Shadowed Authorization Policies - A Disaster Waiting to Happen?

Ehtesham Zahoor[1](✉), Uzma Bibi[1](✉), and Olivier Perrin[2](✉)

[1] Secure Networks and Distributed Systems Lab (SENDS), National University of Computer and Emerging Sciences, Islamabad, Pakistan
{ehtesham.zahoor,uzma.bibi}@nu.edu.pk

[2] LORIA, Université de Lorraine, BP 239, 54506 Vandoeuvre-lès-Nancy Cedex, France
olivier.perrin@loria.fr

Abstract. Information security has been in the mainstream of computing for the last few decades and our increasing reliance on the large scale distributed systems, such as the Cloud, has put greater emphasis on the security capabilities of these systems. The security concerns are amongst the important factors affecting adoption of Cloud. This paper identifies and addresses issues concerning management of hierarchical authorization policies in the Cloud. These policy models pose the risk of policy shadowing where the decision taken at higher levels mask the possibly erroneous or conflicting policies specification at the lower levels. We introduce the notion of shadowed policies and present a model which is based on formal Event-Calculus (EC); for the identification of shadowed policies. The results show that our proposed approach is scalable and practical.

1 Introduction

The need to protect valuable information has always been there. During the early ages, the information about food and shelter was of utmost importance. We live in a digital world now and our valuable digital information, such as business plans, images and confidential documents, needs to be protected from the malicious access. Information security has thus been in the mainstream of computing for the last few decades and would remain same for the foreseeable future. As our information security capabilities have matured and increased, so are the challenges we are being faced with. In this context, the widespread use of internet and adoption of large scale distributed systems, such as Cloud, we are faced with more challenging environments to enforce security principles. The past decade has seen significant increase in the use of Cloud Computing, as many organizations either have private Cloud deployments or they are using services from public Cloud providers. All major Cloud providers thus provide advanced security mechanisms to handle security challenges. One approach to implement security principles within an organization is by the use of a policy. The authorization or access control policy of an organization specifies which users can access which resources, under what conditions, and what actions can they perform on the resources. All major Cloud providers thus provide the support of authorization policies and users can be organized into groups and policies can be assigned to them to specify their access permissions. Even with the advanced security capabilities provided by major Cloud providers, security breaches still happen resulting in loss of revenue and trust. In this context, authorization policies specification and management is a critical task and erroneous specification of even a single policy can lead to undesired consequences. The scale of the services from the Cloud providers makes the authorization management process challenging and complex and this can result in inducing more human errors.

One approach to improve the manageability of authorization process is the hierarchical access control model. In general, such policy models are evaluated from top to bottom, with each level providing more fine-grained access policies. This is the approach taken by major Cloud providers including AWS where policy specification and evaluation concerns different levels ranging from AWS organizations Service Control Policies (SCPs) to permission boundaries for a user or role. Hierarchical policy models pose the risk of policy shadowing where the decision taken at higher levels mask the erroneous or conflicting policies specification at the lower levels. Let us consider an example of hierarchical policy design having three levels, L0, L1 and L2, with the level L0 being the top most. If a permission is denied at the level L0, then any policy specification at the lower levels would be masked. The access control policy on the whole may produce the intended behavior but it poses serious challenges to policy management and any future change may result in erroneous behavior. As per the IBM sponsored 13th annual cost of a Data Breach study, more than one fourth of all breaches are triggered by human error. We believe that the shadowed policy's behavior is masked and not directly evident and thus a policy designer may tend to under constrain the rules assuming them to be handled at the higher level. The problem is amplified for the environments where the hierarchy structure itself can be dynamic, as we will highlight the case of AWS IAM policies where an Organization Unit can be moved within the AWS Organization tree.

We have identified the issues in our paper that are concerned to the management of authorization policies in the Cloud. We introduce the notion of shadowed policies and map their existence in the Amazon Web Services (AWS) Identity and Access Management (IAM) policies. We have highlighted the side-effects of having shadowed policies and how they induce more human errors. We introduce the notion of shadowed policies and present a model which is based on formal Event-Calculus (EC); for the identification of shadowed policies. The results show that our proposed approach is scalable and practical.

2 Background and Related Work

There are many facets of implementing information security within an organization and one way is through the use of security policies. After authentication, the decision of access control to the users is done by the authorization policies. The decision of authorization process is either to permit or deny the access and this decision may be evaluated in a certain context or conditions. One of the major studied area these days is access control and authorization policies management. One major subdomain has been the authorization models being used to implement the authorization process. In the Role Based Access Control (RBAC) model [1,2] authorization policies are based on roles of the users. RBAC has remained popular since its inception and all the major Cloud providers support RBAC. It does suffer from some scalability limitations including the role explosion. In order to ease management, the hierarchical RBAC introduces the concept of role hierarchy and inheritance and a parent role inherits all the permissions of inherited role [3]. For instance, there can be a role named *staff* and above in the role hierarchy can be a role named *manager* which implicitly inherits the permissions of the staff member. Hierarchical RBAC does provide ease of management but it makes difficult to enforce separation of duty (SoD) constraints. In Attribute-based access control (ABAC), the subjects, objects and the environment have attributes and the access decisions are made based on the boolean function on these attributes. ABAC can subsume RBAC and a detailed discussion on tradeoffs and characteristics of both models can be found in [4].

A number of approaches have addressed the need for formally modeling the authorization policies and verifying their consistency. In this context, authors have proposed a verification framework for conflicts detection in policies modeled through event-driven RBAC in [5]. In [6] authors have proposed a privacy preserving policy model and approach for handling policy conflicts. In [7] authors have proposed an approach to specify and verify authorization policies in the composition of Web services. A formal approach based on Fusion Logic for the specification and verification of properties such as consistency and SoD is discussed in [8]. In [9] authors have proposed an approach based on interval temporal logic for the specification and verification of temporal access control policies. In [10] authors have proposed an approach for the specification of policies in first order logic and then they use Prover9 theorem prover for proving proposed identity constraints. The verification of access control policies for SGAC is addressed in [11]. The authors have used *Alloy* and *ProB*, two first order logic model checkers. In [12] authors have introduced the concept of policy quality in terms of consistency, completeness, and minimality dimensions. There are many approaches that have addressed modeling and verifying the consistency of existing authorization languages. One such language is XACML (eXtensible Access Control Markup Language). Many approaches have been proposed to model and verify the consistency of XACML based policies [13–15]. In [16] authors have analyzed the specifications that handles the combination of authorization and management policies that detects inconsistencies and conflicts in policies. They have also modelled authorization in the behavior of system including policy specifications which is based on Event Calculus, but have not addressed shadowed authorization policies and their conflicts in specific. Authors have also addressed the problem of inconsistencies and conflicts in policies. Abnormal behavior is caused in a system due to these conflicting policies, hence, resolving these policy conflicts is highly significant. Chomicki and Lobo in [17] detect and resolve conflicts in ECA policies through a formal logic-based framework. Bandara in [18] analyzes and manages policies through a tool. The tool helps to query policies for validation and review. In [19] authors present a set of algorithms to check consistency among policies. None have addressed the policy conflicts and policy management in a hierarchal structure specifically targeting policy shadowing.

We believe that there is a research gap concerning both the consistency checking of authorization policies for the real world large scale distributed systems, such as Cloud, and the resolution of conflicts that concern policy management and thus stem from the implementation of policies at a larger scale. A formal ABAC based framework modeling policies and identifying inter and intra policy conflicts that exists between them is presented in [20] and authors have also proposed a model where policies from different cloud providers (AWS, GCP and Microsoft Azure) can be combined in a Multi-Cloud project [21]. This work addresses the conflicts related to policy management in the large scale distributed systems. One such conflict is policy redundancy and the redundant rules in an access control policy increase the size of the policy and would affect the performance and management of policies [22]. To best of our knowledge, there exists no approach that considers the case of shadowed authorization policies in large scale distributed systems. We have motivated the problem by presenting the case of AWS Identity and Access Management (IAM) and AWS Organizations, where policies exist at different levels and are evaluated based on a detailed and complex evaluation logic. We have justified the need for a formal approach and have both modified existing models for performance and correctness and presented new models needed for identifying shadowed policies. We have highlighted the side-effects of having shadowed policies and how they induce more human errors.

3 Case Study - AWS Policies Management

The authentication and authorization management service provided by AWS is called the Identity and Access Management (IAM) service. AWS IAM gives the opportunity of users management and their permissions and thus handles both authentication and authorization aspects. It provides a broad set of services ranging from managing users and their permissions to enabling multi-factor authentication (MFA) including auditing services. Policies are created and assigned to users, groups, roles, and resources, to achieve access management using IAM. When a request is made to access a resource, AWS evaluates different policies to reach a decision. The format of policies storage is JSON documents and follow specific syntax and structure. An example AWS policy is shown in Fig. 1. On a high level, AWS policies contain a set of statements and each statement represents a specific access control rule. Each statement contains the *service and resources* element which specify the AWS service, for instance *Amazon S3* and corresponding resource, for instance a S3 bucket, to which this statement applies. Further, using the *action* element of a statement one can specify what service-specific actions one is willing to perform on the resource specified earlier. The *conditions* element of the statement allows to further specify the conditions

```
{
    "Statement": [
        {
            "Sid": "Stmt01",
            "Effect": "Allow",
            "Action": [
                "aws-portal:ViewBilling"
            ],
            "Resource": [
                "*"
            ]
        }
    ]
}
```

Fig. 1. Single statement example from AWS IAM policy

under which the statement applies, for instance specifying the IP addresses from where the request arrives. Finally, the statement *effect* element specifies if the statement outcome is either *Allow* or *Deny* access. AWS supports different types of policies, these include the *Identity-based policies* which are attached to users, groups or roles and these policies grant permissions. The *resource-based policies* are attached to resources such as Amazon S3 buckets. *Permission boundaries* are assigned to users and roles and specify the maximum permissions can be granted to a user or role. In contrast to identity-based policies, permission boundaries on their own does not grant access but only limit the maximum access of identity-based policies. In order to discuss the *service control policies (SCPs)*, we first provide a brief introduction to AWS organizations.

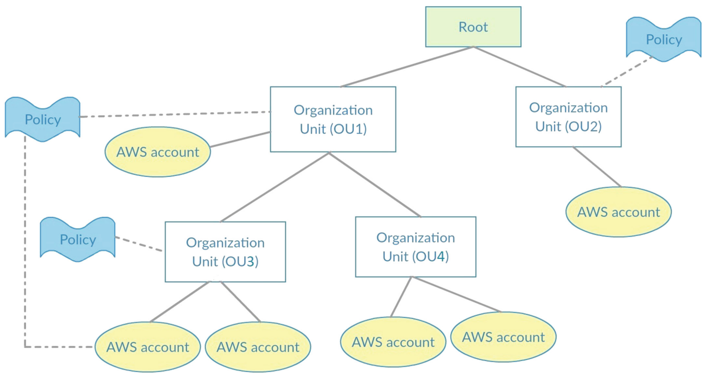

Fig. 2. An example AWS organization structure

AWS Organizations is an account management services by Amazon to provide services such as hierarchical grouping of accounts. Some key concepts include root, which is the top most container for all the accounts within the organization. Within a root, accounts are associated with the *Organization Units (*OUs*)*. Organization units can be organized in a tree-like structure (with fixed depth) and an *OU* can thus be part of another *OU*. Figure 2 shows an AWS organization with four organization units (*OUs*) each having associated accounts. The root is at the top. The type of access control policies applied at the account level granularity for AWS organizations are called Service Control Policies (SCPs). As similar to permission boundaries SCPs do not grant permissions but rather can be considered as a filter applied to the capabilities of an account. SCPs can be applied to different entities within an organization. If the SCP is applied to the root, it implicitly applies to all the *OUs* and associated accounts, as root is the top of the hierarchy. Similarly, if the SCP is applied to an Organization Unit (*OU*) it applies to that *OU* and any sub *OUs* associated with it.

AWS IAM is an example of hierarchical policy management as many such levels exists and there is a detailed and complicated policy evaluation process to check policies at each level in a top-down order. As the number of levels increase so does the risk of shadowed policies and in case of AWS Organizations, the *OUs* can themselves be nested and the maximum nesting supported by AWS is five *OUs* under the root. The SCPs associated with the higher level OUs are inherited at the sub OUs and thus any services blacklisted at the root results in shadowing SCPs at the lower levels. Before presenting the proposed approach, highlighting the risks of shadowed policies is highly significant. The shadowed policy's behavior is masked and not directly evident and thus a policy designer may tend to under specify the rules assuming them to be handled at the higher level. One such example can be of a permission boundary applied to a user that allows read only access to some resources. As the permission boundary is at the higher level, a policy designer may accidentally allow all actions on these resources using AWS IAM wildcards. The effective permission would remain to be read-only but this is a potential risk bound to happen. For an instance, permissions for a top level OU are changed or an *OU* is moved around in the AWS Organization tree.

4 Proposed Approach

The approach we present uses the formal logic based representation of policies to identify shadowed policies. There exist multiple authorization rules (statements in the context of AWS) at multiple levels. Rules at each level are evaluated and combined into a level policy and all the level policies are merged to identify shadowed policies. There can be more than one policy at a level, as supported by AWS. All the rules need to be modeled in Event-Calculus (is done automatically as the proposed models are generic) and we use a reasoner for Event-Calculus to evaluate rules and level policies. The algorithm below presents an abstract view of our proposed work. The algorithm is not optimized for performance but rather resembles the approach used in our EC models and the one taken by the EC reasoner. For simplicity, we assume that there exists a single policy at a level, however, the proposed approach handles multiple policies at same level, as shown later in Sect. 4. This can be the case when multiple sibling *OUs* have associate SCPs.

Algorithm 1. Identification of shadowed policies

```
Require: $rContext$ is the context to identify applicable policies.
 1: procedure DETECTSHADOWING($rSet, rContext$)
 2:   for each $level \in levelSet$ do                ▷ $levelSet$ is a set of all levels
 3:     for each $rule \in rSet_{level}$ do     ▷ $rSet_{level}$ is a set of rules at some level
 4:       $rdecSet_{level} \leftarrow evaluateRule(rule, rContext)$
 5:     end for
 6:     $policyDecision \leftarrow NotApplicable$
 7:     for each $rdecision \in rdecSet_{level}$ do
 8:       if rdecision = deny then
 9:         $policyDecision \leftarrow Deny$
10:         break
11:       end if
12:       $pdecSet_{level} \leftarrow policyDecision$
13:     end for
14:   end for
15:   for each $level \in levelSet$ do
16:     for each $nextLevel \in levelSet$ do
17:       if $pdecSet_{level} = Deny$ & $pdecSet_{nextLevel} = Permit$ then
18:         $pShadow = pShadow \cup (pdecSet_{nextLevel}, pdecSet_{level})$
19:       end if
20:     end for
21:   end for
22: end procedure
```

We iterate through every set of levels (named *levelSet*) and then for each level we iterate through all the rules associated with that level ($rSet_{level}$), lines 2–3. Then, at line 4, we evaluate each *rule* within the $rSet_{level}$ to identify if it permits or denies the access, or if it is not applicable. The decisions are based on the context and all the decisions are added to the decisions set for a level, $rdecSet_{level}$. Once we have a set of decisions for all rules within a level, we can combine them in a level policy and use a rule combining algorithm to conclude a policy decision. The rule combining algorithms include permit-overrides, deny-overrides and others. In practice, the approach taken by AWS is always deny-overrides, that is a single deny rule can cause the complete policy to be considered denied and we have thus used the same case in our algorithm, lines 7–12. We discuss other combining algorithms at the end of this section. We store the decisions of all level policies in *pdecSet* and the decision of policy at a *level* is represented by $pdecSet_{level}$, line 13. We can then identify the shadowed policies by iterating through level policy decisions, lines 15–20. We term a level policy to be shadowing if its level decision is 'deny' and there is a permit decision at the lower level, line 17, and we add the tuples of shadowing and shadowed policies to the set *pShadow*. Once we have identified tuples of shadowing and shadowed policies, it is further possible to identify the rules within the shadowing policy which are causing shadowing. This is needed as the rule combining algorithm being used is deny-overrides and it is possible that a single rule within the level policy is responsible. It can be accomplished by iterating through the rules within the shadowing policy and identifying the rules having deny decision. Let us conclude this section by highlighting the effect of rule combining algorithms and the shadowed policies. The rule combining algorithms specify how the combined individual decisions from multiple rules (and policies) reach a decision. The Deny-overrides combining algorithm considers a policy to be Denied even if it contains a single rule denying the access. Similarly, the Permit-overrides algorithm permits the access if a permit rule is there in the policy. Other algorithms include First-applicable and Only-one-applicable, but we limit our discussion to Permit and Deny-overrides. Let us consider the case of two level policies, pL0 and pL1 and each containing some rules permitting and denying the access. If we consider the deny-overrides to be the combining algorithm, both policies would evaluate to deny decision and even though they have the same decision, there are some permitting rules in pL1 that are being shadowed by pL0. The permit-overrides case is somewhat similar. We address this issue by considering a policy containing rules with multiple decisions to have intra-policy conflict [20] and resolving the conflicts before identifying the shadowed policies.

5 Event-Calculus Formalism

The proposed approach uses Event Calculus. It is a formal language used to represent events and their effects with reasoning. The choice of a formal approach is motivated by a number of factors. First, the authorization rules are evaluated not only based on syntactically matching subjects, objects, actions and decisions but rather on the environment or context as well, which may contain temporal (for example, the timing set for an access policy can be from 9am - 5pm) and other aspects. The subjects and other attributes may themselves have relations (for instance, Alice is member of group Users and some AWS resources is indeed part of S3 bucket). Then, the rules may be combined based on some rule combining algorithms into a policy. The policy shadowing itself can be based on a number of related aspects, such as the rule combining algorithms we discussed earlier or it may be the case that shadowing occurs only for specified time intervals or in some delegated scenario. The use of a formal expressive approach helps in collectively addressing these related aspects. In addition, EC has open-source tool support, *DECReasoner*[1]. The basic elements in EC are *events* (or actions), *fluents* (whose value can change on different time-points based on occurrence of events), and a set of predicates. Some predicates used in our models include the *Initiates(e, f, t)* predicate which specifies that if event e happens at time-point t then the fluent f holds after t. Similarly the *Terminates(e, f, t)* predicate

[1] http://decreasoner.sourceforge.net.

specifies that if event e happens at t then the fluent f does not hold after t. The *Happens(e, t)* predicate specifies that event e happens at timepoint t and the *HoldsAt(f, t)* is true iff fluent f holds at timepoint t. We use Event-Calculus [23] and we will only deal with the models that are simple and shows important aspects, without including the supporting axioms[2].

5.1 Rules Specification

The *rules* construct specifies one access rule. Each rule includes *Target*, an *Effect* and the *Conditions* associated with it. The EC meta-model for rules specification is shown below. The basic idea is to first decide if the rule is applicable in some context (achieved using RuleTargetHolds fluents and Match/Mismatch events) and then decide if the rule has any of the following conditions: permit, deny or not applicable, using fluents *RuleIsPermitted/Denied/NotApplicable* and Approve/DenyRule/RuleDsntApply events. The fluents are initialized in a way that when time=0, they do not hold and the reasoner should try to find a solution leading from initial state to reach the goal.

Model 1 (Meta-model for IAM Rules)

```
;Sorts for rules and their elements
sort rule, subject, object, action

;Fluents for Rules evaluation
fluent RuleTargetHolds(rule), RuleConditionHolds(rule)
fluent RuleEffectIsPermit(rule), RuleIsPermitted/Denied/NotApplicable(rule)
;Events for Rules evaluation
event (Mis)Match(rule), Approve/DenyRule(rule), RuleDsntApply(rule)

;These axioms link fluents with events
Initiates/Terminates (Match/Mismatch(rule), RuleTargetHolds(rule), time).
Initiates(Approve/DenyRule(rule), RuleIsPermitted/Denied(rule), time).
Initiates(RuleDsntApply(rule), RuleIsNotApplicable(rule), time).

;Conditions on events occurrence
Happens(ApproveRule(rule), time) → HoldsAt(RuleTargetHolds(rule), time) &
& HoldsAt(RuleEffectIsPermit(rule), time).
Happens(RuleDsntApply(rule), time) → !HoldsAt(RuleTargetHolds(rule), time).

;Initial state of the Fluents
!HoldsAt(RuleIsPermitted/Denied/NotApplicable(rule),0).
;The goal for the reasoner
HoldsAt(RuleTargetHolds(rule),1) / !HoldsAt(RuleTargetHolds(rule),1).
HoldsAt(RuleIsPermitted/Denied/NotApplicable(rule),2).
```

The model given above shows some EC sorts which can be considered as types for instantiating individual elements. For instance, *rule SomeRule* in an EC model would declare *SomeRule* to have type *rule*. We have used Initiates along with the definition of fluents, events and Terminates axioms to link them together. For instance, the Initiates axiom for RuleTargetHolds axiom specify that if the event Match happens at time point t, the fluent RuleTargetHolds

[2] Complete models along with setup and execution instructions are available at https://www.icloud.com/iclouddrive/0E4u-NuXGiGkpoql5BMamWhGQ#wise19.

would hold at t+1 and afterwards. The Terminates axiom for Mismatch event has opposite effect. The model above (and all other core models) are organized into files to be included in an EC model for any specific rule. This helps us in both manageability and have allowed us to develop automated tools to directly convert policies fetched from the Cloud (in JSON) to EC models. For instance, we modelled a specific rule, *L0Rule1*, to show the usage of generic model, given below:

Model 2 (Level0 rule specification)

```
load includes/rules/... ;generic model files
load includes/input.e ;Contextual attributes would be specified in input.e

rule L0Rule1
;Specifying when the rule target holds
Happens(Match(L0Rule1),time) →
{subject, object, action} subject = Alice & object = SomeRsrc & action = SomeActn.
Happens(Mismatch(L0Rule1),time)→subject!=Alice /object!=SomeRsrc /action!=SomeActn.
!HoldsAt(RuleEffectIsPermit(L0Rule1),0).
```

At first, the meta-model files are included, as shown earlier. The contents of file input.e will provide the context under which this rule needs to be evaluated such as the value of subject, object and action attributes. The rule is named as *L0Rule1* and we then a conditional axiom is stated that the event *Match* only holds true if there exists any match in the attribute name value pairs. We define the fluent *RuleEffectIsPermit* not to hold true at time=0, that is the rule denies the access and not permits. The rule model itself is very simple thanks to separating the core meta-model.

Solution 1 (Rule evaluation using DECReasoner)

```
0
RuleEffectIsPermit(L0Rule1).
Happens(Match(L0Rule1), 0).
1
+RuleTargetHolds(L0Rule1).
Happens(DenyRule(L0Rule1), 1).
2
+RuleIsDenied(L0Rule1).
```

If EC reasoner, *DECReasoner* are invoked, the solution shown above is returned. The reasoner first encodes the EC model in a SAT problem invoking an off the shelf SAT-solver (relsat in this case). The models found are then formatted to show events occurrence and fluents state at specific time-points. In this case, the event match happens at time-point 0 (as the value species in the input.e match the ones specify in the rule) and thus the RuleTargetHolds fluent holds at time-point 1 (shown with a + sign). Further, as the rule effect is not specified to be permit, the event *DenyRule* happens and the rule is considered denied. We can similarly model multiple rules in a level. We consider following additional rules, L0Rule2 concerns the user Bob and thus it does not apply. Similarly, the rule L0Rule3 concerns *SomeOtherActn* instead of *SomeActn* as specified in the input.e, so it does not apply as well. As per our shadowing identification logic, the rules at every level are grouped in a level policy. Instead of detailing policy meta-model and instantiation, we present the solution of level L0 policy that groups the three rules at level L0.

Solution 2 (Level policy evaluation using DECReasoner)

```
model 1:
0
Happens(Match(L0Rule1)/Mismatch(L0Rule2)/Mismatch(L0Rule3), 0).
1
+RuleTargetHolds(L0Rule1). Happens(DenyRule(L0Rule1), 1).
Happens(RuleDoesntApply(L0Rule2), 1). Happens(RuleDoesntApply(L0Rule3), 1).
2
+RuleIsDenied(L0Rule1). +RuleIsNotApplicable(L0Rule2).
+RuleIsNotApplicable(L0Rule3).Happens(DenyPolicy(L0Policy), 2).
3
+PolicyIsDenied(L0Policy).
```

The policy at level L1 is thus denied as it contains at least one rule as denying access. The policy evaluation algorithms in our case is chosen to be deny-overrides, permit-overrides and not applicable otherwise. As discussed earlier a policy cannot contain both permit and deny rules and is considered a conflict. In order to present policy shadowing identification models, let us consider another level L1, having two policies *L11Policy* and *L12Policy* (as per our naming convention, L11 means first policy of level 1). Both policies have three separate rules but intentionally tailored to have L11Policy resulting in permitting the access, while the L12Policy is not applicable. For the identification of shadowed policies, we can group multiple level policies and use EC axioms to identify shadowed policies.

Model 3 (Meta-model for the identification of shadowed policies)

```
predicate ParentOf(policy, policy) fluent PolicyShadowed(policy, policy)

event Shadowing(policy, policy) event NoShadowing(policy, policy)

Initiates(Shadowing(policy1,policy2), PolicyShadowed(policy1,policy2), time).
Terminates(NoShadowing(policy1,policy2), PolicyShadowed(policy1,policy2), time).

!HoldsAt(PolicyShadowed(policy1,policy2),0).
Happens(Shadowing(policy1,policy2),time) & ParentOf(policy1,policy2) →
HoldsAt(PolicyIsDenied(policy1), time) & HoldsAt(PolicyIsPermitted(policy2), time).

Happens(NoShadowing(policy1,policy2),time) & ParentOf(policy1,policy2) →
(!HoldsAt(PolicyIsDenied(policy1), time) /
(HoldsAt(PolicyIsDenied(policy1),time) & HoldsAt(PolicyIsDenied(policy2),time)) / (Holds
At(PolicyIsNotApplicable(policy1),time) / HoldsAt(PolicyIsNotApplicable(policy2),time))).
```

In the model above, we first define a predicate *ParentOf*, which defines the relationship amongst policies. Then, we define a fluent named *PolicyShadowed*, whose state would eventually represent if a policy is shadowing another policy. We then define some events and Initiates and Terminates axioms to link these events with the fluent. We further define some axioms to define that the event

Shadowing can only happen amongst policies if there exists a *ParentOf* predicate amongst them and if the parent policy is denying the access and the child policy is permitting the access. Similarly, we define that the event *NoShadowing* can only happen if there exists a *ParentOf* predicate amongst policies and there is no deny or permit relation amongst policies.

Model 4 (Model for aggregating level policies)

```
load includes/rules/... ;generic model files
load includes/policy/defined/L0Policy/L11Policy/L12Policy.e
load includes/input.e
;Contextual attributes would be specified in input.e

[policy1, policy2] (policy1 = L0Policy & policy2 = L11Policy) /
(policy1 = L0Policy & policy2 = L12Policy) <-> ParentOf(policy1, policy2).
```

The meta-model can be instantiated to specify a specific model for identifying shadowed policies and in the model above, we first include policy files for different policies and then define the *ParentOf* relations amongst them. More specifically, we define that *L0Policy* is parent of both *L11Policy* and *L12Policy*. The complete solution showing the evaluation results for policies and associated rules (if they are permitted, denied or not applicable) is returned, when the DECReasoner is invoked for the instantiated model above. The solution shows that the *L0Policy* does not shadow *L12Policy*, as one is denying access and the other is not applicable. The solution also shows that the *L0Policy* does shadow *L11Policy*, as one is denying the access and other is permitting the access. We have thoroughly tested our models on a number of complex configurations.

Solution 3 (Shadow policies identification using DECReasoner)

```
model 1:
0
RuleEffectIsPermit(L11Rule... L12Rule3).Happens(Match(L0Rule1/L11Rule1), 0).
Happens(Mismatch(L0Rule2/...L12Rule3), 0).
1
+RuleTargetHolds(L0Rule1/L11Rule1). Happens(ApproveRule(L11Rule1), 1).
Happens(DenyRule(L0Rule1), 1).
Happens(RuleDoesntApply(L0Rule2...L12Rule3), 1).
2
+RuleIsPermitted(L11Rule1). +RuleIsDenied(L0Rule1).
+RuleIsNotApplicable(L0Rule2...L12Rule3).
Happens(ApprovePolicy(L11Policy), 2). Happens(DenyPolicy(L0Policy), 2).
Happens(PolicyDoesntApply(L12Policy), 2).
3
+PolicyIsDenied(L0Policy). +PolicyIsNotApplicable(L12Policy).
+PolicyIsPermitted(L11Policy).
Happens(NoShadowing(L0Policy, L12Policy), 3).
Happens(Shadowing(L0Policy, L11Policy), 3).
4
+PolicyShadowed(L0Policy, L11Policy).
```

6 Performance Evaluation

For testing the correctness and scalability of our approach, we have created different test cases and evaluated them on Amazon EC2 c5.xlarge instance having

4 vCPUs and 8 GiB memory running Ubuntu Server 16.04 LTS. We have setup *DECreasoner* on the EC2 instance and have used the modified and improved version as proposed in [24]. Three test cases were evaluated; first we increase the number of levels with each level having maximum of two children each. As per our shadowed policies identification algorithm, multiple rules at a level are organized in a level policy. However, there can be multiple policies directly assigned to a level (as is the case with AWS Organizations) and these policies are at the same level. So, in the first test case, we assume the maximum policies at a level to be two. For the second test case we increase the number of policies at a level to a maximum of five and finally as an extreme case, we create a full binary tree having $2^h - 1$ policies, where h is the tree height (or implicitly the number of levels + 1). All the policies contain three rules and they are intentionally tailored to make the policies deny, permit and non applicable. Thus, if we have three policies, one is denied, the second one is permitted and the last one is not applicable.

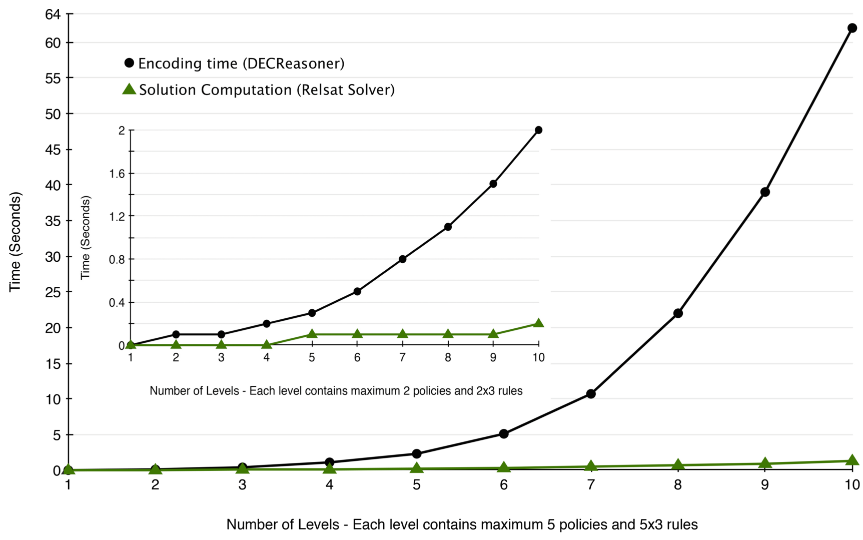

Fig. 3. Performance evaluation results

The performance evaluation results are shown in Fig. 3. In order to manage space limitations, we have merged the results of two different cases in a single figure. For both the cases, the Y-axis shows the time-taken in seconds while the X-axis shows the increase in the number of levels. The performance results are encouraging even though the EC to SAT encoding process does not scale well, a well known limitation of *DECReasoner*. The shadowing identification process does not require strict response time guarantees and is a occasional process used by policy designers to better manage the security policies of an organization. For simpler models, having two policies at a level and probably more common occurring use case, even at ten levels time taken by both encoding process and the SAT solver is around 2 s. For complex models, having five policies at a level and with 10 levels, the performance results are acceptable. We have evaluated further complex scenarios, which are rare to experience in practice, where we have a full binary tree and thus at a height of 6 the total number of policies is 63, each having three distinct rules. In this extreme scenario, not shown in the Fig. 3 due to space limitations, the time taken for EC to SAT encoding process is 3.5 min and the solution takes 2.9 s.

7 Conclusion

The proposed approach in the paper identifies and addresses issues concerning the management of hierarchical authorization policies in the Cloud based systems. These policy models pose the risk of policy shadowing where the decision taken at higher levels mask the possibly erroneous or conflicting policies specification at the lower levels. We introduce the notion of shadowed policies and to the best of our knowledge, there isn't any approach so far that deals with shadowed authorization policies in distributed systems. We have motivated the problem by presenting the case of AWS IAM. We have justified the need for a formal approach and have both modified existing models for performance and correctness and presented new event-calculus models needed for identifying shadowed policies. The results that we have presented show that our approach is scalable and practical.

References

1. Ferraiolo, D., Kuhn, R.: Role-based access controls. In: Proceedings of the 15th National Computer Security Conference, pp. 554–563 (1992)
2. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-based access control models. IEEE Comput. **29**(2), 38–47 (1996)
3. Sandhu, R.S., Munawer, Q.: The RRA97 model for role-based administration of role hierarchies. In: 14th Annual Computer Security Applications Conference (ACSAC 1998), Scottsdale, AZ, USA, 7–11 December 1998, pp. 39–49 (1998)
4. Coyne, E., Weil, T.R.: ABAC and RBAC: scalable, flexible, and auditable access management. IT Prof. **15**(3), 14–16 (2013)
5. Shafiq, B., Vaidya, J., Ghafoor, A., Bertino, E.: A framework for verification and optimal reconfiguration of event-driven role based access control policies. In: 17th ACM Symposium on Access Control Models and Technologies, SACMAT 2012, Newark, NJ, USA, 20–22 June 2012, pp. 197–208 (2012)
6. Wang, H., Sun, L., Bertino, E.: Building access control policy model for privacy preserving and testing policy conflicting problems. J. Comput. Syst. Sci. **80**(8), 1493–1503 (2014)
7. Rouached, M., Godart, C.: Specification and verification of authorization policies for web services composition. In: CAiSE 2007 Forum, Proceedings of the CAiSE 2007 Forum at the 19th International Conference on Advanced Information Systems Engineering, Trondheim, Norway, 11–15 June 2007 (2007)

8. Cau, A., Janicke, H., Moszkowski, B.C.: Verification and enforcement of access control policies. Formal Methods Syst. Design **43**(3), 450–492 (2013)
9. Janicke, H., Cau, A., Siewe, F., Zedan, H.: Dynamic access control policies: specification and verification. Comput. J. **56**(4), 440–463 (2013)
10. Sabri, K.E.: Automated verification of role-based access control policies constraints using prover9. CoRR abs/1503.07645 (2015)
11. Huynh, N., Frappier, M., Mammar, A., Laleau, R.: Verification of SGAC access control policies using alloy and prob. In: 18th IEEE International Symposium on High Assurance Systems Engineering, HASE 2017, Singapore (2017)
12. Bertino, E., Jabal, A.A., Calo, S.B., Verma, D.C., Williams, C.: The challenge of access control policies quality. J. Data Inf. Qual. **10**(2), 6 (2018)
13. Turkmen, F., den Hartog, J., Ranise, S., Zannone, N.: Formal analysis of XACML policies using SMT. Comput. Secur. **66**, 185–203 (2017)
14. Nguyen, T.N., Thi, K.T.L., Dang, A.T., Van, H.D.S., Dang, T.K.: Towards a flexible framework to support a generalized extension of XACML for spatio-temporal RBAC model with reasoning ability. In: ICCSA, vol. 5 (2013)
15. Kolovski, V., Hendler, J.A., Parsia, B.: Analyzing web access control policies. In: WWW, pp. 677–686 (2007)
16. Bandara, A.K., Lupu, E., Russo, A.: Using event calculus to formalise policy specification and analysis. In: 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2003), Lake Como, Italy (2003)
17. Chomicki, J., Lobo, J., Naqvi, S.: A logic programming approach to conflict resolution in policy management. In: 7th International Conference on Principles of Knowledge Representation and Reasoning (KR 2000), Morgan Kaufman (2000)
18. Bandara, A.K.: Formal approach to analysis and refinement of policies. PhD thesis, University College London, University of London (2005)
19. Agrawal, D., Giles, J., Lee, K.W., Lobo, J.: Policy ratification. In: Sixth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY 2005), pp. 223–232 (2005)
20. Zahoor, E., Asma, Z., Perrin, O.: A formal approach for the verification of AWS IAM access control policies. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 59–74. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_5
21. Zahoor, E., Ikram, A., Akhtar, S., Perrin, O.: Authorization policies specification and consistency management within multi-cloud environments. In: Gruschka, N. (ed.) NordSec 2018. LNCS, vol. 11252, pp. 272–288. Springer, Cham (2018). https://doi.org/10.1007/978-3-030-03638-6_17
22. Guarnieri, M., Neri, M.A., Magri, E., Mutti, S.: On the notion of redundancy in access control policies. In: 18th ACM Symposium on Access Control Models and Technologies, SACMAT 2013, Amsterdam, The Netherlands, 12–14 June 2013, pp. 161–172 (2013)
23. Mueller, E.T.: Commonsense Reasoning. Morgan Kaufmann Publishers Inc., Burlington (2006)
24. Zahoor, E., Perrin, O., Godart, C.: An event-based reasoning approach to web services monitoring. In: ICWS (2011)