Time in ms

Query(from [7])	*X-Policy*	DYNPAL	*RW*
Query 4.2	254	120	447
Query 4.3	286	125	782
Query 6.8	360	120	945

In this section we try to compare the performance of *X-Policy* against similar tools, namely a version of *RW* that does not support guessing strategies, and DYNPAL. The queries described in Section 4 cannot be verified by *RW* as it does not support compound actions. However, in order to compare performance, we take 3 queries from [7] that can be handled by both tools and DYNPAL. Note that DYNPAL differs from *RW* and *X-Policy* as it does not consider agent's knowledge in its system representation, search algorithm and state transitions. This table demonstrates that while *X-Policy* tool uses Binary Decision Diagrams (BDD) which are prone to state explosion problems, it still manages to have a reasonable evaluation time. We have implemented *X-Policy* using JavaCC to build the X-policy compiler. The BDD representation and manipulation has been implemented using a Java API called BuDDY. The query evaluation time for RW is different from those in [7, 2] as we are using a version of RW that does not support guessing strategy. For DYNPAL, we use the performance data provided in [2].

6. Related Work

Recently, there has been a plethora of languages and logics to express access control policies [3, 1, 2, 4]. However, we only mention several important related papers in the context of dynamic access control policies. SecPAL [3] and Cassandra [1] and other authorisation languages lack the ability to express the dynamic aspect of access control where policies depend on and update the system state like those we have in $\boldsymbol{EC}$. They, also, cannot express the effect of actions as part of the language and it has to be hard-coded in an ad-hoc way.

DYNPAL [2] aims to specify dynamic policies with the ability to specify the effect of executing these actions. However, DYNPAL's declarative nature, lake of knowledge reasoning and its minimalistic approach make it hard to follow the control flow of the actions. Also the lack of parameter typing does not allow us to establish the relation between the agent who can execute an action and the action itself. DYNPAL focuses on answering the question "under what conditions can an action be executed?" rather than "under what conditions can an agent execute an action?". This is indeed necessary to enable us to define agent coalitions and establish which agent is executing an action. It allows us to detect attacks where we are interested in who can execute a set of actions rather than whether a set of actions can be executed regardless of the actors involved. RW framework [7], a precursor of X-Policy , can perform knowledge-based analysis. However, RW , unlike X-Policy, cannot express actions with multiple assignments needed to preserve the integrity constraints of the modelled system.

7. Conclusion

We present X-Policy , a knowledge-based verification tool to model and verify dynamic access control policies. X-Policy is based on RW but extends it with the ability to express and analyse compound actions. We demonstrate the applicability of X-Policy to real-life web-based collaborative systems. Using X-Policy, we verify three security properties for EasyChair and we discuss the discovered attacks. In future work, we would like to model and verify more such systems. We are also planning to extend the expressiveness of the query language and verification algorithm to express and verify properties like accountability as a system-wide property.

References

[1] M. Y. Becker and P. Sewell, “Cassandra: distributed access control policies with tunable expressiveness”, 5th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY), **(2004)**.

[2] M. Y. Becker, “Specification and analysis of dynamic authorisation policies”, Computer Security Foundations Symposium, IEEE, **(2009)**, pp. 203–217.

[3] M. Becker, C. Fournet and A. Gordon, “Design and semantics of a decentralized authorization language”, In Computer Security Foundations Symposium, 2007, CSF '07, 20th IEEE, **(2007)**, pp. 3–15.

[4] J. DeTreville, "Binder, a logic-based security language", In Proceedings of the 2002 IEEE Symposium on Security and Privacy, **(2002)**.

[5] D. Jackson, "Micromodels of Software: Lightweight Modelling and Analysis with Alloy", Software Design Group, MIT Lab for Computer Science, **(2002)** February, http://alloy.mit.edu/.

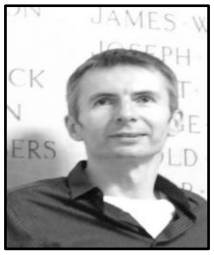
[6] H. Qunoo and M. Ryan, "Modelling dynamic access control policies for web-based collaborative systems", In S. Foresti and S. Jajodia, Eds., DBSec, vol. 6166, Lecture Notes in Computer Science, Springer, **(2010)**, pp. 295–302.

[7] N. Zhang, M. Ryan and D. P. Guelev, "Synthesising verified access control systems through model checking", Journal of Computer Security, **(2005)**.

Authors

Hasan N. Qunoo

I am a researcher in Computer Security. I am a member of Computer Security Group at The University of Birmingham. I do teaching and research. My research is focusing on verifying access control policies via model checking.



Mark D. Ryan

I am Professor in Computer Security and EPSRC Leadership Fellow in the School of Computer Science at the University of Birmingham, U.K.