Towards Modelling and Verifying Dynamic Access Control Policies for Web-based Collaborative Systems

Hasan Qunoo, Masoud Koleini and Mark Ryan

School of Computer Science, University of Birmingham, UK

Abstract. We propose a modelling language and verification tool, called X-Policy, for web-based collaborative systems with dynamic access control policies. The access to resources in these systems depends on the state of the system and its configuration. The X-Policy language expresses systems as a set of programs. Those programs can model system operations which are executed by users. The X-Policy language allows us to specify execution permissions on each program using complex access conditions which can depend on data values, other permissions, and agent roles. We also discuss the challenges to design and implement the verification tool.

1 Introduction

In the recent years, there has been an ever increasing use of web-based systems for managing collaborative work. Systems like social networking websites, conference reviewing systems, document development tools, and application processing systems are all examples of central systems that gives users the ability to create and control access to their data using a set of configuration and a predefined policy. Access to resources in these systems is dynamic; it depends on the state of the system and its configuration.

For instance, large conference management systems like EasyChair[6], iChair[1], HotCRP[4] are widely used to manage academic conferences. However, the size and the complexity of the system policy makes it difficult to analyse its security and correctness properties. Those systems are designed to preserve the system integrity and serve their desired purpose. Systems might not always succeed; users can circumvent the system to gain illegitimate access usually by interactions of rules, co-operations between agents and multi-step actions.

Example: Consider the conference paper review system $\boldsymbol{EC}$. It consists of a set of agents which are program chair(s) or programme-committee (PC) members and a set of papers to be reviewed by the PC members. The following rules apply:

1. PC chair can assign PC members to review a paper.
2. PC members can invite another user to sub-review a paper that is assigned to them. Sub-reviewers may accept or reject the invitation.
3. Sub-reviewers send their reviews (outside the conference management system) to the reviewer.
4. Once the reviewer receives the paper review from the subreviewer, the reviewer can submit the review to the system.

The purpose of these rules is to collect a number (usually between 3 and 4) of reviewer's opinions of a submitted paper. Those opinions determine whether a paper should be accepted or rejected. For these rules to be fair, no single reviewer should be able to determine the outcome of a paper reviewing process by writing all the three reviews of that paper. However, as we can see in the following strategy the intention of these rules can be breached by interaction of rules to allow a single user to write all the three reviews

of the paper. Our analysis of the system only requires one agent to be acting intentionally to circumvent the system.

Strategy:

1. Chair assigns three PC members, Alice, Charlie and Bob, to review the paper p1.
2. Alice assigns Eve as her sub-reviewer.
3. Bob assigns Eve as his sub-reviewer.
4. Charlie assigns Eve as his sub-reviewer.
5. Eve accepts both roles and send Alice, Charlie and Bob three similar reviews.
6. Alice, Charlie and Bob receive Eve's reviews and submit it to the system.

As we can see in this strategy, Eve manages to write all the three paper reviews while all the agents still comply with the system rules. This example shows how hard it is to reason about dynamic policies because it is not enough that each rule is sound by itself. The interaction of rules in the live system can cause an unforeseen behaviour which highlight the need to model these systems and analyse their security properties in a formal and automated way.

In this paper, we propose a modelling and verification framework, called X-Policy, to model and verify large web-based collaborative management systems. X-Policy framework should be able to model these systems. The X-Policy modelling language allows users to specify systems as a set of atomic programs which can change one or multiple variables at the same time. Program execution permissions are specified as preconditions which the user has to satisfy to execute the program. The program execution update the system state.

We have used X-Policy to model an existing conference management systems, EasyChair, as a case study for our framework. X-Policy is expressive enough to model such a system operations and we report on the result of our security properties analysis.

This paper is structured as following: related work goes in section 2. We discuss our X-Policy modelling language in section 3. We discuss the model checking issues and challenges in section 4. Conclusion and future work is in section 5.

2 Related Work

Mark-up Languages like eXtensible Access Control Mark-up Language (XACML)[5] has been standardised to express access control policies. Although XACML can express access control policies, the correctness of these policies are remained unproven. Model Checking based verification frameworks like RW[7,8,9] and Margrave[3] answer this question. They have the advantages of model checking approach which has the ability to:

- build a model M, based on the policy, which enables us to understand the policy as a whole. Any change made to a single rule or caused by adding/deleting a rule will be reflected in M. This makes the study of interactions of rules easier.
- explore possible consequences of multi-step actions by performing temporal reasoning.

RW and Margrave have proved successful on small sized examples. However, there is still a long way to producing a *scalable* and *expressive* access control policy *verification* framework for large collaboration systems which is guaranteed to match the high-level policies which is the aim of X-Policy.

3 *X*-Policy framework

3.1 Modelling and Verifying EasyChair Conference Management System using *X*-Policy:

EasyChair is a web-based centralised conference management system. The system is implemented as a database back-end with a web based front end. Users are presented with a web page which contains links. Clicking on these links executes a program, with certain parameters. The program execution may result in an update of the database and will result in a new page being displayed.

We have built a model ***EC***, which is based on our understanding of a fragment of EasyChair within a single conference system which we model using *X*-Policy below. We specify system operations as *X*-Policy programs which can be either *write programs* that change the state of the system or *read programs* that give the user/agent the knowledge about the state of the system. Programs in *X*-Policy can not read and change the state of the system at the same time. Although this is formally a restriction, most actions in collaborative web-based systems are indeed either a read or write and rarely both. This is true for EasyChair in particular. We believe that this is a sensible heuristic for modelling web-based systems. Users are only enabled to perform only one operation per time.
A *read program* allows the user to know the value of a ground proposition by returning the value of that proposition to the user who executed the program. A *write program* allows the user to assert the value of a set of ground propositions using assignment statements in the form $p(\overrightarrow{y}) := \top$; or $p(\overrightarrow{y}) := \bot$; where $p(\overrightarrow{y})$ is a ground proposition. We allow a proposition to occur at most once at the left of ":=". The assignment statements within the same program can be written in any order. Such an assumption result in making the programs effect independent from the state of the system and each program has the same effect at all the time. A program permission statement exec(g,u) defines the conditions for an agent u to execute a program g. These conditions are defined as propositional logic formulae using the ground propositions and logical connectors.

In the following, we encode the example in the introduction which is a fragment of the ***EC*** policy in *X*-Policy:

```
Program AddReview(p:Paper,a:Agent,b:Agent):-
  {
  Submitted-review(p,a,b) :=T;
  }

exec(AddReview (p:Paper,a:Agent,b:Agent), u): ((Chair (u) or PCmember(u))
                              and Reviewer(p,u))

Program AddReviewerAssignment(p,a):-
  {
  reviewer(p,a):=T;
  }

exec (AddReviewerAssignment(p:Paper,a:Agent) ,u) : (Chair(u)
                              and (PCmember(a) or Chair(a)));

Program RequestReviewing(p:Paper,a:Agent,b:Agent):-
  {
  Requested-subreviewing(p,a,b):= T;
  }

exec(RequestReviewing(p:Paper,a:Agent,b:Agent),u) : (Chair(u)
                              or (PCmember(u) and Reviewer(p,u)
```

```
                    or (PCmember(u) and not Reviewer(p,u))));

Program AcceptReviewingRequest(p:Paper,a:Agent,b:Agent):-
{
  Decided-subreviewing(p,a,b):=T;
  Subreviewer(p,a,b):=T;
}

exec(AcceptReviewingRequest(p:Paper,a:Agent,b:Agent),u) : (Requested-subreviewing(p,a,b)
                    and (not Decided-subreviewing(p,a,u) or u = a);
```

Each write program will transfer the model from a pre-execution state m_i in which the program is executed at to a post-execution state m_{i+1}. Executing a write program will update the model state. It updates the values of the set of ground propositions which are changed by the program to true or false from the state m_i. All the other ground propositions in the state m_i will remain unchanged. Read programs do not change the state of the system. However, we model it to represent the process of reading a sensitive data which can be part of an attack strategy.

4 Model Checking: Abstraction and Refinement

Once the model is defined, we define a query which consist of a coalition of users, an initial condition and a goal condition. X-Policy model-checking algorithm searches for a strategy which is a sequence of programs that the users in the coalition can execute to drive the model from the initial state to the goal state. The verification method we use is based on backward reachability algorithm. Model-checking begins from the goal states. It performs a backwards search by finding pre-states based on programs permissions and their effects. The algorithm analyses the found pre-states and checks whether specification is satisfied. This process will continue until it reaches the initial states or no more states are found.

However, model checkers suffer from the state explosion problem when the number of propositions grows in the model. The number of states in the system grows exponentially as the number of propositions in the model increases. A number of techniques are used to help model-checkers to handle large systems with huge number of variables. Abstraction is one of the techniques which reduces the number of states in the abstracted model compared to the original model. This enables the abstracted model to be analysed by regular model-checkers. CEGAR[2] is a framework for counter-examples guided abstraction and refinement. The CEGAR abstraction methodology consists of three steps: generate the initial abstraction, model check the abstract structure and refine the abstraction. We propose to use CEGAR in our model checking technique.

4.1 Model Abstraction and Refinement

Our proposed abstraction technique is based on variable hiding methods by existentially removing propositions from the concrete model to construct the abstract model. The initial abstraction begins with removing all the propositions from the concrete model except those that were contained in the specification formula ϕ where ϕ is an ACTL* formula. If a specification ϕ is not satisfied in a model M, then ϕ is an invariant of M. Let $prop(\phi)$ be the set of propositions in the formula ϕ, the number of propositions in the initial abstract model is $|prop(\phi)|$.
As many abstraction methods, our method satisfies the property: If the abstracted model M' is constructed from the concrete model M, then the following property holds:

$$M' \models \phi \rightarrow M \models \phi$$

where ϕ is an ACTL* specification formula. We verify whether the goal specification is satisfied in the model. If the specification is satisfiable in the abstract model, the model checker will output a strategy. We check whether the found strategy can exist in the concrete model. We apply the programs in the strategy sequentially starting from the initial state in the concrete model. This process determines which state in the concrete model is the failure state. It also determines which transition is a spurious transition. If the strategy is spurious, we refine the abstract model to make the abstract model more precise.

The refinement process puts back some of the propositions that were removed from the model in abstraction process. Additions of new propositions will split the failure state. The failure state should be split in such a way that the spurious transition can not happen in the refined model.

We propose a proposition ranking method based on the analysis of the execution permissions. All propositions start with the rank one. Every time a proposition appears in a program execution permission statement, we increase the proposition rank. We apply this to all the permission statements. At each refinement step we add a proposition from the set of propositions with the highest rank. We then model check the refined model against the specification. If the model checker outputs another strategy, we apply the refinement process again. We recursively apply this process until the model checker yell no strategy or a valid strategy in the concert model. Our refinement process has the maximum number of refinement steps as $|P| - |prop(\phi)|$ where P is the set of the concrete model propositions.

5 Conclusion and Future Work

In this paper we present a modelling language, $\mathcal{X}$-Policy framework, to model the dynamic execution policy of large web-based collaborative systems. We demonstrate the applicability of $\mathcal{X}$-Policy to model real life web based collaborative systems like EasyChair. We discuss the challenges to solve the state explosion problems in our model checking method.
In future work, we are planning to adapt the model checking algorithm for $\mathcal{X}$-Policy model. We are also working to develop the abstraction and refinement technique as discussed in section 4. We also plan to develop a tool which implements our framework.

References

1. Thomas Baignères and Matthieu Finiasz. iChair conference management system. `http://lasecwww.epfl.ch/iChair//`.
2. Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50:752–794, Sep 2003.
3. Michael Matthew Greenberg, Casey Marks, Leo Alexander Meyerovich, and Michael Carl Tschantz. The soundness and completeness of margrave with respect to a subset of xacml. Technical Report CS-05-05, Computer Science Department, Brown University, April 2005.
4. Eddie Kohler. Hotcrp conference management software. `http://www.cs.ucla.edu/~kohler/hotcrp/index.html`.
5. Sun Microsystems. Sun's XACML implementation, Aug 2003. Information about this implementation can be found at `http://sunxacml.sourceforge.net/`.
6. Andrei Voronkov. EasyChair conference system. `http://www.easychair.org//`.
7. Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems in XACML. In *2004 ACM Workshop on Formal Methods in Security Engineering*, pages 56–65, Washington DC, USA, Oct 2004. ACM Press.
8. Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems through model checking. *Journal of Computer Security (in print)*, 2005.
9. Nan Zhang, Mark D. Ryan, and Dimitar P. Guelev. Evaluating access control policies through model-checking. In *8th Information Security Conference (ISC'05)*, Singapore, Sep 2005. Springer-Verlag.