Access Control Policy Verification and Repairing in Alloy

Alexandr Murashkin, Ming Matthew Ma
The David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada
amurashk, m22ma@uwaterloo.ca

ABSTRACT

Databases can store important information that may be accessible in non-intended way. Therefore, data access control becomes significant [3]. To express access control policies, several languages such as XACML [9] are used to specify whether subjects are allowed to access sets of resources or services to perform specific actions on them.

In this paper, we address the problem of access control policy model verification and repairing. We develop a tool based on first-order logic modeling to detect and visualize possible conflicts within sets of access control policies expressed in XACML. We first translate the model into a relational first order logic language called Alloy [6], and then analyze interactions and conflicts among access control policies using Alloy Analyzer. We then propose potential repairs to the user through the user interface, and automatically apply the fixes selected by the user. We have shown that our tool can automatically determine inconsistencies in user specified model, prioritize possible fixes and successfully apply the selected repairs to the model.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Graphical user interfaces (GUI)*

General Terms

Computer Aided Verification, Access Control Policy, Verification, Model Repair

Keywords

access control policy, Alloy, XACML, GUI

1. INTRODUCTION AND MOTIVATION

Although XACML has already achieved a considerable degree of industrial acceptance [8], managing access control policies is still error-prone and complicated [4]. "Nowadays data is increasingly available on-line through the web and other distributed protocols. This heightens the need to carefully control data access" [3]. For example, students should be able to view their marks, but they are not allowed to modify their marks. This is responsibility of professors, and they have to be able to modify student marks as well as view them. All such rules are defined within access control policies.

However, "access control is not means not only preventing the leakage of data but also permitting access to necessary information" [3]. Therefore, faults in access control policies that allow students to modify their grades or restrict the access of professors to doing that are both dangerous.

The larger the system, the more complicated access control is required. Moreover, in big systems, access control policies may be defined more than once, they are being refined by different people several times. Therefore, it is hard to keep track of all consequences [3]. This, in turn, makes the error localization hard.

Even if error is localized, there can be many ways to repair access control policies. Manual repairing with consecutive change-impact analysis is definitely complicated. We will justify this problem in the following section and show in this paper that such inconsistencies can be detected and fixed automatically using our tool.

1.1 Problem Statement

The process of verification and repairing of access control policies is time-consuming, and development of verification tools like Margrave [3] is a good evidence. Once inconsistencies are found, the user has to think on how to repair policies, but there is no guarantee that the fixes can indeed repair the model without introducing new inconsistencies. Thus, the entire model construction process, model verification process, finding inconsistencies and repairing process can be overwhelming.

In this paper, we propose a tool we developed that can automatically determine inconsistencies (or faults), recommend the repair to the users and automatically apply fixes to the access control model.

1.2 Paper structure

In the *Background* section we provide relevant information on model verification and repairing, XACML and Alloy.

All related work we considered is listed in *Related work* section.

Theoretical approaches section will introduce verification and model repairing approaches we used for our tool and present its components and the implementation at different stages.

In the *Implementation* section we will present the components of our tool and its implementation at different stages. *Evaluation* section contains our toy experiment results. In *Future work* section, we discuss our tool limitations and bring up possible future improvements. Conclusion section summarizes completed work and achieved results.

2. BACKGROUND

In this section, we introduce model verification and model repairing, XACML standard for defining access control policies and Alloy language that we used for verification in our work.

2.1 Model verification and model repairing

Given a model M, some property φ and satisfaction relation $\models$, verification is the process of determining whether $M \models \varphi$. If $M \not\models \varphi$, the model repairing is the process of finding some M', such that $M' \models \varphi$ and the derivation of M' from M' is minimal. In this paper, we will try to verify and repair models that represent access control policies. Our approach is not limited to one property, but currently we can successfully repair the models that violate the following abstract property similar to the third property in [8]:

Property 1: *Within a policy set that has OnlyOneApplicable policy combining algorithm, there is no two policies that for a given request return different decisions.*

2.2 XACML access control policies

XACML is OASIS standard and stands for eXtensible Access Control Markup Language [9]. It defines structure, interaction and conventions related to policy sets, policies, rules and request within an access control system. In our work we use a simplified subset of XACML used in [8]. In our case, policy set is a set of policies composed using policy combining algorithm, which can be *PermitOverrides*, *DenyOverrides* or *OnlyOneApplicable*[1]. Each policy contains the set of rules and rule combining algorithm which can be *PermitOverrides* and *DenyOverrides*. Each request contains one subject, one action and one resource (Fig. 1), and the system calculates the response of applicable rules, policies, and can make desicions based on that.

A decision can be *Permit*, *Deny* or *NotApplicable*, the latter says that the request is not handled by certain rule, policy or policy set.

Usually XACML access control policies are stored in XML files. Those XML-files are complex and have a lot of redundant information, so for our example we can use tabular form of defining policy sets (it is not commonly used, but it is more compact and straight-forward in out case), see Fig.2.

[1]we do not consider *FirstApplicable* since it will make our model more complicated. We also omit dashes and use capitalization for consistency with our model

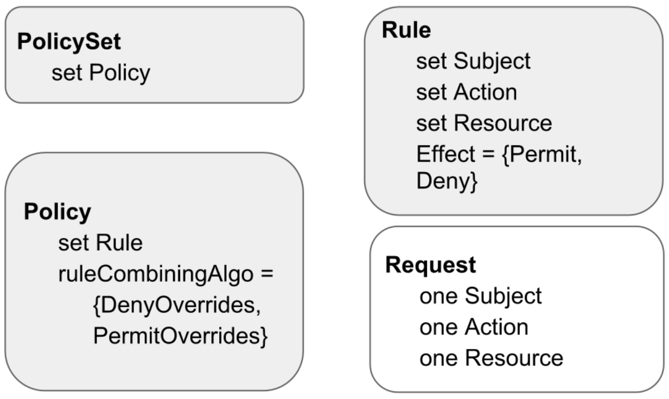


Figure 1: Shortened representation of some XACML entities: policy set, policy, rule, and request. Alloy-friendly notation

Policy P1	Policy P3
Rules: 1) Student ⇒ Read ⇒ Marks ⇒ *Permit* 2) Professor ⇒ Read | Modify ⇒ Marks ⇒ *Permit*	**Rules:** 1) Professor ⇒ Read | Modify ⇒ Marks ⇒ *Deny* 2) Professor ⇒ Modify ⇒ Marks ⇒ *Permit* 3) Professor ⇒ Read ⇒ Marks ⇒ *Permit*
RuleCombiningAlgorithm: DenyOverrides	**RuleCombiningAlgorithm:** DenyOverrides

Figure 2: Policy Set PS contains two policies: P1 and P3, each has a set of rules and rule combining algorithm

So, in our example a policy set PS contains two policies: $P1$ and $P3$, the first rule of $P1$ is $Student \rightarrow Read \rightarrow Marks \rightarrow Permit$ is applied when the subject has the role $Student$, the resource is the $Marks$ file and the action is $Read$, and this access is granted ($Permit$). In policy $P3$, the first rule denies both reading and modifying marks for professor. The policy has $DenyOverrides$ rule combining algorithm which says that this rule is dominant over others in case that more than one rule is applicable for some relevant request. The policy response is equal to the one rule which is applicable or by the dominant rule if multiple rules are applicable. The same is true for policy set response, but it has $OnlyOneApplicable$ policy combining algorithm as well, which says that the policy set response in indeterministic if more than one policy is applicable in the context of a given request.

2.3 Alloy

Alloy [6] is a modeling language based on relational first-order logic. We model concepts in Alloy using so called signatures and relations. An abstract signature operates as a type or class definition. For example, we can define a policy as follows (as in [8]):

```
abstract sig Policy {
  policyTarget : one Target,
  rules : set Rule,
  combiningAlgo : one RuleCombiningAlgo
}
```

The keyword *extends* can operate like instantiation of a class, and we can define atom (an atomic signature) that "extends" another signature (actually, the former becomes an atomic element of the set of the latter [6]). So, Policy1 becomes an instance of Policy:

```
one sig Policy1 extends Policy {}{
  policyTarget = T0
  rules = Policy1_Rule_Student_Read_Marks_Permit
    + Policy1_Rule_Professor_ReadModify_Marks_Permit
  combiningAlgo = DenyOverrides
}
```

A relation is a structure that relates atoms [6]. In our example, *rules* is relation that relates each policy to a set of rules. Constraints are typically represented in Alloy as global or local facts. A fact is a logical statement that is assumed to always hold. In our example above, "assignments" of $policyTarget$, *rules* and *combiningAlgo* are indeed local facts.

We can also specify functions that take in atoms and return other atoms. In our case, we define such functions to calculate policy response with overridden rules. Alloy also supports predicates that return $true$ or $false$ depending on an input parameters.

Alloy Analyzer is a tool that takes the model in Alloy language and finds variables that satisfy them. It can be used both for model exploration and verification. The way it works in our case we describe in section 4.1.

3. RELATED WORK

Regarding access control policy verification, there are several various related approaches. [5] proposed encoding XACML access control policies ordering relations and then translation of them into SAT solver for verification. This paper defined some Alloy notation as well. The papers [4] and [8] defined XACML access control policies notation in Alloy and deal with verification problem. The latter approach - [8] - is used in our paper, it introduces the usage of predicates for access control policy model verification and validation. We found the way the models are defined in this paper straightforward, so we used this approach and extended it. Other than that, [2] describes inconsistency checking in role-based access control policies (RBAC). [3] proposes Margrave - a tool for access control policies (XACML and other formats) verification and change-impact analysis. [7] used description logics to formalize and verify access control policies. The authors also specified semi-automated access control policy repair as their future work.

None of the paper above solves the problem of access control policy repairing. It does not seem to be published prior related work in this topic[2]. Rather than that, [11] offers access control policies verification in the language called RW and then synthesize verified specifications in XACML. [1] considers another type of access control policies - XML write access control policies - offers repairs in case of that some actions can be simulated by multiple another actions. Regarding repair in general and different domains, [10] proposes repair trees for inconsistency solving in design models. We use similar trees in our repairing approaches. There are works on model repairing in temporal logic, but we do not consider the notion of time in our work. Semi-automated repairing of XACML access control policies is the most important contribution of this paper.

[2]We do not consider temporal logic

4. THEORETICAL APPROACHES

In this section, we present the verification and repair approaches we use in our work.

4.1 Verification approach

The verification procedure is based on the one in the technical report [8]. We use the same idea in our paper and make it more general. The first step is to define the Alloy input file.

1. First, a meta-model (or abstract model) is defined at the beginning of the file. This meta-model is domain-specific. It consists of signatures ($Policy$, $PolicySet$, $Rule$, $Effect$, $RuleCombiningAlgorithm$, etc.) and relations ($rules$ relation links a policy and rules it contains).

2. Next, a concrete model is defined. It consists of signatures ($Policy1$, $Policy2$, $Rule1$, $Rule2$) and facts ($Policy1$ contains $Rule1$ and $Rule2$) regarding the relations specified in meta-model. Concrete model can vary within the given domain, so one meta-model can be used for many concrete models within the domain.

3. And finally, a property predicate is defined. This predicate is domain-specific as well, and each predicate represents the negation of the property we need to verify. In this paper, we focused on an abstract property. It is stated as follows:

Property 1: *Within a policy set that has OnlyOneApplicable policy combining algorithm, there is no two policies that for a given request return different decisions.*

The full body of the predicate is given in the Appendix, and its signature looks as follows:

```
pred InconsistentPolicySet [ps : PolicySet,
    req : Request,  p1: Policy, p2: Policy,
    r1: Rule, r2: Rule]
```

The predicate is supposed to find a policy set ps that has *OnlyOneApplicable* policy combining algorithm and contains policies $p1$ and $p2$, such that policy $p1$ contains rule $r1$, policy $p2$ contains rule $r2$. At the same time, $r1$ defines the response of policy $p1$: the rule $r1$ is either the only applicable rule in context of the given request req, or dominates all other rules of the policy $p1$ after apllying the rule combining algorithm. And similarly, $r2$ defines the response of policy $p2$.

And the second step is to send the Alloy file to Alloy Analyzer and execute the property predicate. We will denote the part of Alloy file without predicate as a model (so, the model consists of two parts: abstract model and concrete model). If the model is consistent, then the predicate is inconsistent with respect to the model, and Alloy Analyzer cannot generate an instance (example). If the predicate is consistent, it means that the model is inconsistent with respect to the property (the property is not satisfied in the model), and a counterexample is returned.

This verification approach gives certain advantages. We can localize the inconsistency, since Alloy Analyzer will show the values of all the arguments of the predicate in the generated instance, including the request req, even if there is no definition of request. It is not the case if we make the predicate denote the property itself, not its negation: in this case, if the property is not satisfied, then $UNSAT$ cores can be shown, but it takes computation time to minimize them, and they still might not be minimal. So, for error localization the taken approach is better.

4.2 Repair approach

Once there is an inconsistent policy set, the verification procedure will return the policy set ps, policies $p1$, $p2$ and the corresponding rules $r1$ and $r2$. Our repair procedure is applied to policies or rules, and this will affect the consistency of the policy set ps. Since our verification procedure returns two rules of two policies, actually we can try to apply the same set of repairs to each of them by simmetry. Therefore, we can consider the following simple repair ways in the context of the returned policy set ps, and one policy $p1$ and rule $r1$ within it.

1. Switch the effect of the rule $r1$. If it was $Permit$, it is changed to $Deny$, and conversely.

2. Switch the policy combining algorithm of the policy $p1$. If it was $PermitOverrides$, it becomes $DenyOverrides$, and conversely.

3. Remove the rule $p1$ from the policy $p1$.

4. Remove the policy $p1$ from the policy set ps.

5. Switch the policy combining algorithm of the policy set ps from $OnlyOneApplicable$ to either $DenyOverrides$ or $PermitOverrides$.

The ways 1 and 2 seems to be the best. Just switching the values is definitely less radical. Ways 3 and 4 are dangerous: first, they might affect other requests; next, they can bring us to the empty model which satisfies everything. Way 5 is quite global to apply it, and again, it will hide the conflicts at all, and this can cause more problems in the future. There can be other ways, though: changing the subject, action and resource, but this is risky and can increase the number of states.

So, for this project, we end up with the first two repair ways and we can apply them to both $p1$ and $p2$, so actually we have 4 repair ways:

1. Switch the effect of the rule $r1$.

2. Switch the policy combining algorithm of the policy $p1$.

3. Switch the effect of the rule $r2$.

4. Switch the policy combining algorithm of the policy $p2$.

However, the application of one repair way might not be enough, in case that the core of inconsistency is not represented by two rules only. First, there might be redundant rules. For example, if the rule $Student \rightarrow Read|Modify \rightarrow Marks \rightarrow Permit$ is present, then the rule $Student \rightarrow Read \rightarrow Marks \rightarrow Permit$ is redundant, and changing the rule effect of the latter will cause the application of the former rule and this will not solve the problem in one step. This case is handled during the optimization during the conversion of the source file to Alloy file. Next, there are cases that a policy set is faulty for more than one request (not only for the request $Professor \rightarrow Read \rightarrow Marks$, but with the request $Professor \rightarrow Modify \rightarrow Marks$, for instance), and it definitely needs more than one repair procedure in our approach. This is why we propose the displaying of the approximate numbers of next fixes.

The approximate number of next fixes is calculated as follows. After repairs have been proposed, the system tries to apply each of them gaining partially repaired models. Then, it runs the verification again for each of the partially repaired models. If some partially repaired model is consistent, it means that this model is fully repaired. So in this case, the approximate number of next fixes is zero. If some partially model is still inconsistent, then the system tries to apply fixes to the partially repaired model again, and adds 1 to the number of next fixes. This process can run infinitely and cause state explosion problem, so we limit the depth to certain amount (in our case we specified 2).

If the number of next fixes for each fix is greater than zero, then user needs to apply fixes multiple times. The user will go down the repair tree until he will get the fully repaired model (Fig.4).

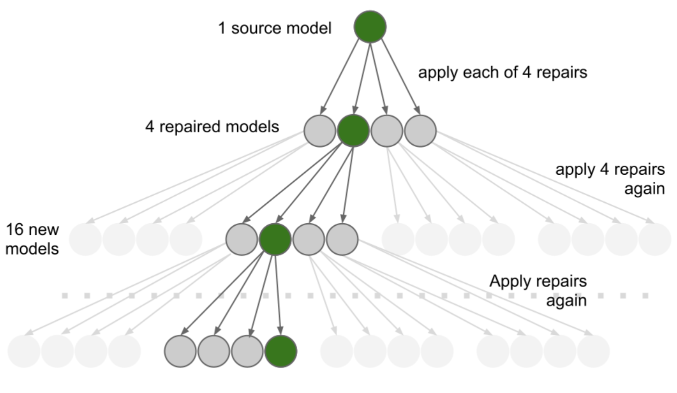
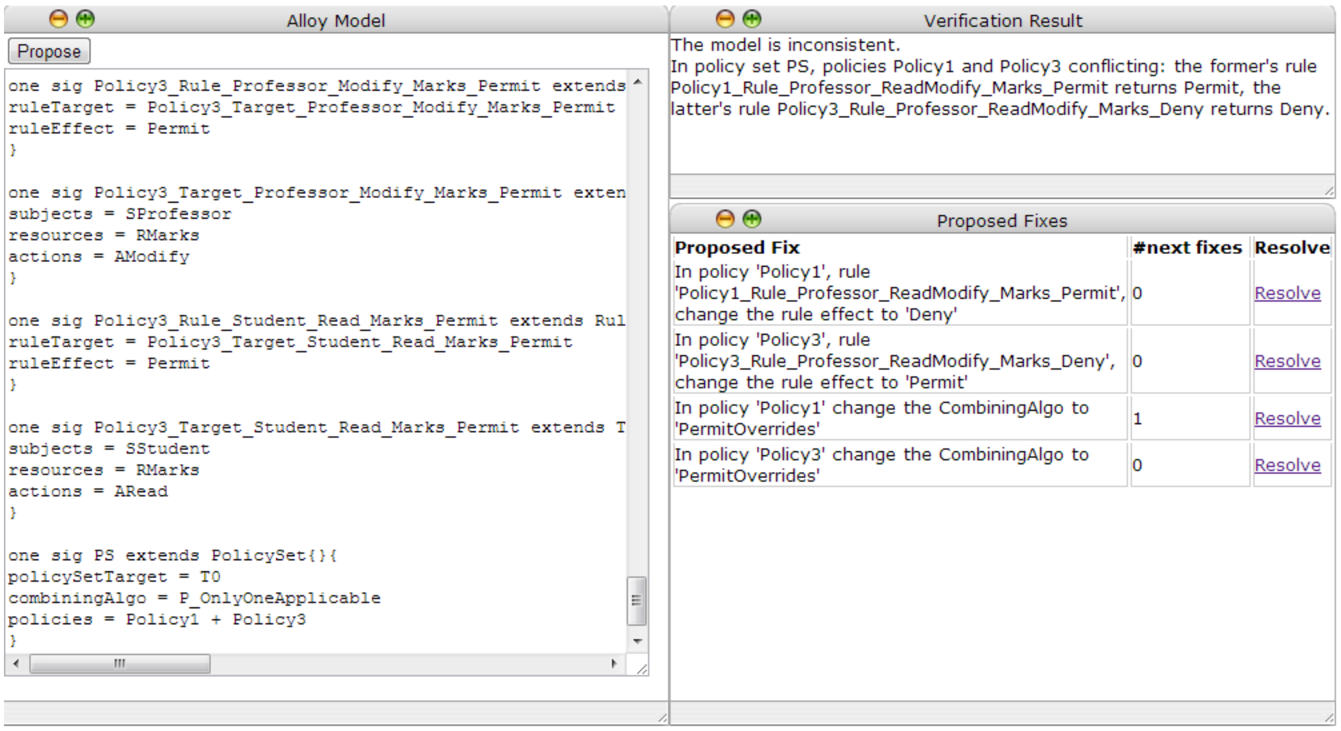


Figure 4: Repair trees. Green nodes are decisions of the user

In this approach, more than one path can produce the repaired model. It does not matter whether this is a minimal one or not: if the user chooses the path with minimal number of next fixes, then eventually he will get the repaired model with minimal number of changes.



```
one sig Policy3_Rule_Professor_Modify_Marks_Permit extends
ruleTarget = Policy3_Target_Professor_Modify_Marks_Permit
ruleEffect = Permit
}

one sig Policy3_Target_Professor_Modify_Marks_Permit exten
subjects = SProfessor
resources = RMarks
actions = AModify
}

one sig Policy3_Rule_Student_Read_Marks_Permit extends Rul
ruleTarget = Policy3_Target_Student_Read_Marks_Permit
ruleEffect = Permit
}

one sig Policy3_Target_Student_Read_Marks_Permit extends T
subjects = SStudent
resources = RMarks
actions = ARead
}

one sig PS extends PolicySet{}{
policySetTarget = T0
combiningAlgo = P_OnlyOneApplicable
policies = Policy1 + Policy3
}
```

The model is inconsistent.
In policy set PS, policies Policy1 and Policy3 conflicting: the former's rule Policy1_Rule_Professor_ReadModify_Marks_Permit returns Permit, the latter's rule Policy3_Rule_Professor_ReadModify_Marks_Deny returns Deny.

Proposed Fix	#next fixes	Resolve
In policy 'Policy1', rule 'Policy1_Rule_Professor_ReadModify_Marks_Permit', change the rule effect to 'Deny'	0	Resolve
In policy 'Policy3', rule 'Policy3_Rule_Professor_ReadModify_Marks_Deny', change the rule effect to 'Permit'	0	Resolve
In policy 'Policy1' change the CombiningAlgo to 'PermitOverrides'	1	Resolve
In policy 'Policy3' change the CombiningAlgo to 'PermitOverrides'	0	Resolve

Figure 3: Our tool: the User Interface

However, there is a problem with this approach - cycles. For example, switching rule combining algorithm may not be helpful. So the system will propose to switch rule combining algorithm and then, at the next step, switch it back. However, our approach helps to solve this problem: if cycles are present, then the way that leads to cycles will be annotated with bigger number of next fixes automatically, so it is less likely that the user will want to choose this way.

5. IMPLEMENTATION

The function of our tool is to automatically determine inconsistencies within access control policy defined by XACML, and recommend the potential repair to the user, and finally apply the user chosen fixes automatically. Throughout this process, user only have to review the recommended fixes, and choose the fix by clicking a button in user interface, and then the model should be fixed automatically by our tool (Fig. 3).

To increase the usability of our tool, we have implemented external module, a converter that can automatically generate Alloy model based on a table template (or xml, csv file template) so that the user does not need to have complete literacy in Alloy modeling language. As the content of the such a template, the user is asked to specify the rule, subject, action, resource and effect in predefined format. Our converter will than generate the Alloy model and its predicate to verify inconsistencies. Once Alloy model is generated, we input the model into our inconsistency detection and repair tool.

5.1 Architecture and data flow

Our tool consists of three major parts: the User Interface (UI), PropertyVerifier&Fixer and Alloy Analyzer. The architecture is shown in the figure below (Fig.5).

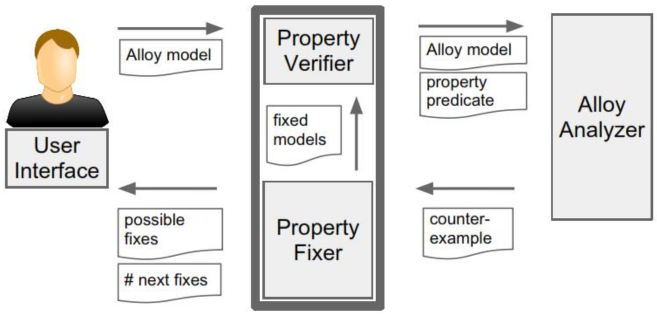


Figure 5: Tool Architecture and Data Flow

At the beginning, the UI first takes in the Alloy model created, and sends it to PropertyVerifier&Fixer. Next, the PropertyVerifier part starts Alloy analyzer to process Alloy model and its property predicate. It should be noted that we write our predicate in a way that it checks for if two policies return inconsistent results. PropertyFixer then takes in the results from Alloy Analyzer and extracts relevant information, such as inconsistent rules, subject, action, resource and effect, and process those information to propose the potential fixes. The fix recommendation from PropertyFixer is passed to the UI, and the user is shown with a list of fixes. Once the user chooses one particular fix, our tool will automatically update the original Alloy model. This updated model is then passed to PropertyVerifier to check if there are any inconsistencies. This process is repeated until the model is verified to be consistent.

In order to realize the above mentioned architecture, we have created several Java modules which are summarized below.

5.1.1 InteractiveRepairer

This is the core of our tool, it contains:

- Main engine
- PropertyVerifier
- PropertyFixer

The Main engine controls the communication between the user interface and our Fixer and Verifier; it process input from UI and outputs the results to UI for user inspection. PropertyVerifier calls Alloy Analyzer through our API and obtains XML representation of the generated instance that contains information about the policies, rules, request and effects. PropertyVerifier then follows XPATH standard to extract and store that information from XML for later processing. Once the XML nodes are extracted with relevant information from Alloy Analyzer, PropertyFixer takes in those information and proposes potential fixes. Once the user selects the preferred fix, this decision is communicated to PropertyFixer again through Main engine, and PropertyFixer will automatically apply the fixes to the original Alloy model.

Note that PropertyFixer and PropertyVerifier function as a pair, and one pair is required for each property to be verified; in our study, we limit the property verification to the safety property but this is extensible by creating another pair of PropertyFixer and Verifier.

5.1.2 AlloyRunner

AlloyRunner is an API that facilitate the communication between our tool with Alloy analyzer. Mainly, it initializes the Alloy Analyzer, sends model to Alloy analyzer, and get XML instances from Alloy Analyzer. Alloy Analyzer serves as verification engine for our input model and will return instances or counter example which we use in later processing.

5.1.3 Presentation Layer

Presentation uses dynamic web application technologies such as Javascript and Node.js to allow interactive model repair. Node.js executes interactive repair with user inputs and calls back to the Main engine in InteractiveRepairer to apply the repair. For instance, the Propose button triggers Main engine to start PropertyVerifier to execute the model verification, and the Resolve links will cause the PropertyFixer to apply the repair on the original Alloy model. The return values are associated with each function call-back of the button, and represents error location and proposed fixes for Propose command and new model for Resolve command. All proposed fixes are shown and annotated by the approximated number of next fixes.

6. EVALUATION

Despite XACML is well-adopted standard, we have not evaluated our tool on industry models given the time constrains. By now, for our dataset, we have defined 6 toy examples in tabular format. We intentionally made 5 inconsistent examples of policy sets and a consistent one. Policy sets contained from three to four policies with three to six rules each. The tool has successfully identified inconsistent policy sets and proposed the ways of fixing them. The priority given to each fix was indeed the right priority, and the number of next fixes matched our own manual solution. Conducting a proper study with the industry examples is a part of the future work.

7. FUTURE WORK

Although our tool demonstrate potential value in automated access control model repair, while significantly reducing user's manual work, we see future improvements in our tool. We noticed the following limitations in our tool:

1. Verification is bounded
2. Can verify one property in a time until first counterexample is found
3. Subset of XACML is covered
4. Repair procedure depends on the property and requires pre-defined prospective repair ways

The first two points are pretty clear. Since we use Alloy as it is for bounded model checking, we can guarantee that the tool will give correct results within the given scope. Although we significantly dependent on Alloy, we mainly interfaced the communication with Alloy API, so in the future, we need to communicate with Alloy back-end to prevent this problem.

Although we strive to make our tool scalable, the automated Alloy model generation from a user specified table brings limitations in the structure of the model we generate. Although this access control model works well for our present study, it only represents a subset of XACML. In the future, we would like to explore different ways to increase the flexibility of our model. One potential way is to incorporate sketching techniques and having multiple rule tables so that the user can choose the structure of the model our tool generates.

Even if the state explosion problem was handled, it is not addressed in the smartest way due to time constraints. By now, we use repair trees and limit the depth of the potential solution to two. This might still bring problems if the model is large and the number of fixes is big. We would like to study more on ways to deal better with this common problem in verification and repair.

One important future improvement is on the repair recommendation. Currently, we only propose repairs based on rule effect and combining algorithm. We derived these approach from observation and testing. We will explore other factors such as relation, and derivation in the future to enhance our repair framework.

And finally, we will conduct more effective evaluation of our tool with industry examples.

8. CONCLUSION

In this report, we presented our tool which can automatically determine inconsistencies in XACML access control policies, recommend possible fixes to the user and automatically apply the repair on the model. We reduced the manual work, and the tool does not require user knowledge of Alloy as a modeling and verification tool.

We have improved the verification approach and meta-model presented in [8] and proposed repair approach related to this verification approach.

We have implemented the core part of the tool that takes in the Alloy model and runs Alloy Analyzer in the backend, and retrieves the information on generated inconsistent instances to propose potential fixes. Among the fixes, it also calculates the approximate number of next fixes required for the selected fix. We have designed a user interface of the tool that allows user to input the model or action and displays verification and repair results.

From evaluation results, we have shown the feasibility of our approach and that our tool can help the user to repair inconsistent access control policy model effectively, requiring only user input for fix selection as a manual part, the rest actions are automated.

And finally, we have created the input converter so that user can convert policy from other formats.

8.1 Implications

First, we have concluded that semi-automated model repair is possible for access control policy defined by XACML structure. We have made implications on the fact that user input is still necessary to reduce the complicated computation and steps taken to fixing the model. By allowing user to select from the list of recommended fixes, we were able to efficiently repair the model successfully dealing with state explosion.

Next, we have shown that our tool should be easily extensible so that model repairing is not limited to the property we considered and access control policies domain we addressed. We see our InteractiveFixer module as a "head" which we can change according to the model from another domain or other properties that need to be verified.

9. REFERENCES

[1] Loreto Bravo, James Cheney, and Irini Fundulaki. Repairing inconsistent xml write-access control policies. In *Proceedings of the 11th international conference on Database programming languages*, DBPL'07, pages 97–111, Berlin, Heidelberg, 2007. Springer-Verlag.

[2] V. Cridlig, R. State, and O. Festor. A model for checking consistency in access control policies for network management. In *Integrated Network Management, 2007. IM '07. 10th IFIP/IEEE International Symposium on*, pages 11 –19, 21 2007-yearly 25 2007.

[3] Kathi Fisler, Shriram Krishnamurthi, Leo A. Meyerovich, and Michael Carl Tschantz. Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 196–205, New York, NY, USA, 2005. ACM.

[4] Graham Hughes and Tevfik Bultan. Automated verifcation of access control policies. Technical report, 2008.

[5] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *Int. J. Softw. Tools Technol. Transf.*, 10(6):503–520, October 2008.

[6] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis.* The MIT Press, revised edition, 2012.

[7] Vladimir Kolovski and James Hendler. Xacml policy analysis using description logics. Technical report, 2008.

[8] Mahdi Mankai and Luigi Logrippo. Access control policies: Modeling and validation. Technical report, 2008.

[9] OASIS Open. *eXtensible Access Control Markup Language (XACML).* The MIT Press, 2.0 edition, 2005.

[10] Alexander Reder and Alexander Egyed. Computing repair trees for resolving inconsistencies in design models. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 220–229, New York, NY, USA, 2012. ACM.

[11] Nan Zhang, Mark Ryan, and Dimitar P. Guelev. Synthesising verified access control systems in xacml. In *Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, FMSE '04, pages 56–65, New York, NY, USA, 2004. ACM.

APPENDIX

A. VERIFICATION PROPERTY PREDICATE

The predicate below (written in Alloy) is used for automated verification of the following property:

Property 1: *Within a policy set that has OnlyOneApplicable policy combining algorithm, there is no two policies that for a given request return different decisions.*

```
pred InconsistentPolicySet [ps : PolicySet,
    req : Request,  p1: Policy, p2: Policy,
    r1: Rule, r2: Rule]{

  ps.combiningAlgo = P_OnlyOneApplicable
  p1 in ps.policies
  p2 in ps.policies
  p1 != p2
  r1 in p1.rules
  r2 in p2.rules
  policyResponse[p1, req] = Permit
  (
    p1.combiningAlgo = DenyOverrides and
    (no r1':Rule | r1' in p1.rules
      and ruleResponse[r1', req] = Deny)
    and ruleResponse[r1, req] = Permit
  )
  or
  (
    p1.combiningAlgo = PermitOverrides
    and ruleResponse[r1, req] = Permit
  )
  policyResponse[p2, req] = Deny
  (
    p2.combiningAlgo = PermitOverrides and
    (no r2':Rule | r2' in p2.rules
      and ruleResponse[r2', req] = Permit)
    and ruleResponse[r2, req] = Deny
  )
  or
  (
    p2.combiningAlgo = DenyOverrides
    and ruleResponse[r2, req] = Deny
  )

}
```