Towards Automatic Update of Access Control Policy

Jinwei Hu[†‡]*, Yan Zhang*[†]*, and Ruixuan Li*[‡⋆]
[†]*Intelligent Systems Laboratory, School of Computing and Mathematics*
University of Western Sydney, Sydney 1797, Australia
[‡]*Intelligent and Distributed Computing Laboratory, School of Computer Science and Technology*
Huazhong University of Science and Technology, Wuhan 430074, China
jwhu@hust.edu.cn rxli@hust.edu.cn yan@scm.uws.edu.au
[⋆] *Corresponding author*

Abstract

Role-based access control (RBAC) has significantly simplified the management of users and permissions in computing systems. In dynamic environments, systems are subject to changes, so that the associated configurations need to be updated accordingly in order to reflect the systems' evolution. Access control update is complex, especially for large-scale systems; because the updated system is expected to meet necessary constraints.

This paper presents a tool, RoleUpdater, which answers administrators' high-level update request for role-based access control systems. RoleUpdater is able to automatically check whether a required update is achievable and, if so, to construct a reference model. In light of this model, administrators could fulfill the changes to RBAC systems. RoleUpdater is able to cope with practical update requests, e.g., that include role hierarchies and administrative rules in effect. Moreover, RoleUpdater can also provide minimal update in the sense that no redundant changes are implemented.

1 Introduction

Role-based access control (RBAC) [11, 35] simplifies access control management. In an RBAC system, users are assigned to roles such as manager and employee, and a role in turn is defined as a set of permissions. The key to RBAC is that users are assigned to roles and thus obtain roles' permissions, instead of being assigned permissions directly. Essentially, an RBAC configuration manages three kinds of relations: a user-role relation, a role-role relation, and a role-permission relation. The *user-role* relation assigns users to roles. The *role-role* relation describes how roles' permissions are inherited by other roles. The *role-permission* relation describes which permissions are accorded to each role. An RBAC system consists of two components, the RBAC configuration and the administration configuration. A running example RBAC system, which is used throughout the paper, is comprised of the RBAC configuration in Figure 1 and the administration configuration in Figure 2.

The role-role relation needs to be a partial order over roles; usually we refer to the role-role relation as a role hierarchy. The role hierarchy embodies two inheritance relationships among roles. Take the RBAC configuration in Figure 1 for example. (r_1, r_7) belongs to the hierarchy and we say r_1 is senior to r_7; it means that r_1 inherits all permissions of r_7 (i.e., p_3 and p_4) and that all members of r_1 are also members of r_7 or in other words, r_7 inherits all users of r_1.

RBAC is able to model a wide range of access control requirements, including discretionary and mandatory access control policies [30]. Hence, RBAC is widely supported in commodity operating systems and database systems [15, 17, 25], and is deployed inside many organizations [37].

We call a snapshot of an RBAC system an *RBAC state*. We denote the current state of the running example RBAC system as γ. Administrators can perform administrative actions to take an RBAC system from one RBAC state to another. Usually, the administration configuration is supposed to be static; that is, only the RBAC configuration may be changed. The actions available to administrators we consider are two types:

- `admin` **`assign`** p to r, and
- `admin` **`revoke`** p from r.

Administrators' powers are regulated by the administration configuration. We support variants of the PRA97 component of the ARBAC97 administrative model for RBAC [34]. The administrative model is instantiated by a set of assignment rules and a set of revocation rules. Figure 2 presents the administration configuration of γ. An *assignment rule* is of the form "ar can assign p to r if p assigned to c", which means an administrator in role ar can assign a permission p to r, if p is also assigned to

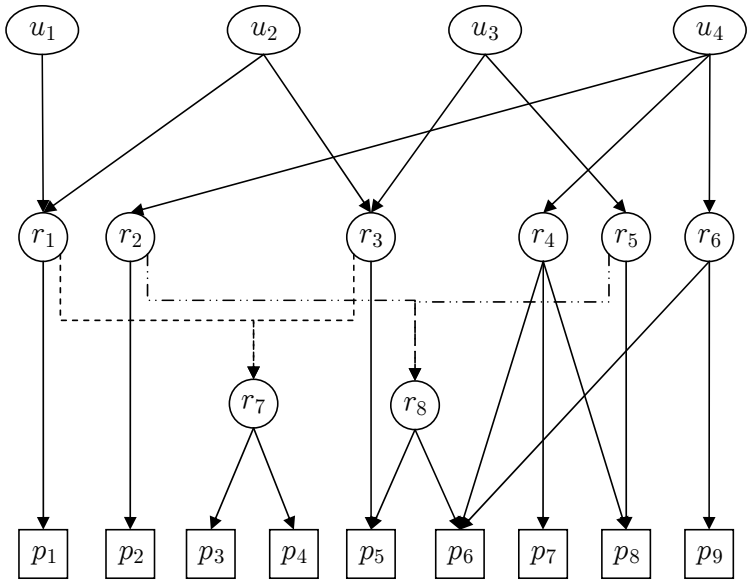


Figure 1: An example RBAC configuration. Users are represented as ellipses, roles as circles, and permissions as rectangles. Arrows between users and roles denote user-role assignments, arrows between roles and permissions denote role-permission assignments, and dashed arrows between roles denote role-role relationships (role hierarchy).

c. The expression c is constructed by roles and the connector $\wedge$. For example, consider the rule "ar_2 can assign p to r_1 if p assigned to $r_2 \wedge r_3$"; then the administrator $\texttt{admin}_2$ can assign a permission p to r_1 if p is assigned to r_2 and r_3.[1] A *revocation rule* is of the form "ar can revoke p from r", expressing that an administrator in role ar can revoke a permission p from r.

Update of RBAC systems is complex and challenging, especially for large-scale RBAC deployments. Existing tools mainly help administrators analyze and manage the RBAC system; they put little emphasis on suggesting to administrators how to configure the system. As shown in Figure 3a, with existing tools, administrators may have to update the system in a manual way. Figure 3b shows a typical process of manual update when one administrator is present. The administrator first determines and specifies, in some language, the update objective and the constraints that the final resulting system should satisfy. Usually, an update objective is initially formulated as high-level objectives (e.g., being able to assign $\{p_5, p_8, p_9\}$ to a user) . Arbitrary update may hinder the security and availability of the RBAC system. For example, revocation of a doctor's permission to write to a patient's medical record as a result of updating is not

[1]Consider, for example, the following situation: an administrator wants to enable an engineer to release the source code of a piece of software; however, the administrator can not do so unless the product manager and the quality manager are authorized to release the source code.

```
assignment rules:
  ar1 can assign p to r6
       if p assigned to r1 ∧ r2;
  ar2 can assign p to r1
       if p assigned to r2 ∧ r3;
  ar2 can assign p to r1
       if p assigned to r2 ∧ r4;

  ar0 can assign p to r1;
                ...
  ar0 can assign p to r6;

revocation rules:
  ar1 can revoke p from r4;
  ar1 can revoke p from r6;
  ar2 can revoke p from r1;
  ar2 can revoke p from r2;
  ar2 can revoke p from r3;
  ar3 can revoke p from r5;
  ar3 can revoke p from r6;

  ar0 can revoke p from r1;
                ...
  ar0 can revoke p from r6;

administrative role assignments:
  admin0 in ar0; admin1 in ar1;
  admin2 in ar2; admin3 in ar3;
```

Figure 2: An example administration configuration.

acceptable.

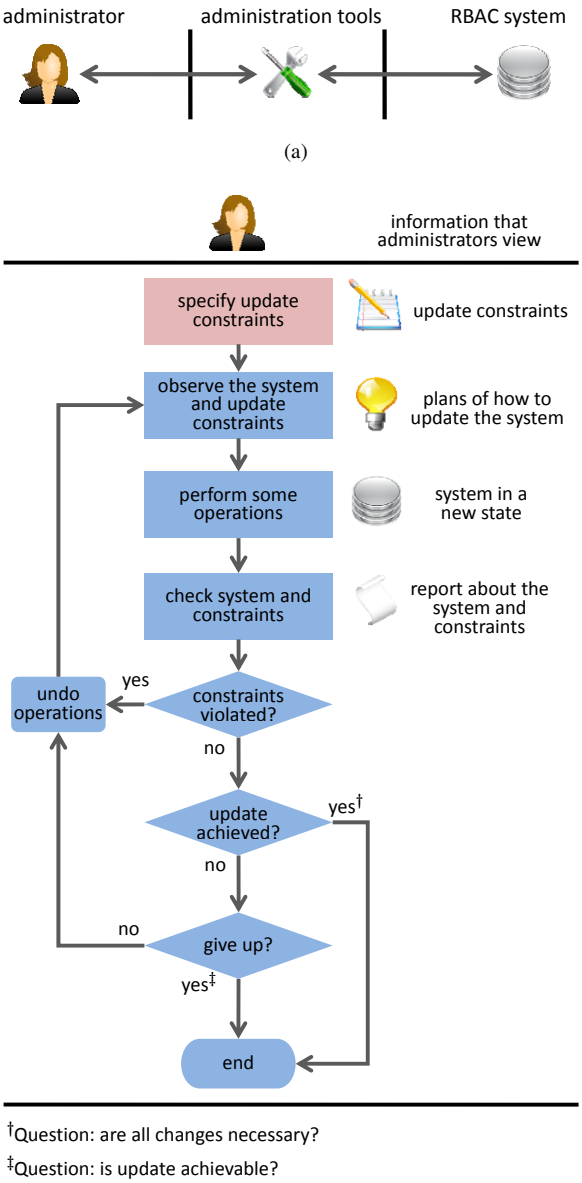
To modify system configurations, an administrator needs to observe the system and the constraints, and devises an update plan, which consists of a sequence of administrative actions. The administrator implements those actions, which take the system to a new state. There is no guarantee that all constraints are met and that this new state is the desired one. Hence, the administrator proceeds to check if these two conditions hold. When either one does not hold, the administrator may need to undo some previous actions and repeat the process. Roughly speaking, this is a trial-and-error approach. For large and complex systems, one can fail to achieve update after several trials; in this case, the question is whether to give up or not. Thus there arises a question: is the update achievable at all? An answer to this question helps the administrator make proper decisions. A positive answer implies that the update can be achieved and that the administrator should persevere in trying, whereas a negative one saves the administrator from continuing with pointless attempts.

On the other hand, suppose that the administrator finally manages to update the system without violating constraints. In this case, how different is the updated system from the original one? The less different it is, the more easier for one to understand and maintain the system, and thus the more preferable the update is. In other words, we may pursue an update that incurs minimal differences.

When multiple administrators are involved, the problem become more complicated. The actions an administrator can take might depend on others' actions. That is, administrators have mutual influence on each other in terms of administrative power. Cooperation among administrators is required in this case, which increases the complexity and cost of manual update. In summary, manual administration for update is work-intensive, inefficient and, when the objective is not achievable at all, very frustrating.

Access control update is demanded when security requirements are changed. In addition, RBAC systems may need updating in response to the following developing situations:

Misconfiguration Repair Misconfigurations in access control systems can result in severe consequences [4]. In a health-care situation, for instance, lack of legal authorization could lead to the delay of treatment. Modern access control systems include hundreds of rules, which are managed by different administrators in a distributed manner. The increasing complexity of access control systems gives rise to more likelihood of misconfigurations [2, 3]. As such, correcting misconfigurations is essential to systems' usability and security. Updating is necessary when misconfigurations in RBAC systems are detected.



(b) Workflow of manual update.

Figure 3: Illustration of updating without RoleUpdater.

Task Assignments To accomplish a task, a set of permissions should be assigned to a set of users to empower them to perform task operations [13]. For a new task, it is likely that the present RBAC configuration fails to enable exactly the needed user-permission assignments. In this case, administrators may resort to adjusting role configurations.

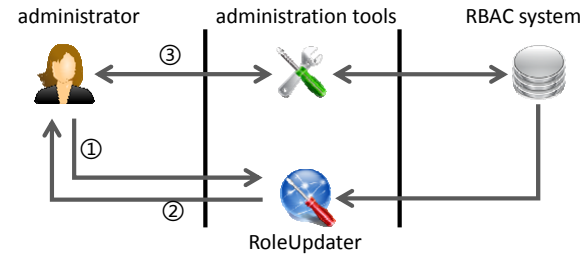
Property satisfaction An RBAC system should ex-

hibit various properties, including simple availability/safety and containment availability/safety [14, 22, 23, 24]. A simple availability/safety property asks whether a user Alice has a permission, e.g., access to a confidential file. Containment safety properties encode queries such as whether any user who can access printers are members of staff, whereas containment availability properties may ask whether all students have permission to use a library.

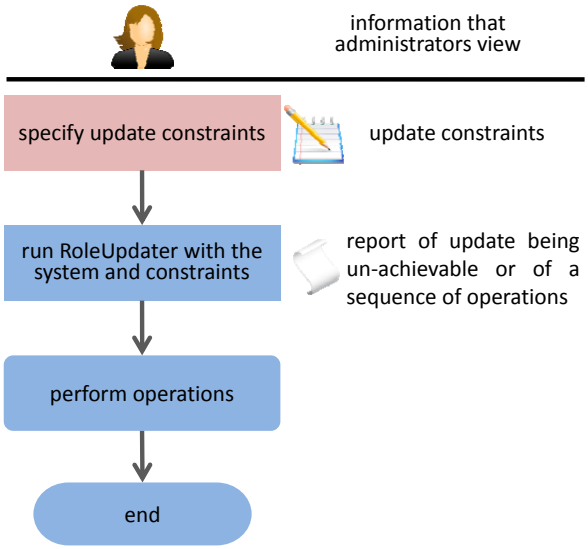
If an RBAC system was not designed with these properties in mind, it is unlikely that all properties would happen to hold. Particularly, for legacy systems, there is no guarantee of automatic establishment of security properties when they are migrated to RBAC management. On the other hand, even if all desired security properties hold currently, requirements are not static. For example, it may be desired that now only managers, instead of employees, have access to an internal document. To assure these properties, one may have to update the RBAC system.

Updating is a key component of maintenance in the RBAC life-cycle [18], and accounts for a great proportion of the total cost of maintenance [29]. RoleUpdater assists administrators with update tasks. As shown in Figure 4a, prior to updating the system, the administrator first interacts with RoleUpdater, and then manipulates the system using suggestions from RoleUpdater. Figure 4b shows the workflow of updating with RoleUpdater. The administrator still needs to specify the update constraints, and invoke RoleUpdater with the request. RoleUpdater checks, in an automatic way, whether the request achievable or not; and if so, a sequence of actions, which take the system to the expected state, is reported. RoleUpdater can also deal with the case where multiple administrators are involved.

RoleUpdater makes novel use of model checking techniques [6]. Figure 5a illustrates the basic idea of model checking. A model checker takes a description of a system and a property as inputs, and examines the system for the property. If the system exhibits the property, the checker reports that the property is true. If the system is found to lack the property, the model checker produces one counter-example. The counter-example, usually a sequence of system state transitions, explains how the system transits to a state where the property fails. Figure 5b illustrates how to use model checking as the basis for update. We check the property that the requested state is never reached; when the property does not hold, one is not only informed of the existence of an update but also a counter-example that corresponds to the update. RoleUpdater transforms update problems into model checking problems, where the failure of the model is synonymous with existence of a solution:



(a)



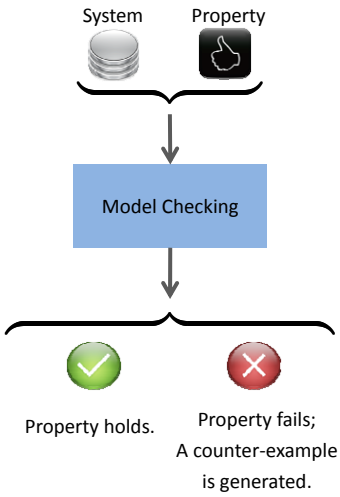
(b) Workflow of update with RoleUpdater.

Figure 4: Illustration of updating with RoleUpdater.

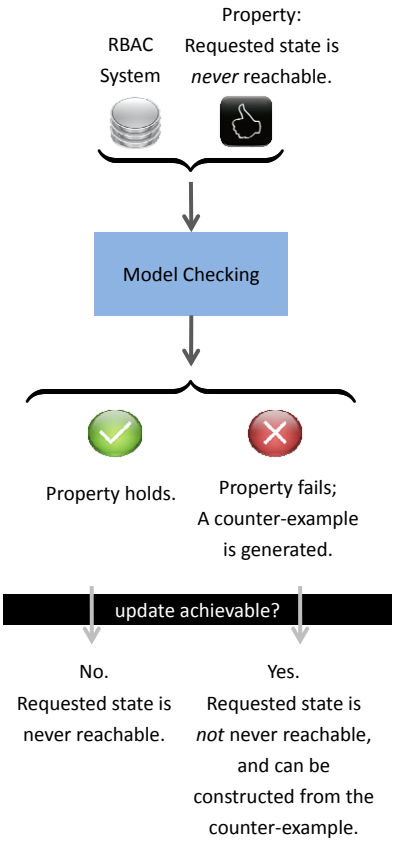
- if the property is determined to be true, the update objective is not achievable;
- otherwise, the model checker returns a counter-example, from which an update is constructed.

RoleUpdater employs NuSMV [5] to perform model checking. NuMSV is a open-source symbolic model checker. For better performance, a collection of reductions and optimization techniques are implemented in RoleUpdater.

The rest of this paper is structured as follows. Related works are given in Section 2. We demonstrate the use of RoleUpdater by showing how it handles a high-level update request specification in Section 3. Section 4 presents the design and implementation of RoleUpdater. We show some experimental results of running RoleUpdater in Section 5, illustrating its effectiveness and efficiency. Section 6 concludes the paper.



(a) The basic illustration of model checking.



(b) Update via model checking.

Figure 5: Illustration of model checking and its usage for updating.

2 Related Work

RBAC administration and analysis Many convenient RBAC administration models (e.g., [8, 21, 34]) are at our disposition. They provide significant advantages in access control management. They define administrative rules, e.g., specifying which administrator can perform what operations. However, high-level update is rarely supported. It is generally difficult and error-prone, because usually the resulting state is expected to meet various constraints.

To help administrators understand RBAC policies, various RBAC policy analysis tools (RPATs) have been invented [4, 14, 23, 38, 39, 44]. RPATs usually answer if an RBAC system satisfies a property. However, little effort has been devoted to answering the question: what if the RBAC system fails to meet the property? When administrators find abnormalities with RPATs, RoleUpdater can assist in correcting them.

Most security analysis problems in literature basically can be stated as: given the current state γ, a query q (e.g., whether accesses to internal documents are only available to employees), and a state-change rule φ, can γ be taken to a state γ' where q evaluates to true? If this is the case, the steps taking γ to γ' may also be reported to administrators so that they can follow them to make γ'. However, as the objectives are different, we believe this kind of reporting could hardly be considered sufficient for the role updating problem. RPATs' objective is to analyze the system. So, their input is just the property to be examined. By contrast, RoleUpdater aims to update the system; the input is the update request. RPATs explore every possible sequence of actions, as long as they are allowed by φ, to test if there is such a γ' where q is true. In this case, administrators do not have any control of the resulting state. By contrast, RoleUpdater seeks a resulting state that complies with administrators' request. In addition, most RPATs focus on user-role assignments. Although it is argued that the role-permission relation might be treated similarly to the user-role relation, the role-permission relation also deserves its own attention [29], especially in terms of role updating.

Various access control properties are proposed and verification schemes are devised to check the satisfiability of properties. In [23], authors propose a tool to answer a set of interesting properties, including simple availability/safety, bounded safety and containment availability/safety. The tool provides a means to guarantee that security requirements are always met as long as trusted users abide by certain behavior patterns [22, 23]. However, an assumption is needed for the usage of security analysis: the properties hold in the current RBAC state [22, 23]. As mentioned above, this is not always the case. Role updating can be used to adjust the current RBAC state so as to exhibit desired properties, while keeping the changes to the customized extent.

```
update
  make P = {p5, p8, p9} available via T = {r1, r2, r3, r4, r5, r6}
  with
    administrators admin1, admin2;
    user-permission constraints
      (u1, no-less-than {p1}, no-more-than {p1, p3, p4}),
      (u2, no-less-than {p1, p3, p4, p5}, no-more-than {p1, p3, p4, p5}),
      (u3, no-less-than {p3, p4, p5}, no-more-than {p3, p4, p5, p6, p8}),
      (u4, no-less-than {p7, p8, p9}, no-more-than {p3, p5, p6, p7, p8, p9});
    restricted-role constraints
      (r4, no-less-than {p6, p7}, no-more-than {p6, p7, p8, p9}),
      (r8, no-less-than {p5, p6}, no-more-than {p5, p6});
    role-hierarchy = {(r2, r8), (r3, r7)};
    minimal;
```

Figure 6: An example high-level update request specification.

Role engineering Role engineering attracts much research effort [7, 10, 26, 40, 41, 45]. Existing role engineering tools (eRETs) take user-permission assignments as input and output user-role assignments and role-permission assignments. eRETs may take into account some other information such as business meanings, semantics, and users' attributes. Taxonomically, RoleUpdater can be viewed as a role engineering tool. However, role updating works when RBAC states have been defined and possibly deployed, whereas eRETs usually define roles from scratch. The focuses are also different. Role updating aims to answer administrators' question whether an update is achievable with respect to update constraints and how to generate one, if any. By contrast, eRETs put more emphasis on how to define an appropriate set of roles. In the context of a role life cycle, RoleUpdater is for role maintenance, while eRETs help with role design. Thus, one may consider RoleUpdater as a complement to eRETs; RoleUpdater can be used to fine-tune the ideal state generated by eRETs.

RBAC udpate Ni et al. [29] studied the role adjustment problem (RAP) in the context of role-based provisioning via machine-learning algorithms. Though similar, the role updating problem differs from the RAP in several aspects. First, customized constraints on updates are enforced in RoleUpdater, whereas it is unclear if these constraints could be supported in RAP. Second, our role updating is request-driven, whereas RAP is a learning process. RAP and RoleUpdater are both assistant tools for administrators but with different usage and orientation.

Fisler et al. [16] investigated the semantic difference of two XACML policies and the related properties. However, they do not consider how to make a different desired state from the current one. Ray [32] studied the real-time update of access control policies, in the context of database systems. They focused on transaction properties, instead of RBAC policies.

```
admin2 assign p8 to r1;
admin2 assign p8 to r2;
admin1 assign p8 to r6;
admin2 revoke p8 from r1;
admin2 revoke p8 from r2;
admin1 revoke p6 from r6;
admin2 assign p5 to r1;
admin1 assign p5 to r6;
admin2 revoke p5 from r1;
```

Figure 7: The update returned by RoleUpdater when running with the request in Figure 6.

3 High-Level Update Request Specifications

We do not consider the update of user-role assignments, because users' role memberships are determined by their attributes, jobs, titles, etc. When this information is renewed, administrators can accomplish user-role assignments straightforwardly.

Suppose the administrators want to update the RBAC configuration in Figure 1. Suppose further that the administrators specify the update request as in Figure 6. This specification expresses the customized conditions on the potential updated system. In the rest of this section, we illustrate the use of RoleUpdater through this example. Running with this example, RoleUpdater returns the steps, as shown in Figure 7, that the administrators can follow to make the changes; in the updated state, the administrators can assign $\{p_5, p_8, p_9\}$ to users via r_6.

Administrative power Line 4 specifies which administrators are going to update the system. As mentioned before, it is common for administrative rules to regulate administrators' operations; that is, administrators have limited administrative power. A proposed update does not make sense unless the needed changes lie within administrators' capabilities.

RoleUpdater appears more useful when multiple administrators are involved. Observe the five actions by $\texttt{admin}_1$ and $\texttt{admin}_2$: $\texttt{admin}_2$ assigns p_8 to r_1 and r_2, $\texttt{admin}_1$ assigns p_8 to r_6, and $\texttt{admin}_2$ revokes p_8 from r_1 and r_2. These interleaving operations require close cooperation between $\texttt{admin}_1$ and $\texttt{admin}_2$ and careful examination. By contrast, RoleUpdater takes the cooperation among administrators into account automatically.

Suppose that we replace Line 4 with the following.

administrators $\texttt{admin}_3$;

That is, the administrator $\texttt{admin}_3$, instead of $\texttt{admin}_1$ and $\texttt{admin}_2$, wants to update the system. Then RoleUpdater suggests an alternative: first revoke p_6 from r_5 and then revoke p_6 from r_6. However, $\texttt{admin}_1$ and $\texttt{admin}_2$ are not authorized to perform this alternative. Note that administrators' powers are configured in Figure 2.

Controllable effects Administrators should be able to confine the effects of an update. With RoleUpdater, administrators can specify a certain set of users U and define what changes could happen to users' permissions. For example, Alice at least has access to files under "/foo/bar1" but at most "/foo/bar1" and "/foo/bar2". Line 5 to Line 9 are constraints on users' permissions after update. For example, by Line 6, administrators requires that u_1 have at least permission p_1, but at most p_1, p_3, and p_4 in the potential new state. Note that users still obtain permissions via roles and even that users' role assignments remain the same.

For another example, Line 7 prescribes that u_2's permissions are exactly $\{p_1, p_3, p_4, p_5\}$. Consider the solution in Figure 7; administrators have to revoke p_8 from r_1, for u_2 is assigned to r_1 and cannot have permission p_8, as required by Line 7.

By properly specifying constraints, administrators guarantee the tasks associated with users in U progress smoothly. Suppose that u_2 and u_4 cooperate to finish a task $\texttt{t}$, which requires that u_2 and u_4 are entitled to privileges $\{p_1, p_3, p_4, p_5\}$ and $\{p_7, p_8, p_9\}$, respectively. Then Line 7 and Line 9 guarantee that the updated state, if any, would not disable $\texttt{t}$.

When administrators are specifying U, U often contains those users for whom the administrators are not responsible so that they have to ensure that the potential update does not affect such users, and/or those users whose permissions are designated by the administrators and vary within a range. For users outside the set U, their current role assignments and permissions in γ are neglected by RoleUpdater; that is, updates may change their role-assignments and permission-assignments.

Restricted update The principle of *least privilege* is important in computer security and well supported by RBAC. Users activate only the roles necessary to finish the underlying work, but not all assigned roles. For example, a user Alice may activate the role manager when she wants to evaluate an employee under her department, and activates the employee role for routine works. As a result, upper bounds should be put on roles' permission sets in compliance with the least privilege principle. On the other hand, some roles are designed with expected functions; users should be able to perform a particular job with such a role. If associating with the role a set of permissions less than necessary, administrators may make the role useless. Hence, it would be handy if administrators are able to set the permission sets of certain roles within a range.

Line 10 to Line 12 shows constraints on roles' permissions after update. For each selected role (e.g., r_4), administrators can impose a lower bound (e.g., $\{p_6, p_7\}$) and an upper bound (e.g., $\{p_6, p_7, p_8, p_9\}$) on the role's permissions. RoleUpdater assures that the role is assigned to permissions no less than those in the lower bound and also no more than those in the upper bound.

A requirement is that, the upper bound (or the lower bound) of the range should be a superset (or subset) of the set of all permissions that r is currently assigned in γ. This is reasonable, because the permissions r has currently in γ are enough to make it useful. We also find that, without this requirement, RoleUpdater's efficiency degrades.

Line 12 indicates that r_8's permissions must still be $\{p_5, p_6\}$ after update, because the lower bound equals the upper bound. We call roles like r_8 *invariant roles*. Despite the importance of update, it is likely that administrators demand some roles be invariants in order to, for example, preserve roles' intuitions, business meanings or definitions. In this case, by letting the lower bound of r be its upper bound, administrators request RoleUpdater to find an update which does not change r's permission assignments. In other words, RoleUpdater may change those non-invariant roles' permission assignments in the hope to find an update. In practice, non-invariant roles are usually the ones under administrators' control; otherwise, even though an update is found, administrators would not be able to implement it and thus the update is of little value.

If the administrators impose another restricted-role

```
admin2 assign p8 to r1;
admin2 assign p8 to r2;
admin1 assign p8 to r6;
admin2 revoke p8 from r1;
admin2 revoke p8 from r2;
admin1 revoke p6 from r6;
admin2 revoke p3 from r3;
admin2 revoke p4 from r3;
```

Figure 8: An alternative when the role hierarchy (r_3, r_7) is not required.

constraint besides those in Figure 6.

$$(r_6, \textbf{no-less-than}\ \{p_6, p_9\},\ \textbf{no-more-than}\ \{p_6, p_9\})$$

Then RoleUpdater reports that the requested update does not exist, which is indeed the case.

Role hierarchy Role hierarchy is recognized by the proposed NIST standard for RBAC as one of the fundamental criteria [11]. It further mitigates the burden of security administration and maintenance. Usually, there could be a natural mapping between role hierarchy and organization's structure. It is imprudent to alter a role hierarchy arbitrarily. Administrators can ask RoleUpdater to preserve the whole or part of the original role hierarchy. Line 13 tells that r_2 and r_3 are still senior to r_8 and r_7, respectively, in the updated system.

The requirement that r_3 be senior to r_7 stops RoleUpdater from suggesting another solution, as shown in Figure 8. If following this approach, administrators can assign $\{p_5, p_8, p_9\}$ via r_3 and r_6; however, r_3 is no longer a senior role of r_7.

Minimal update As long as an update is implemented, some changes are made to the system. When two update solutions are available, which one is more preferable? One perspective is to compare the changes they recommend. The fewer changes are needed, the closer the resulting state to the original state. Ideally, we may find an update such that none of its changes is redundant; that is, failure to implement any change thereof gives rise to a disqualified state. We say the update is *minimal*.

Minimal update is valuable in several ways. First of all, minimal update causes few difficulties for administrators to understand the new RBAC state. The administrators are responsible for the maintenance of the RBAC system. It is essential for them to comprehend the system's behavior. We can assume that administrators understand the system well before updating. However, changes to the system configuration have the potential to obfuscate the system. Obviously, a smaller gap between the updated state and the original one usually means a smaller degree to which administrators have to re-examine and re-learn the system.

```
admin2 assign p8 to r1;
admin2 assign p8 to r2;
admin1 assign p8 to r6;
admin2 revoke p8 from r1;

admin1 revoke p6 from r6;
admin2 assign p5 to r1;
admin1 assign p5 to r6;
admin2 revoke p5 from r1;
```

Figure 9: Update in response to the request in Figure 6 but without the minimality requirement.

Secondly, minimal update possibly preserves more previously computed analysis results. It is reasonable to assume that the current RBAC state satisfies necessary properties (otherwise, it should have been adjusted). It is likely that more properties might be preserved with minimal update. Finally, minimal update is also desirable when authorization recycling is deployed in access control implementation [42, 43].[2]

In RoleUpdater, administrators can choose to require each returned update to be minimal in the sense that no change is redundant. However, there is a tradeoff between doing this and incurring extra computing overhead. In Figure 6, Line 14 indicates administrators' willingness to find a minimal update. If turning the minimal requirement off, RoleUpdater would possibly not insist on the revocation of p_8 from r_2, for p_8 being assigned to r_2 does not contradict with the constraints. That is, RoleUpdater returns the update in Figure 9.

4 Design and Implementation

Figure 11 shows the architecture of RoleUpdater. Its interface accepts administrators' input and parses the request. We say a request is *canonical* if (1) all administrative operations are available, (2) users' permissions are required to remain unchanged, and (3) no role hierarchy is required to be preserved. Figure 10 shows an example canonical request, where P_i is the set of permissions that user u_i has prior to updating.

[2] Authorization decision-making is time-consuming and costly. Authorization recycling caches the authorization decisions that are made previously and infer decisions for forthcoming authorization requests. As an important mechanism for access control implementation, authorization recycling makes use of "cache" to enhance performance; there, policy update is a major concern. For details, readers are referred to [9, 42, 43].

```
update
  make P available via T
  with
    administrators all-administrators
    user-permission constraints
      (u1, no-less-than P1,
                   no-more-than P1),
      (u2, no-less-than P2,
                   no-more-than P2),
      ...;
    restricted-role constraints ∅;
    role-hierarchy = ∅;
```

Figure 10: An example canonical request.

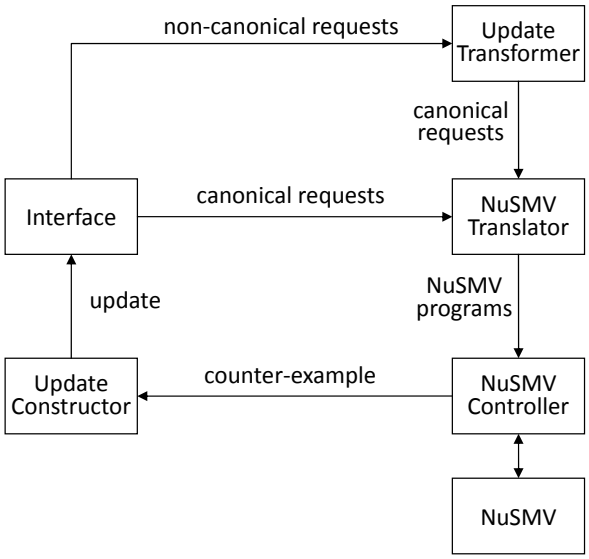


Figure 11: The architecture of RoleUpdater.

If canonical, the request is forwarded to the NuSMV Translator; otherwise, it is first processed by the Update Transformer, where non-canonical requests are transformed into canonical ones. Afterwards, the NuSMV Translator converts requests into NuSMV programs. The NuSMV Controller invokes NuSMV to execute those programs. According to the results returned from NuSMV, the Update Constructor generates an update report, either a sequence of administrative operations which lead to desired RBAC system state or a message that the request is unachievable.

Algorithm 1 presents RoleUpdater's pseudo-code. Line 2 belongs to the Interface module. Line 3 represents the Update Transformer. Line 4 and Line 5 are the main components of the NuSMV Translator module. Normally, the NuSMV Translator would create a set G of NuSMV programs for each canonical request. However, since the execution of the NuMSV programs translated directly from the request easily result in state explosions and memory crashes, some reductions are performed in advance [12]. The set G has the property: an update is found, if and only if, the run of NuSMV with at least one program in G reports a counter-example. As indicated by Line 6, on receiving the NuSMV programs, the NuSMV Controller schedules NuSMV programs in increasing order by the number of variables, because NuSMV's performance highly depends on the number of variables in the input program. The NuSMV Controller proceeds to execute each program with NuSMV; if any execution returns a counter-example, it informs the Update Constructor of the counter-example. The Update Constructor generates the needed update and administrative operations necessary to institute the changes (Line 10 to Line 12). If minimal update is required, further processing (Section 4.3) will be done. In the rest of this section, we give details of each component.

4.1 Handling non-canonical requests

We tried to use the model checking approach directly to evaluate non-canonical update requests. Our experience is that, an extensive number of variables are needed to model complex requests, which often gives rise to state explosions and memory crashes. The reasons are twofold. First, non-canonical requests enable much more potential combinations of role-permission assignments than canonical requests do. Second, some reductions in [12] are not applicable to non-canonical ones. It is not clear how to reduce non-canonical requests effectively.

Consider a non-canonical update request issued against γ in Figure 12. Non-canonical requests are transformed into canonical ones by adding dummy elements (e.g., users, roles, user-role assignments, and role-permission assignments) to γ; these dummy elements simulate those non-canonical conditions on the update. Usually, the obtained RBAC state, against which the canonical request is checked, is more complicated than γ. Fortunately, the construction is polynomial. We trade off the simplicity of RBAC states for the ability to cope with complex updates. By this modeling, we need only to focus on one unified problem: evaluating canonical requests.

4.2 NuSMV program generation

NuMSV is the symbolic model checker that RoleUpdater employs to perform model checking. The NuSMV Translator converts update requests into NuSMV programs. A set of boolean variables are defined to model the RBAC system. To use NuSMV, let ϕ denote the statement that a user could acquire exactly the permissions in $\mathcal{P}$ via roles in $\mathcal{T}$; we ask if $\neg\phi$ is always true in all reach-

Algorithm 1: Algorithm of RoleUpdater.

Input: High-level update request H, γ, and NuSMV property *type*: "AG" or "$AX\ AG$"
Output: update report

```
1  begin
       /* Parse(H,Q) parses H and reads information into Q; it returns a boolean value
          showing if any error happened.                                          */
2      if !Parse(H, Q) then show error message;
3      if Q is non-canonical then Q ←TransCanonical(Q);
       /* perform reductions on Q                                                 */
4      Reduce(Q);
5      G ←TransNuSMV(Q, type);
       /* NuSMV's performance highly depends on the number of variables in the input
          program; so schedule NuSMV programs in increasing order by the number of
          variables.                                                              */
6      S_G ←Schedule(G);
7      foreach g ∈ S_G do
8          Invoke g with NuSMV;
9          if a counterexample is returned then
10             construct an update γ′ from the counter-example;
11             if Minimal update is required then γ′ ←Minimize(Q, γ, γ′);
               /* compute the needed administrative operations that take the RBAC system
                  from γ to γ′                                                    */
12             AdminOp ←computeAdminOperation(γ, γ′);
13             show AdminOp and γ′;
14             return γ′;
15     show "update unachievable" report;
16     return ϵ;
17 end
```

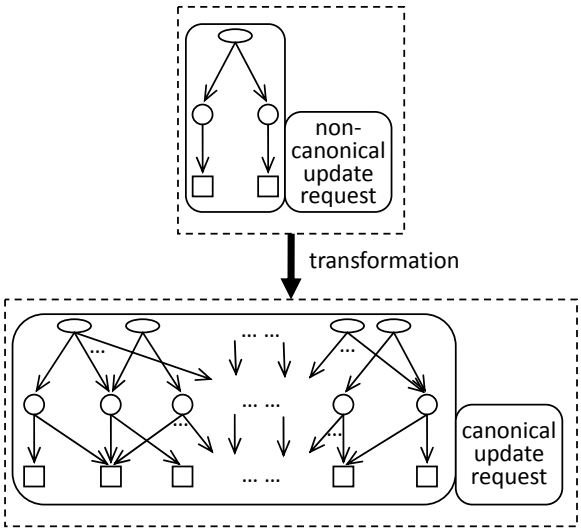


Figure 12: An illustration of the transformation from non-canonical update requests to canonical ones.

able (NuSMV) states;[3] If it evaluates as true, the user can never obtain exactly all permissions in $\mathcal{P}$ via roles in $\mathcal{T}$, indicating that one cannot fulfill the request without violating the update constraints. Otherwise, NuSMV will generate a counter-example, from which RoleUpdater constructs an update.

In the current implementation of RoleUpdater, only boolean variables and `TRANS` declarations are used. An RBAC state is represented by a valuation of boolean variables, whereas `TRANS` declarations capture transitions among RBAC states. Further explorations of NuSMV features and other model checking techniques could improve RoleUpdater's efficiency.

4.3 Minimal Update

Interestingly, the minimal update can be obtained in the same way we seek an update. Once an update is found, denote the RBAC state after update as γ'. As illustrated in Figure 13, if a role-permission assignment appears in exactly one of γ and γ', this assignment is changed

[3] ϕ is defined over the boolean variables. The checked property is $AG\neg\phi$, where A means always and G means globally; $AG\neg\phi$ is a CTL (Computational Tree Logic) formula, which is used to specify properties in NuSMV.

(either removed or added); denote the set of all such changed assignments as CA. Then the minimal update requirement is to determine if all changes in CA are necessary. The basic idea is to ask if the same goal could be achieved with a proper subset of CA. To answer this, we define variables to simulate CA and treat assignments outside CA as constants. This is done by adding dummy elements and imposing new update constraints. A new update request is issued against RoleUpdater; this request is the same as the original one except that new restricted-role constraints are added.

However, the checked property is whether, starting from the next state of γ', all reachable states satisfy $\neg\phi$.[4] If so, then γ' itself is minimal. Otherwise, from the returned counter-example, we could obtain γ''. This γ'' is closer to the minimal update than γ', because only a proper subset of CA is implemented. Note that this is a recursive process; and thus a minimal update could be reached.

Take the request in Figure 6 for example. Figure 14 shows an example calling stack of RoleUpdater. Receiving a request with the minimality requirement, RoleUpdater first removes this requirement and searches for an update. Suppose that RoleUpdater finds the update shown in Figure 9; it proceeds to compute CA, which is $CA = \{(r_6, p_8), (r_2, p_8), (r_6, p_5), (r_6, p_6)\}$. By composing a new update request, RoleUpdater goes on checking if there exists such an update that the resulting changes are a proper subset of CA. This starts a recursive call. Then the same processes are applied. This time, an update shown in Figure 7 is found and CA is computed to be $\{(r_6, p_8), (r_6, p_5), (r_6, p_6)\}$. Again, another round commences. This time, RoleUpdater could not find any update, which implies that the update in Figure 7 is minimal; RoleUpdater returns this update in response to the original request.

5 Experiments

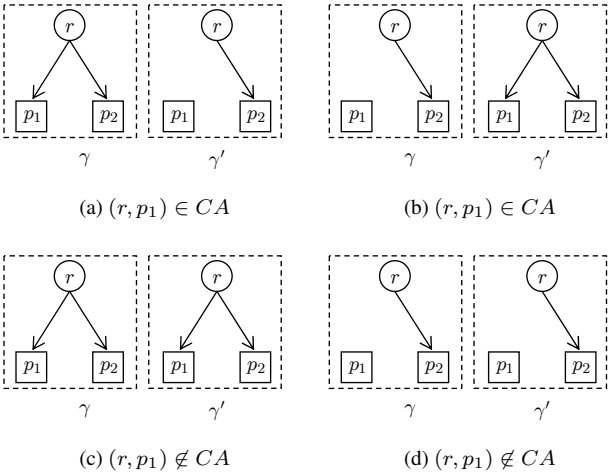
We implemented a prototype of RoleUpdater in Java. Experiments were performed with randomly-generated RBAC systems on a machine with an Intel(R) Core(TM)2 CPU T5500 @ 1.66GHz, and with 2GB of RAM running Microsoft Windows XP Home Edition Service Pack 3.

Data generation

To generate each RBAC system, we adapted algorithms from [41, 45][5]; γ is parameterized by noU (the number of users), noR (the number of roles), noP (the number

[4] In NuSMV, this is expressed by $AXAG\neg\phi$ in CTL.

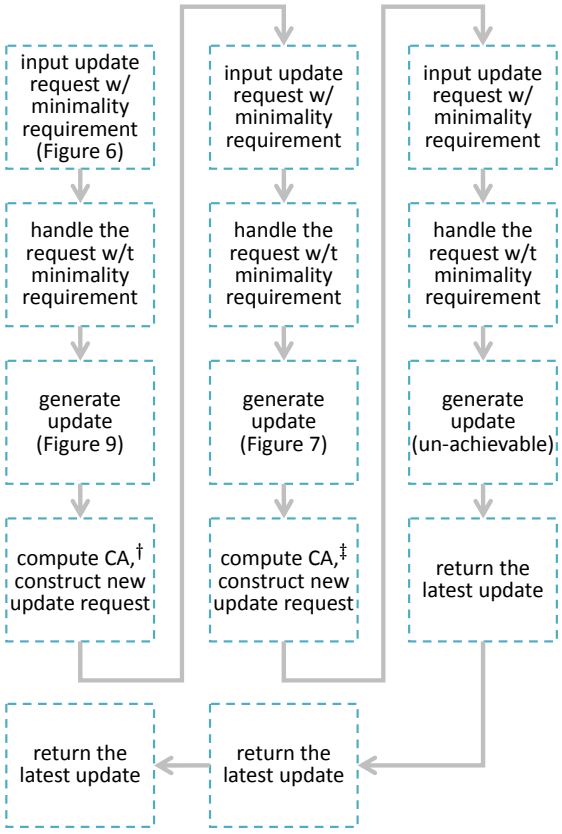
[5] The latter is accessible via `http://ww2.cs.mu.oz.au/~zhangd/roledata/`.



(a) $(r, p_1) \in CA$ (b) $(r, p_1) \in CA$

(c) $(r, p_1) \notin CA$ (d) $(r, p_1) \notin CA$

Figure 13: Examples of assignments in CA.



†$CA = \{(r_6, p_8), (r_2, p_8), (r_6, p_5), (r_6, p_6)\}$

‡$CA = \{(r_6, p_8), (r_6, p_5), (r_6, p_6)\}$

Figure 14: The recursive calling procedure.

```
1  update
2    make P = input available via T = γ.R
3    with
4      administrators all-administrators
5      user-permission constraints
6        (u1, no-less-than P1,
7                     no-more-than P1),
8        (u2, no-less-than P2,
9                     no-more-than P2),
10       ...
11     restricted-role constraints ∅;
12     role-hierarchy = γ.RH;
```

Figure 15: Experimental update request specification.

of permissions), $noUR$ (the *maximum* number of roles that a user may be assigned to), and $noRP$ (the *maximum* number of permissions that a role may be assigned to). γ's user-role relation (resp. γ's role-permission relation) is generated by associating a number k of roles (resp. permissions) with each user (resp. role), where k is randomly from $[1, noUR]$ (resp. $[1, noRP]$). Without otherwise stated, the parameters used for tests are "$noU = 2000$, $noR = 500$, $noP = 2000$, $noUR = 5$, $noRP = 150$, $noReqps = 200$" and the role hierarchy is empty. One or more parameters are made variable in each group of tests.

Update requests are parameterized by $noReqps$ (the number of requested permissions) and is generated by randomly choosing a number $noReqps$ of permissions from γ's permission set. We let $\mathcal{T}$ be γ's role set. Figure 15 shows the experimental update request, lines of which may be replaced in each group of tests and where P_i is the set of permissions that user u_i has prior to updating.

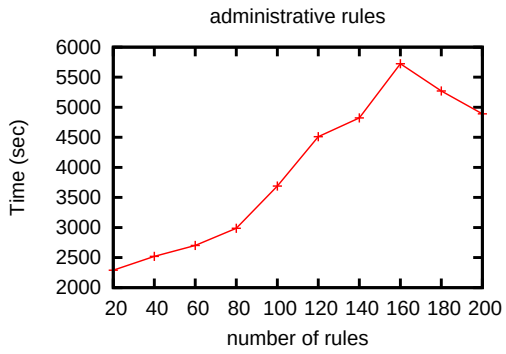
Results

Figure 16 shows the computing time required for each test. Since the data set is randomly created, for each configuration of parameters, we ran the test 5 times. The times in Figure 16 are averaged over the 5 runs.

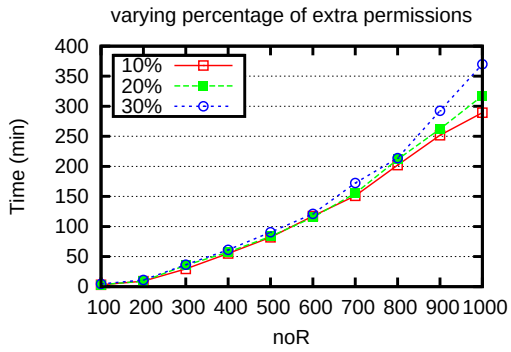
Administrative rules Figure 16a shows performance with respect to varying number of administrative rules ($noRules$). We let an administrator `admin` be a member of role ar and replace Line 4 of Figure 15 with the following.

```
administrators admin
```

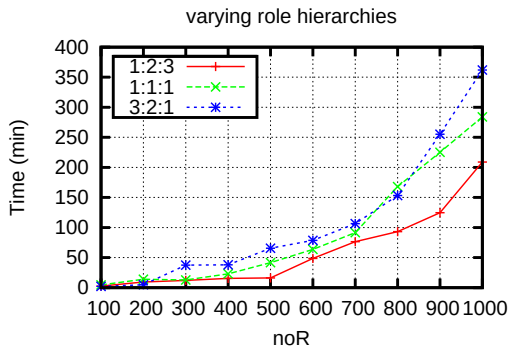
Each assignment rule "ar can assign p with r if p assigned to c" is constructed as follows: (1) denote the number of roles in c as $|c|$ and we let $|c| \in [1, 4]$, and (2) randomly choose roles in c. For revocation rules "ar



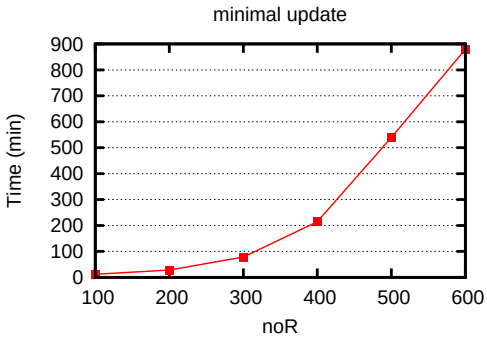
(a)



(b)



(c)



(d)

Figure 16: The computing time for evaluating update requests.

can revoke p from r", r is also randomly chosen. Note that we guarantee that rules have effects on the roles that might be changed. The speed of RoleUpdater is quite good as far as administrative rules are concerned. The reasons are two-fold: (1) The transformation into canonical requests is fast. (2) During the transformation, RoleUpdater only increases noU and noR but not $noUR$; fortunately, RoleUpdater is scalable to noU and noR [12].

Controllable effects To test RoleUpdater's performance with respect to controllable effects, we generated a ratio α of extra permissions. For each user u_i, we define the following constraint and substitute it for the corresponding line

$$(u_i, \texttt{no-less-than}\ P_{l,i},\ \texttt{no-more-than}\ P_{m,i})$$

where $P_{l,i} \subset P_i$ and $|P_{l,i}| = (1-\alpha) * |P_i|$, and $P_{m,i} \supset P_i$ and $|P_{m,i}| = (1+\alpha)*|P_i|$. Extra permissions were randomly chosen. Recall that P_i is the set of permissions that u_i has prior to updating. Figure 16b shows the results when α takes 10%, 20%, and 30%, respectively. It seems from this experiment that RoleUpdater is not sensitive to α, especially when $noR \leq 800$.

Role hierarchy Figure 16c gives the test results when the RBAC state involves role hierarchies. Role hierarchies were created in the following way. We created three sets of roles R_1, R_2, and R_3 such that $R_i \cap R_j = \emptyset$ for $i, j \in [1, 3]$ and $i \neq j$; we randomly created $\gamma.RH \subset (R_1 \times R_2) \cup (R_2 \times R_3)$ (where $\gamma.RH$ denotes γ's role hierarchy) such that each role may have only a number h of junior roles where $h \in [1, 3]$. This two-level layered role hierarchy is common in practical systems [19, 27, 31]. The x-axis is $|R_1| + |R_2| + |R_3|$. We tested three configurations by varying $|R_1| : |R_2| : |R_3|$. As the RBAC configuration needs to be flattened, $noUR$ is increased by 2 on average. This results in notable overhead. However, the time taken was sensitive to the structures of role hierarchies: almost all runs with $1 : 2 : 3$ were much faster than $3 : 2 : 1$. That is, the less senior roles there were, the faster RoleUpdater dealt with role hierarchies.

Minimal update To evaluate how well RoleUpdater treats minimal update, the minimality requirement is inserted into the specification in Figure 15. Figure 16d reports the computing time when minimal update is pursued. Note that the time was averaged over 5 achievable requests. When $noR = 600$, the computing time could be almost 18 times greater than the case without the minimal update requirement. This is because RoleUpdater has to compute a number of intermediate updates, with the number depending on $|CA|$. It would be interesting and useful to investigate how to reduce the number of intermediate steps.

In real-world large-scale RBAC systems, we expect that only a small portion of users have a number $noUR$ of roles and that the number of roles that are under specified administrators' control will be small. Hence, we conjecture RoleUpdater will be able to handle update requests in these RBAC systems, especially with the advances in model checking.

6 Conclusion

To update an access control system, we have presented a tool RoleUpdater, which accepts and answers high-level update requests. Experiments confirm the effectiveness and efficiency of RoleUpdater. We have reported the theoretical results of RoleUpdater in [12], including the computational complexity, the formal transformation into model checking problem, and the reductions. However, the full-fledged RoleUpdater is first reported here. RoleUpdater is still experimental and we regret that it is not yet available to the public.

There are several avenues for future work. RoleUpdater becomes awkward when dealing with administrative rules with negations, e.g., "ar can assign p if p assigned to r_1 but not r_2". The problem with more sophisticated administrative models, where negative conditions are allowed, deserves further investigation. In addition, *separation-of-duty* (SoD) policies are important in RBAC systems; however, enforcing SoD policies is difficult by itself [20]. The interaction between updating and SoD policies poses new challenges. On the other hand, if a series of update requests are issued, the final updated RBAC state may depend on the order of the requests. These composite requests may take place in distributed RBAC systems. We plan to investigate properties of composite update requests and extend RoleUpdater to address this problem.

7 Acknowledgment

We would like to thank Dr. Alva L. Couch for shepherding this paper and the anonymous reviewers for their helpful comments. This work is supported by National Natural Science Foundation of China under Grant 60873225, 60773191, 70771043, National High Technology Research and Development Program of China under Grant 2007AA01Z403, and Natural Science Foundation of Hubei Province under Grant 2009CDB298. This research is supported in part by an Australian Research Council Discovery Grant (DP0988396). This publication was made possible by a grant from the Qatar National Research Fund under its NPRP Grant No. 09-079-1-013.

Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the Qatar National Research Fund.

Author Biographies

Jinwei Hu is a PhD student in School of Computer Science and Technology at Huazhong University of Science and Technology, when submitting this paper. His current interests are the specification and analysis of access control policies.

Yan Zhang is a professor of School of Computing and Mathematics at University of Western Sydney. His research interests are in the areas of knowledge representation, logic, and model checking.

Ruixuan Li is an associate professor of School of Computer Science and Technology at Huazhong University of Science and Technology. His research interests are in the areas of distributed computing and distributed system security.

References

[1] AHMED, T., AND TRIPATHI, A. R. Static verification of security requirements in role based cscw systems. In *SACMAT'03*, pp. 196–203.

[2] AL-SHAER, E., MARRERO, W., EL-ATAWY, A., AND ELBADAWI, K. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *ICNP* (2009), pp. 123–132.

[3] ALIMI, R., WANG, Y., AND YANG, Y. R. Shadow configuration as a network management primitive. In *SIGCOMM* (2008), pp. 111–122.

[4] BAUER, L., GARRISS, S., AND REITER, M. K. Detecting and resolving policy misconfigurations in access-control systems. In *SACMAT'08*, pp. 185–194.

[5] CIMATTI, A., CLARKE, E., GIUNCHIGLIA, E., GIUNCHIGLIA, F., PISTORE, M., ROVERI, M., SEBASTIANI, R., AND TACCHELLA, A. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification (CAV 2002)* (2002), LNCS, pp. 359–364.

[6] CLARKE, E. M., GRUMBERG, O., AND PELED, D. A. *Model Checking*. MIT Press, 1999.

[7] COLANTONIO, A., PIETRO, R. D., OCELLO, A., AND VERDE, N. V. A formal framework to elicit roles with business meaning in rbac systems. In *SACMAT'09*, pp. 85–94.

[8] CRAMPTON, J. Understanding and developing role-based administrative models. In *CCS* (Alexandria, VA, USA, Nov. 2005), pp. 158 – 167. CCS'05.

[9] CRAMPTON, J., LEUNG, W., AND BEZNOSOV, K. The secondary and approximate authorization model and its application to bell-lapadula policies. In *ACM Symposium on Access Control Models and Technologies* (2006), pp. 111–120.

[10] ENE, A., HORNE, W. G., MILOSAVLJEVIC, N., RAO, P., SCHREIBER, R., AND TARJAN, R. E. Fast exact and heuristic methods for role minimization problems. In *SACMAT'08*, pp. 1–10.

[11] FERRAIOLO, D. F., SANDHU, R. S., GAVRILA, S. I., KUHN, D. R., AND CHANDRAMOULI, R. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Syst. Secur. 4*, 3 (2001), 224–274.

[12] HU, J., ZHANG, Y., LI, R., AND LU, Z. Role updating for assignments. In *Proceedings of 15th ACM symposium on access control models and technologies (SACMAT 2010)* (Pittsburgh, USA, June 9-11 2010), pp. 89–98.

[13] IRWIN, K., YU, T., AND WINSBOROUGH, W. H. Enforcing security properties in task-based systems. In *SACMAT'08*, pp. 41–50.

[14] JHA, S., LI, N., TRIPUNITARA, M., WANG, Q., AND WINSBOROUGH, W. Towards formal verification of role-based access control policies. *IEEE Trans. Dependable Secur. Comput. 5*, 4 (2008), 242–255.

[15] KARJOTH, G. The authorization service of tivoli policy director. In *ACSAC '01: Proceedings of the 17th Annual Computer Security Applications Conference* (Washington, DC, USA, 2001), IEEE Computer Society, p. 319.

[16] KATHI FISLER, SHRIRAM KRISHNAMURTHI, L. M., AND TSCHANTZ, M. Verification and change impact analysis of access-control policies. In *ICSE* (May 2005).

[17] KERN, A. Advanced features for enterprise-wide role-based access control. In *ACSAC '02: Proceedings of the 18th Annual Computer Security Applications Conference* (Washington, DC, USA, 2002), IEEE Computer Society, p. 333.

[18] KERN, A., KUHLMANN, M., SCHAAD, A., AND MOFFETT, J. D. Observations on the role life-cycle in the context of enterprise security management. In *SACMAT* (2002), pp. 43–51.

[19] KERN, A., SCHAAD, A., AND MOFFETT, J. D. An administration concept for the enterprise role-based access control model. In *SACMAT* (2003), pp. 3–11.

[20] LI, N., BIZRI, Z., AND TRIPUNITARA, M. V. On mutually-exclusive roles and separation of duty. In *CCS* (2004), pp. 42–51.

[21] LI, N., AND MAO, Z. Administration in role-based access control. In *ASIACCS* (2007), pp. 127–138.

[22] LI, N., MITCHELL, J. C., AND WINSBOROUGH, W. H. Beyond proof-of-compliance: security analysis in trust management. *J. ACM 52*, 3 (2005), 474–514.

[23] LI, N., AND TRIPUNITARA, M. V. Security analysis in role-based access control. In *SACMAT* (2004), pp. 126–135.

[24] LI, N., TRIPUNITARA, M. V., AND WANG, Q. Resiliency policies in access control. In *CCS* (2006), pp. 113–123.

[25] MCPHERSON, D. *Role-based access control for multi-tier applications using authorization manager: http://technet.microsoft.com/en-us/library/cc780256(WS.10).aspx*.

[26] MOLLOY, I., CHEN, H., LI, T., WANG, Q., LI, N., BERTINO, E., CALO, S. B., AND LOBO, J. Mining roles with semantic meanings. In *SACMAT* (2008), pp. 21–30.

[27] MOLLOY, I., LI, N., LI, T., MAO, Z., WANG, Q., AND LOBO, J. Evaluating role mining algorithms. In *SACMAT* (2009), pp. 95–104.

[28] MONDAL, S., SURAL, S., AND ATLURI, V. Towards formal security analysis of gtrbac using timed automata. In *SACMAT'09*, pp. 33–42.

[29] NI, Q., LOBO, J., CALO, S. B., ROHATGI, P., AND BERTINO, E. Automating role-based provisioning by learning from examples. In *SACMAT* (2009), pp. 75–84.

[30] OSBORN, S. L., SANDHU, R. S., AND MUNAWER, Q. Configuring role-based access control to enforce mandatory and discretionary access control policies. *ACM Trans. Inf. Syst. Secur. 3*, 2 (2000), 85–106.

[31] PARK, J. S., COSTELLO, K. P., NEVEN, T. M., AND DIOSOMITO, J. A. A composite rbac approach for large, complex organizations. In *SACMAT* (2004), pp. 163–172.

[32] RAY, I. Applying semantic knowledge to real-time update of access control policies. *IEEE Trans. Knowl. Data Eng. 17*, 6 (2005), 844–858.

[33] REITH, M., NIU, J., AND WINSBOROUGH, W. H. Toward practical analysis for trust management policy. In *ASIACCS '09*, ACM, pp. 310–321.

[34] SANDHU, R. S., BHAMIDIPATI, V., AND MUNAWER, Q. The ARBAC97 model for role-based administration of roles. *TISSEC 2*, 1 (1999), 105–135.

[35] SANDHU, R. S., COYNE, E. J., FEINSTEIN, H. L., AND YOUMAN, C. E. Role-based access control models. *IEEE Computer 29*, 2 (February 1996), 38–47.

[36] SCHAAD, A., LOTZ, V., AND SOHR, K. A model-checking approach to analysing organisational controls in a loan origination process. In *SACMAT'06*, pp. 139–149.

[37] SCHAAD, A., MOFFETT, J. D., AND JACOB, J. The role-based access control system of a european bank: a case study and discussion. In *SACMAT* (2001), pp. 3–9.

[38] SOHR, K., DROUINEAUD, M., AHN, G.-J., AND GOGOLLA, M. Analyzing and managing role-based access control policies. *Knowledge and Data Engineering, IEEE Transactions on 20*, 7 (July 2008), 924–939.

[39] STOLLER, S. D., YANG, P., RAMAKRISHNAN, C., AND GOFMAN, M. I. Efficient policy analysis for administrative role based access control. In *CCS'07*.

[40] VAIDYA, J., ATLURI, V., AND GUO, Q. The role mining problem: Finding a minimal descriptive set of roles. In *SACMAT* (2007), pp. 175–184.

[41] VAIDYA, J., ATLURI, V., AND WARNER, J. Roleminer: mining roles using subset enumeration. In *CCS* (2006), pp. 144–153.

[42] WEI, Q., CRAMPTON, J., BEZNOSOV, K., AND RIPEANU, M. Authorization recycling in rbac systems. In *SACMAT* (2008).

[43] WEI, Q., RIPEANU, M., AND BEZNOSOV, K. Cooperative secondary authorization recycling. In *HPDC* (2007).

[44] XU, W., SHEHAB, M., AND AHN, G.-J. Visualization based policy analysis: case study in selinux. In *SACMAT'08*, pp. 165–174.

[45] ZHANG, D., RAMAMOHANARAO, K., EBRINGER, T., AND YANN, T. Permission set mining: Discovering practical and useful roles. In *ACSAC* (2008), pp. 247–256.