Adventures in the Analysis of Access Control Policies

Anh Truong(✉)

Ho Chi Minh City University of Technology, VNU-HCM, Ho Chi Minh City, Vietnam
anhtt@hcmut.edu.vn

Abstract. Access Control is becoming increasingly important for today's ubiquitous systems which provide mechanism to prevent sensitive resources against unauthorized users. In access control models, the administration of access control policies is an important task that raises a crucial analysis problem: if a set of administrators can give a user an unauthorized access permission. In this paper, we consider the analysis problem in the context of the Administrative Role-Based Access Control (ARBAC), one of the most widespread administrative models. We describe how we design heuristics to enable an analysis tool, called ASASPXL, to scale up to handle large and complex ARBAC policies and a sequence of analysis problems. An extensive experimentation shows that the proposed heuristics play a key role in the success of the analysis tool over the state-of-the-art analysis tools.

Keywords: Access control · Security analysis · Automated verification · Model checking · Role-Based Access Control

1 Introduction

Modern information systems contain sensitive information and resources that need to be protected against unauthorized users who want to steal it. The most important mechanism to prevent this is Access Control [4] which is thus becoming increasingly important for today's ubiquitous systems. In general, access control systems protect the resources of the systems by controlling who has permission to access what objects/resources.

Today, one of the most widely adopted access control models in the real world is Role-Based Access Control (RBAC) model [12]. In general, RBAC access control policies specify which users can be assigned to roles which, in turn, are granted permissions to perform certain operations in the system. RBAC policies need to be evolved according to the rapidly changing environments and thus, it is demanded to have some mechanisms to control the modification of the policies. Administrative RBAC (ARBAC) [3] is the corresponding widely used administrative model for RBAC policies. In ARBAC, certain specific users, called administrators, are provided some permissions to execute operations, called administrative actions, to modify the RBAC policies. In fact, permissions to perform

© Springer Nature Switzerland AG 2019
T. K. Dang et al. (Eds.): FDSE 2019, LNCS 11814, pp. 467–482, 2019.
https://doi.org/10.1007/978-3-030-35653-8_31

administrative actions must be restricted since administrators can only be partially trusted. For instances, some of them may collude to, inadvertently or maliciously, modify the policies (by sequences of administrative actions) so that untrusted users can get sensitive permissions. Thus, automated analysis techniques taking into consideration the effect of all possible sequences of administrative actions to identify the safety issues, i.e. administrative actions generating policies by which a user can acquire permissions that may compromise some security goals, are needed.

Several automated analysis techniques (see, e.g., [1,5,10,15,16]) have been developed for solving the user-role reachability problem, an instance of the safety issues, in the ARBAC model. Recently, a tool called ASASPXL [11] has been shown to perform better than the state-of-the-art tools on sets of benchmark problems in [9,15]. The main advantage of the analysis technique inside ASASPXL over the state-of-the-art techniques is that the tool can solve the user-role reachability problem with respect to a finite but unknown number of users in the policies manipulated by the administrative actions. However, ASASPXL does not scale to solve problems in some recently proposed benchmarks in [16]. This is because the so-called state explosion problem has not been handled carefully and thus, prevent ASASPXL to tackle such benchmarks. Additionally, ASASPXL does not also scale to solve a sequence of reachability problems. The main reason is that the explored states during the previous analysis processes are not optimized.

In this paper, we study how to design heuristics to enable ASASPXL to analyze large and complex instances of user-role reachability problems and also to analyse the sequence of reachability problems more efficiently. The main idea is to try to alleviate the state explosion problem, which is well-known problem in model checking techniques, and reuse as many as possible the explored states, in the analysis of ARBAC policies. We also perform an exhaustive experiment to conduct the effectiveness of proposed heuristics and compare ASASPXL's performance with the state-of-the-art analysis tools.

The paper is organized as follows. Section 2 introduces the RBAC, ARBAC models, and the related analysis problem. Section 3 briefly introduces the framework to automatically analyse the infinite state transition systems, namely MCMT. The proposed heuristics to enable ASASPXL to scale to solve user-role reachability problem are described in Sect. 4. Section 5 summarizes our experiments and Sect. 6 concludes the paper.

2 Administrative Role-Based Access Control

In the *Role-Based Access Control (RBAC)* model [12], access decisions are based on the roles that individual users have as part of an organization. Permissions are grouped by role name and correspond to various uses of a resource. Roles can have overlapping responsibilities and privileges, i.e. users belonging to different roles may have common permissions. Thus, it would be inefficient to repeatedly specify common permissions for a certain set of roles. To overcome this problem, (so-called) role hierarchies reflect the natural structure of an enterprise and

make the specification of policies more compact by requiring that one role may implicitly include the permissions that are associated with another role.

Once RBAC policies are determined they need to be maintained according to the evolving needs of the organization. For flexibility and scalability, large systems usually require several administrators, and thus there is a need not only to have a consistent RBAC policy but also to ensure that the policy is only modified by the administrators who are allowed to do so. One of the most popular administrative frameworks is the Administrative RBAC (ARBAC) model [3] whose main insight is to use RBAC to control how RBAC policies may evolve through administrative actions that assign or revoke user memberships into roles. Since administrators can be only partially trusted, administration privileges must be limited to selected parts of the RBAC policies, called *administrative domains*. The ARBAC model defines administrative domains by using roles and RBAC itself to control how security officers can delegate (part of) their administrative permissions to trusted users. In this way, several administrators are able to modify the RBAC policy of a large system by following certain rules.

Formalization. Let U be a set of users, R a set of roles, and P a set of permissions. Users are associated to roles by a binary relation $UA \subseteq U \times R$ and roles are associated to permissions by another binary relation $PA \subseteq R \times P$. A role hierarchy is a partial order $\succeq$ on R, where $r_1 \succeq r_2$ means that r_1 is *more senior than* r_2 for $r_1, r_2 \in R$. A user u is a *member* of role r when $(u, r) \in UA$. A user u *has permission* p if there exists a role $r \in R$ such that $(p, r) \in PA$ and u is a member of r. A *RBAC policy* is a tuple $(U, R, P, UA, PA, \succeq)$.

Usually (see, e.g., [15]), administrators may only update the relation UA while PA and $\succeq$ are assumed constant. An administrative domain is specified by a *pre-condition*, i.e. a finite set of expressions of the forms r or $\overline{r}$ (for $r \in R$), called *role literals*. A user $u \in U$ *satisfies* a pre-condition C if, for each $\ell \in C$, u is a member of r when ℓ is r or u is not a member of r when ℓ is $\overline{r}$ for $r \in R$. Permission to assign users to roles is specified by a ternary relation *can_assign* containing tuples of the form (C_a, C, r) where C_a and C are pre-conditions, and r a role. Permission to revoke users from roles is specified by a binary relation *can_revoke* containing tuples of the form (C_a, r) where C_a is a pre-condition and r a role. In both cases, we say that C_a is the *administrative pre-condition*, C is a *(simple) pre-condition*, r is the *target role*, and a user u_a satisfying C_a is the *administrator*. When there exist users satisfying the administrative and the simple (if the case) pre-conditions of an administrative action, the action is *enabled*. The relation *can_revoke* is only binary because simple pre-conditions are useless when revoking roles (see, e.g., [15]).

The semantics of the administrative actions in $\psi := (can_assign, can_revoke)$ is given by the binary relation $\rightarrow_\psi$ defined as follows: $UA \rightarrow_\psi UA'$ iff there exist users u_a and u in U such that either (i) there exists $(C_a, C, r) \in can_assign$, u_a satisfies C_a, u satisfies C (i.e. (C_a, C, r) is enabled), and $UA' = UA \cup \{(u, r)\}$ or (ii) there exists $(C_a, r) \in can_revoke$, u_a satisfies C_a (i.e. (C_a, r) is enabled), and $UA' = UA \setminus \{(u, r)\}$. A *run* of the administrative actions in $\psi :=$

(can_assign, can_revoke) is a possibly infinite sequence $UA_0, UA_1, ..., UA_n, ...$ such that $UA_i \rightarrow_\psi UA_{i+1}$ for every $i \geq 0$.

A pair (u_g, R_g) is called a *(RBAC) goal* for $u_g \in U$ and R_g a finite set of roles. The cardinality $|R_g|$ of R_g is the *size* of the goal. Given an initial RBAC policy UA_0, a goal (u_g, R_g), and administrative actions $\psi = (can_assign, can_revoke)$; (an instance of) the *user-role reachability problem* consists of establishing if there exists a finite sequence $UA_0, UA_1, ..., UA_n$ (for $n \geq 0$) where (i) $UA_i \rightarrow_\psi UA_{i+1}$ for each $i = 0, ..., n-1$ and (ii) in the policy UA_n we have that u_g is a member (explicit or implicit) of each role in R_g.

The definition of the user-role reachability problem considered here is the same of that in [15]. In the rest of the paper, we focus on user-role reachability problems where U and R are finite, P plays no role, and $\succeq$ is ignored because it can be eliminated by a pre-processing [13]. Thus, a RBAC policy is a tuple (U, R, UA) or simply UA when U and R are clear from the context.

3 Framework to Analyse Infinite State Transition Systems

MCMT [7] attempts to solve reachability problems for a certain class of infinite state systems whose state variables are arrays, that can be seen as functions mapping indexes to elements. Such transition systems can be used as suitable abstractions of parametrised protocols, sequential programs manipulating arrays, etc. The main idea underlying MCMT is to use a backward reachability procedure that repeatedly computes pre-images of the set of *goal* states, that is usually obtained by complementing a certain safety property that the system should satisfy. The set of backward reachable states of the system is obtained by taking the union of such pre-images. At each iteration of the procedure, it is checked whether the intersection with the initial set of states is non-empty (*safety* test), reporting the *unsafety* of the system, i.e. there exists a (finite) sequence of transitions that leads the system from an initial state to one satisfying the goal. Otherwise, when the intersection is empty, it is checked if the set of backward reachable states is contained in the set computed at the previous iteration (*fix-point* test), reporting the *safety* of the system, i.e. no (finite) sequence of transitions leads the system from an initial state to one satisfying the goal. The peculiarity of MCMT is that sets of states and transitions are represented by first-order formulae so that the computation of pre-images boils down to logical manipulations and the safety and fix-point tests are reduced to satisfiability checks of first-order formulae. The resulting satisfiability problems are efficiently solved by state-of-the-art tools, called Satisfiability Modulo Theories (SMT) solvers.

MCMT was successfully used for the verification of several infinite state systems [6] and features some interesting heuristics to automatically synthesize invariants that can be used to prune the search space of the system that become also available to the user for a deeper understanding of the structure of the system. For more details on MCMT, the interested reader is pointed to [7]. Since

model checking techniques have been found quite useful for the analysis of authorization policies, it would be interesting to apply MCMT in this context. Unfortunately, as observed in [1], there some practical problems to do this. Two of the most important ones are the following:

P1 MCMT permits only mono-dimensional arrays while RBAC policies seem to require at least bi-dimensional arrays to encode the characteristic function of the binary relation UA.

P2 MCMT restricts the number of existentially quantified variables in formulae representing transitions to (at most) two while can_assign and can_revoke actions usually require more of such variables.

These and other problems have lead us to build a new tool, called ASASP [1], based on the same ideas of MCMT but specifically tuned for the symbolic analysis of authorization policies. We were successfull in doing this as ASASP shows a better scalability than RBAC-PAT [8] on a set of synthetic user-role reachability problems introduced in [15]. However, our aim in building ASASP was to reach a good trade-off between efficiency and expressivity. In fact, we were able to extend the classical ARBAC policies with attributes and to efficiently solve the associated user-role reachability problems in [2]. Key to the expressivity of ASASP input language is the capability of handling arbitrary (possibly infinite) set of roles—such as those depending on parameters, see, e.g., [14]—and several existentially quantified variables in formulae representing transitions that are fundamental to express conditions involving role hierarchies and attribute values [2].

Almost at the same time of [1], a new tool, called MOHAWK [9], has been proposed for the analysis of ARBAC policies especially tailored to error-finding rather than verification as it is the case of both RBAC-PAT and ASASP. In [9], it is shown that MOHAWK outperforms RBAC-PAT on the problems in [15] and on a new set of synthetic, much larger problems. Indeed, we tried ASASP on new problems of MOHAWK and, rather disappointingly, it was not able to scale up and handle large problem instances. After a careful analysis of the behavior of ASASP, we made the following crucial observations.

O1 Since the set R of roles is finite, it is not necessary to use the binary relation UA to record user-role assignments. It is sufficient to replace it with a finite collection of sets, one per role. Formally, let $R = \{r_1, ..., r_n\}$ for $n \geq 1$, define $U_{r_i} = \{u | (u, r_i) \in UA\}$ for $i = 1, ..., n$. Straightforward modifications to the definition of $\rightarrow_\psi$ (for ψ a pair of relations can_assign and can_revoke), given in Sect. 2, allows one to replace UA with the U_{r_i}'s.

O2 Since the role-hierarchy can be pre-processed before attempting to solve a user-role reachability problem (see, e.g., [13]), the definition of $\rightarrow_\psi$, for a given tuple in can_assign or can_revoke, existentially quantifies over two users, the administrator and the user to which the administrative action is going to be applied.

On the one hand, the two observations suggest that ASASP expressivity is too much for modelling large synthetic problems (as those in [9]) that are

characterized by simple pre-conditions in which even role hierarchies are not used. Since it is well-known that expressivity and efficiency are opposing forces when developing automated analysis techniques, we believe this to be one of the main reasons for the poor results obtained by ASASP on the problems in [9]. On the other hand, **O1** and **O2** pave the way to the use of MCMT on this kind of synthetic problems as they remove the two problems discussed above. In particular, we can represent the characteristic function of a U_{r_i} by using a mono-dimensional array a_{r_i} mapping users to Booleans; thus overcoming **P1**. Also **P2** is no more a problem as MCMT supports the definition of transitions by formulae containing (at most) two existentially quantified variables.

We illustrate the encoding in MCMT of user-role reachability problem by means of an excerpt of an example from [15].

Example 1. Let $U = \{u_1, u_2\}$, $R = \{r_a, r_1, ..., r_8\}$, and $UA := \{(u_1, r_1), (u_1, r_4), (u_1, r_7), (u_2, r_a)\}$ and the tuple $(\{r_a\}, \{r_1\}, r_2)$ is in can_assign whereas the tuple $(\{r_a\}, r_1)$ is in can_revoke. The goal of the user-role reachability problem is $\{(u_1, r_6)\}$.

To formalize this problem in MCMT, we introduce one array per role u_r for $r \in R$ mapping a user in U to a Boolean that encodes the characteristic function of the set U_r, as defined above. The initial relation UA can be expressed as follows:

$$\forall x. \begin{bmatrix} u_{r_1}(x) \leftrightarrow x = u_1 \wedge u_{r_4}(x) \leftrightarrow x = u_1 \wedge u_{r_7}(x) \leftrightarrow x = u_1 \wedge u_{r_a}(x) \leftrightarrow u = u_2 \wedge \\ \neg u_{r_2}(x) \wedge \neg u_{r_3}(x) \wedge \neg u_{r_5}(x) \wedge \neg u_{r_6}(x) \end{bmatrix}$$

The tuple $(\{r_a\}, \{r_1\}, r_2)$ in can_assign is formalized as

$$\exists x_a, x. \left[u_{r_a}(x_a) \wedge u_{r_1}(x) \wedge \forall y.(u'_{r_2}(y) \leftrightarrow (y = x \vee u_{r_2}(x))) \right]$$

and $(\{r_a\}, r_1)$ in can_revoke as

$$\exists x_a, x. \left[u_{r_a}(x_a) \wedge u_{r_1}(x) \wedge \forall y.(u'_{r_1}(y) \leftrightarrow (y \neq x \wedge u_{r_2}(x))) \right]$$

where u_r and u'_r indicates the value of U_r immediately before and after, respectively, of the execution of the administrative action and we have omitted identical updates, i.e. a conjunct $\forall y.(u'_r(y) \leftrightarrow u'_r(y))$ for each r not mentioned in the formula. Finally, the goal can be represented as $\exists x.u_{r_6}(x) \wedge x = u_1$. □

We built a translator to create MCMT reachability problems out of user-role reachability problems and MCMT on the problems resulting from the translations of those in [9]: the results were still disappointing in terms of scalability. After analysing the behavior of MCMT, we identified another important source of complexity: the number of transitions or, equivalently, the number of tuples in can_assign and can_revoke is so large that the heuristics implemented in MCMT to control the state space explosion problem are not enough. This is due to the fact that the problems that MCMT is successful in solving comprise tens of transitions, each one involving complex conditions and updates of data structures

and updates. Instead, the transitions obtained from the translation of user-role reachability problems in [9] are hundreds or thousands (up to 80,000) involving, as discussed above and shown in the example, simple conditions and updates. In the following section, we describe how this difficulty can be tamed by designing a suitable wrapper around MCMT in order to generate a sequence of user-role reachability problems with an increasing number of transitions. We will also explain how the solution of the problem P_i (for $i > 0$) in the sequence can be speeded up by caching the invariants synthesized by MCMT in the runs to solve $P_0, ..., P_{i-1}$. The synthesized invariants encode "structural" properties of the user-assignment relation UA; e.g., 'a given role can be assigned to no user' or 'if a user is not assigned to a certain set of roles then he/she will not be assigned to the roles in another set.' Clearly, this kind of properties is using when solving problem P_i. For example, if we know from a previous run of MCMT that no user can be assigned to role r_7 and the goal of the user-role reachability problem is $(u, \{r_1, r_7, r_9\})$, we can immediately conclude that the goal is unreachable.

4 asasp 2.1: New Clothes for Analyzing ARBAC Policies

In this section, we describe the architecture of our technique. ASASP 2.1 receives an ARBAC policy and a safety query as its input. In case it finds a sequence of transitions (including can_assign, can_revoke rules) that leads the RBAC system to the query state, the result UNSAFE is reported. Otherwise, ASASP 2.1 reports that the system is SAFE. Figure 1 illustrates the architecture of ASASP 2.1.

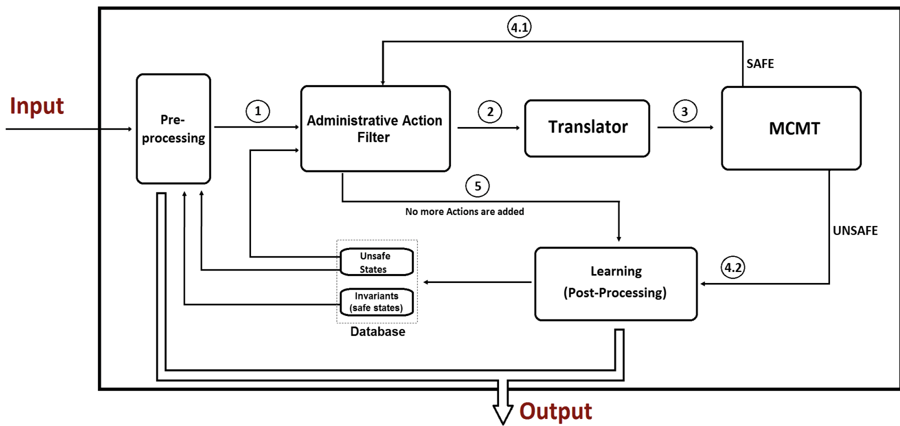

Fig. 1. ASASP 2.1 architecture

In general, the policy and the query are verified through the following steps:

Pre-processing: The query is analyzed to find the answer in the database. If an answer is found, the result is reported without running the remains of the technique.

Administrative Action Filter: The policy and the query are analyzed to filter the related transitions and roles for current verification step.

Model checking: The technique verifies the filtered policy. The model checker MCMT [7] is used in the verification.

Learning: The learning component analyzes the verification result before outputting them.

We discuss the details of these steps here in the remains of this section. Because of the separate administration restriction [..], we do not care the administrative roles in ASASP 2.1.

4.1 Pre-processing

In general, the time spent in running the model checker holds the most important part in the amount of time taken for the verification process of automatic analysis techniques. Therefore, if we reduce the time spent in running the model checker, the total time will also be reduced significantly. With our technique, if the answer is found in this pre-processing step, ASASP 2.1 returns the result without calling other components, including MCMT, and hence the time for verification process is reduced.

In order to speed up the verification processes, the states which have been visited in the previous verifications are saved to a database. When a new query is issued, ASASP 2.1 first checks the database to find the answer for the query. If it finds the answer, the result is reported. Otherwise, the remains of the technique are called. For example: in the database, we have a state in which "User x is assigned both role A and role B." This state is marked as an unsafe state. It means that the state is reachable from the initial state of the policy. When a new query such as "User x can be assigned role A or not" is issued, ASASP 2.1 finds in the database and returns `UNSAFE`. This means that there exists a sequence of transitions which lead the system from the initial state to the query state. Another example is that a state in which "User y is assigned both role B and role C" is marked as a safe state in the database. A safe state means that it is unreachable from the initial state. If ASASP 2.1 receives a query such as "User y can be assigned role A, role B and role C at the same time or not", it returns the result SAFE quickly.

4.2 Administrative Action Filter

For a complex ARBAC policy with many users, roles and transitions, it is not efficient to analyze the entire policy. Many previous techniques fail to verify these policies. The reason is that the model checkers used in these techniques require large resources, including memory and processor, to verify the entire policies, meanwhile, hardware technologies currently can not satisfy these requirements. Therefore, there is a need to filter these policies before sending to the model checkers for verifying. The model checkers then can verify these filtered policies in a reasonable time.

In this analysis step, our technique removes all the redundant transitions, users and roles which do not relate to the current verification process. Moreover, the technique also reuses the states which have been verified in the previous verification processes to speed up the current verification.

The idea to filter the complex policies comes from the one in [9]. In this technique, the authors refine the complex policies according to the related-by-assignment relationship between roles. At each refinement step, the technique selects the users, roles and transitions which relate to the roles in the current priority degree. The refined policy is then verified by a model checker. In our technique, we do not define the priority of each role, we just care which roles is related to the current analysis step. The roles related to the first analysis step are the roles in the safety query. We call the set of the roles related to the current analysis step as a current related role set. In the current analysis step, ASASP 2.1 analysis the policy to extract the transitions related to current analysis step. A transition is related to the current analysis step if its goal contains at least one role in the current related role set. For example, in the first analysis step of the policy in Fig. 2, the current related role set contains role Manager. The set of the transitions related to the first analysis step contains $(Admin, QC \wedge IT, Manager)$, $(Admin, Tester, Manager)$, and $(Admin, Manager)$.

Roles: Admin, Manager, Developer, QA, QC, Tester, IT
UA: (Alice, Admin) (Bob, Developer) (Bob, QA)
Can_revoke rules:
(Admin, Manager)
(Admin, Developer)
(Admin, Tester)
Can_assign rules:
(Admin, Developer ^ QA, Tester ^ QC)
(Admin, QC ^ IT, Manager)
(Admin, Tester, Manager)
(Admin, True, Developer)
Safety query:
Bob, Manager

Fig. 2. An example of ARBAC policy

Moreover, ASASP 2.1 also considers removing the transitions which are not able to be executed in the current verification step. The idea comes from the observation that a transition may be executed if the preconditions of the transition are satisfied (e.g., the transition $(Admin, QC \wedge IT, Manager)$ may be executed if there is at least one user who has both roles QC and IT). In our technique, we use the idea with less constraint as follows: a transition may be executed (and therefore this transition will be added to the current filtered policy) if each role in its precondition is:

- In the initial state or
- In the current related role set and there is at least a transition in the original policy whose goal contains the role.

For example in Fig. 2: we assume that the current related role set includes Manager and Tester. The transition ($Admin$, $Developer \wedge QA$, $Tester \wedge QC$) is a transition related to the current analysis step because its goal $Tester \wedge QC$ contains the role Tester. Furthermore, all roles in its precondition $Developer \wedge QA$ are in the initial state. This transition therefore is added to the current filtered policy. The transition ($Admin$, $QC \wedge IT$, $Manager$) is also a transition related to the current analysis step. However, it is not added to the current filtered policy because the roles in its precondition are not in the current related role set and there is no transition whose goal contains the role IT.

After running the current analysis step, ASASP 2.1 will calculate the set of the roles related to next analysis step. The next related role set will contain all roles in the current related role set. Moreover, for each assignment transition (can_assign rule) in the set of the transitions related to the current analysis step, the roles in its precondition will also be added to the next related role set if: for each role in the precondition, there is at least a transition in the original policy whose goal contains the role. For example in Fig. 2, we assume that the current related role set includes Manager. The set of transitions related to the current analysis step is $\langle(Admin, QC \wedge IT, Manager), (Admin, Tester, Manager), (Admin, Manager)\rangle$. After running the analysis step, the next related role set will be $\langle Manager, Tester\rangle$. The roles QC and IT of the transition ($Admin$, $QC \wedge IT$, $Manager$) are not added to the next related role set because there is no transition in the original policy whose goal contains the role IT.

One of the key points in our technique is to reuse the states which have been visited in the previous verifications. Normally, the users tend to query the policy many times with different safety queries, hence the reuse of visited states will reduce the time taken by the model checker significantly. In our technique, the results of the previous verifications were analyzed by the learning component after the model checker found the answer for the previous queries. All states extracted from the results and their statuses (safe or unsafe) were saved to the database. This information is then added to each analysis step of the current verification. Intuitively, each unsafe state can be considered as an initial state of the filtered policy of the current analysis step. If the state in the query can be reachable from one of these initial states, it is also unsafe. However, this implementation requires that the model checker supports the system with multiple initial states. In the current version of ASASP 2.1, we use another way to add the visited states to the current filtered policy. The idea is that:

- For each unsafe state, a new transition which "connects" the initial state to the unsafe state will be added to the filtered policy.
- For each safe state, we remove from the filtered policy the transitions whose preconditions contain the roles in the safe state.

For example in Fig. 3(a), from the previous verifications, state B, C and D are un-safe states while state G is a safe state. At the current verification, two new transitions are added and one transition is removed. The current policy is shown in Fig. 3(b). If we want to check state E, we just go backward to state C and

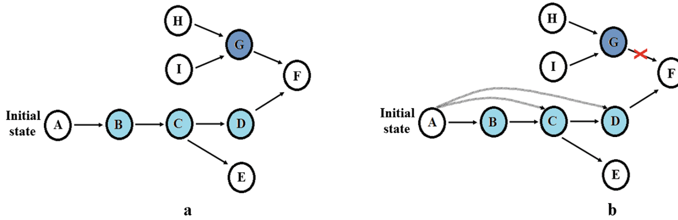

Fig. 3. Reusing visited states

then can reach the initial state. If we want to check state F, we do not care the states G, H and I.

After executing the current analysis step, the filtered policy is transfer to the verification step in order to determine whether the query state is safe or not. If the state is unsafe, ASASP 2.1 will call the learning step to analyze the results and then terminate the verification process. Otherwise, ASASP 2.1 will call the next analysis step to refilter the policy. After the next analysis step, if some transitions can be added to the current filtered policy, the verification step then will be called to verify the new filtered policy. If the current filtered policy is not changed, the results of the verification step will be transfer to the learning step and the verification process is terminated after finishing the learning step.

4.3 Model Checking

In this step, ASASP 2.1 translates the filtered policy and the query to an infinite state based system in the MCMT code. It then invokes MCMT to verify the system. If MCMT determines that the query is satisfied in the filtered policy, it returns the result UNSAFE as well as a log file. The log file contains a sequence of transitions that leads the system from the initial state to the query state. Similarly, the result SAFE and a log file are also returned if the model checker determines that the query is not satisfied in the filtered policy. In this case, the log file provides a list of invariants which are tracked during the verification. In case MCMT returns the result UNSAFE, the log file and the result are transferred to the learning component. In this component, the log file is analyzed in order to extract the unsafe states. No additional verification step is executed and the result is outputted after being analyzed. Conversely, ASASP 2.1 calls again the analysis step to refilter the policy with the next related role set and an additional verification step may then be executed. After the analysis step, if some additional transitions are added to the previous filtered policy, ASASP 2.1 will call the verification step again. Otherwise, it transfers the result and the log file of the previous verification step to the learning component. In this case, the learning component analyzes the log file to get the safe states and then outputs the result SAFE.

4.4 Learning

After verifying the policy, our technique will analyze the output of the verification step and save the states which have been visited during the verification. According to the result of the verification step, the learning component will process as follows:

– If the result UNSAFE is returned, the learning component analyzes the sequence of transitions which leads the RBAC system from the initial state to the query state. The states which are extracted from the analysis of these transitions then will be saved as unsafe states.
– If the result is SAFE, the learning component will extract invariants which have been tracked during the verification step. These invariants then will be saved as safe states.

In the next section, we provide an example to describe ASASP 2.1's operations in detail.

4.5 Worked Out Example

We illustrate ASASP 2.1's operations using the example in Fig. 2. The safety query in this example asks whether Bob can be assigned to the role Manager or not. Table 1 describes the status of the analysis component during the verification: (RS: Current related role set; RT: Current set of related transitions; NRS: Next related role set; FT: the set of transitions in the current filtered policy). After the analysis step 1, MCMT returns the result SAFE. Therefore, a new analysis step is executed. When MCMT verifies the filtered policy in the analysis step 2, the result UNSAFE is returned. ASASP 2.1 then calls the learning component to analyze the output of the verification step 2. The query state can be reached by executing the following sequence of transitions:

– *Bob* is assigned to *Tester* and *QC* by the transition $(Admin, Developer \wedge QA, Tester \wedge QC)$.
– *Bob* is then assigned to *Manager* by the transition $(Admin, Tester, Manager)$.

Table 1. The status of the analysis component during verification

Step	RS	RT	NRS	FT	Result
1	*Manager*	$(Admin, QC \wedge IT, Manager)$ $(Admin, Tester, Manager)$ $(Admin, Manager)$	*Manager* *Tester*	$(Admin, Manager)$	SAFE
2	*Manager* *Tester*	$(Admin, Manager)$ $(Admin, Tester)$ $(Admin, Developer \wedge QA, Tester \wedge QC)$ $(Admin, QC \wedge IT, Manager)$ $(Admin, Tester, Manager)$	*Manager* *Tester*	$(Admin, Manager)$ $(Admin, Tester)$ $(Admin, Developer \wedge QA, Tester \wedge QC)$ $(Admin, Tester, Manager)$	UNSAFE

The learning component finds and saves the unsafe state (Bob, $Developer \wedge QA \wedge Tester \wedge QC \wedge Manager$) to the database. The state means that the user *Bob* has roles *Developer*, *QA*, *Tester*, *QC* and *Manager* at the same time. At the next verification, if the query asks whether *Bob* can be assigned to the role *QC* or not, ASASP 2.1 returns the result UNSAFE without running the model checker.

5 Experimental Evaluation

In this section, we show the evaluation we conducted in order to evaluate the effectiveness of our technique. In [9], the authors compared MOHAWK to other state-of-the-art verification tools for ARBAC policies such as symbolic model checking, bounded model checking and RBAC-PAT [8]. They also demonstrated that MOHAWK is more efficient than other tools. In this paper, instead of comparing our tool to the symbolic model checking, the bounded model checking or RBAC-PAT, we just compare ASASP to MOHAWK and show that our tool analyzes ARBAC policies better than MOHAWK.

We use the dataset which was used to compare MOHAWK to other verification tools as shown in [9]. The dataset includes three test suites. The first one (called as Test suite 1) contains policies with positive conjunctive can_assign rules and non-empty can_revoke rules. The second one (called as Test suite 2) includes policies with mixed conjunctive can_assign rules and empty can_revoke rules while the last one (Test suite 3) contains policies with mixed conjunctive can_assign rules and non-empty can_revoke rules. For each policy in the test suites, we call ASASP and MOHAWK 5 times with different safety queries and measure the average time taken for the verification process. All experiments were performed on an Intel Core 2 Duo T6600 (2.2 GHz) CPU, 2 GB Ram and Ubuntu 11.10.

Table 2 describes the experimental results.

MOHAWK takes two stages to verify an ARBAC policy. In the first stage, it slices the original policy and then, in the second stage, the sliced policy is verified by the refinement steps. We consider that the time taken for the verification process is the sum of time consumed by these stages. The table shows that our tool answers the safety queries faster than MOHAWK in most cases. In other word, these experimental results demonstrate the effectiveness of our technique.

We also test our technique in verifying a sequence of safety queries. It means that ASASP receives a set of safety queries, it verifies the first query and then the second query, and so on until the last query. We perform the verification of an ARBAC policy with a set of 8 different safety queries. Figure 4 shows that the time which ASASP takes for verifying n-th query is smaller than the one of the previous verifications. The reason is that ASASP reuses the states which have been visited in the previous verifications and hence, the time taken for the current verification may be reduced.

Table 2. Experimental results for single queries

Test suite	Number of roles, rules	Time for verification			Variance	
		Mohawk (Slicing time + Refinement time)		ASASP 2.1	Mohawk	ASASP 2.1
Test suite 1	3, 15	**0.42**	(0.17 + 0.25)	0.12	0.00034	0.00126
	5, 25	**0.50**	(0.20 + 0.30)	0.22	0.00104	0.02188
	20, 100	**0.60**	(0.28 + 0.32)	0.11	0.00048	0.00314
	40, 200	**0.94**	(0.39 + 0.55)	0.10	0.19242	0.00294
	200, 1000	**2.65**	(1.25 + 1.40)	0.18	0.7027	0.02758
	500, 2500	**4.87**	(2.27 + 2.60)	0.43	7.0337	0.29594
	4000, 20000	**16.90**	(11.41 + 5.49)	1.64	1.26694	0.11166
	20000, 80000	**71.56**	(44.70 + 26.86)	24.17	7.56264	0.27724
	30000, 120000	**195.54**	(119.39 + 76.15)	59.08	66.4833	0.38058
	40000, 200000	**455.14**	(263.82 + 191.32)	109.07	32.35406	2.42496
	80000, 400000	**2786.33**	(1600.22 + 1186.11)	398.63	1251.832	0.51542
Test suite 2	3, 15	**0.40**	(0.16 + 0.24)	0.12	0.00046	0.00204
	5, 25	**0.50**	(0.19 + 0.31)	0.21	0.0019	0.02012
	20, 100	**0.54**	(0.25 + 0.29)	0.10	0.00036	0.00242
	40, 200	**1.21**	(0.37 + 0.84)	0.10	1.07136	0.00108
	200, 1000	**2.54**	(1.24 + 1.30)	0.14	0.6452	0.01008
	500, 2500	**5.02**	(2.29 + 2.73)	0.43	5.91882	0.32836
	4000, 20000	**14.33**	(9.65 + 4.68)	1.48	0.53058	0.06206
	20000, 80000	**74.32**	(45.35 + 28.97)	24.99	13.9347	0.0716
	30000, 120000	**194.85**	(115.58 + 79.27)	57.09	42.39056	0.18292
	40000, 200000	**470.89**	(262.39 + 208.50)	98.49	585.6608	0.26196
	80000, 400000	**2753.12**	(1589.97 + 1163.15)	360.96	1493.596	3.19596
Test suite 3	3, 15	**0.41**	(0.17 + 0.24)	0.09	0.00012	0.00078
	5, 25	**0.47**	(0.19 + 0.28)	0.08	0.00164	0.0001
	20, 100	**0.77**	(0.29 + 0.48)	0.54	0.0771	0.08822
	40, 200	**0.77**	(0.38 + 0.39)	0.37	0.0012	0.00468
	200, 1000	**5.93**	(1.53 + 4.4)	1.51	47.2814	0.20348
	500, 2500	**3.78**	(2.15 + 1.73)	1.12	0.05662	0.00298
	4000, 20000	**14.05**	(9.96 + 4.09)	11.13	0.09255	0.317425
	20000, 80000	**80.61**	(48.64 + 31.97)	27.25	23.98093	2.974775
	30000, 120000	**259.15**	(148.35 + 110.80)	97.55	325.5216	6343.912
	40000, 200000	**604.17**	(346.10 + 258.07)	110.65	1247.141	110.9948
	80000, 400000	**3477.19**	(1951.41 + 1525.78)	402.22	2776.703	0.50856

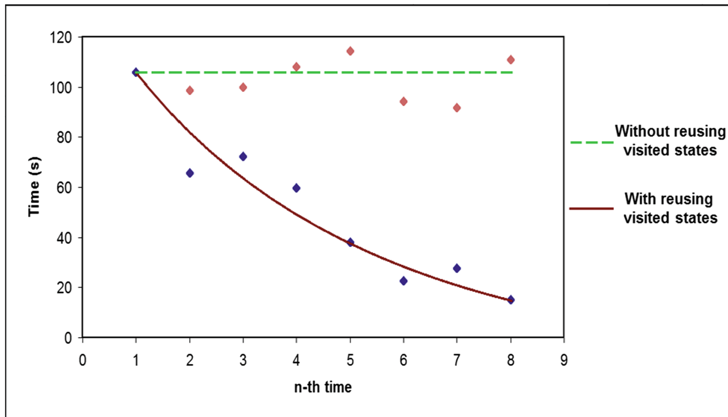

Fig. 4. Experimental results for a sequence of queries

6 Conclusions, Related and Future Work

We have presented techniques to enable the MCMT framework to solve instances of user-role reachability problem. We have also designed a set of heuristics that help our analysis techniques to be more scalable. The main idea is to reduce as much as possible the number of administrative actions in the original problem and reuse the visited states of previous analysis processes. We have shown that the proposed techniques do not miss errors in buggy policies and performs significantly better than MOHAWK on the larger problem instances in [9]. As future work, we plan to consider the combination of backward and forward reachability procedure to speed up the analysis of the model checker.

Acknowledgements. This work was funded by Vietnam National University-Ho Chi Minh City under the research project C2018-20-10.

References

1. Alberti, F., Armando, A., Ranise, S.: ASASP: automated symbolic analysis of security policies. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS (LNAI), vol. 6803, pp. 26–33. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22438-6_4
2. Alberti, F., Armando, A., Ranise, S.: Efficient symbolic automated analysis of administrative role based access control policies. In: ASIACCS. ACM Pr. (2011)
3. Crampton, J.: Understanding and developing role-based administrative models. In: Proceedings 12th ACM Conference on Computer and Communication Security (CCS), pp. 158–167. ACM Press (2005)
4. De Capitani di Vimercati, S., Foresti, S., Jajodia, S., Samarati, P.: Access control policies and languages. Int. J. Comput. Sci. Eng. (IJCSE), **3**(2), 94–102 (2007)

5. Ferrara, A.L., Madhusudan, P., Nguyen, T.L., Parlato, G.: Vac - verifier of administrative role-based access control policies. In: Biere, A., Bloem, R. (eds.) CAV 2014. LNCS, vol. 8559, pp. 184–191. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08867-9_12
6. Ghilardi, S., Ranise, S.: Backward reachability of array-based systems by SMT solving: termination and invariant synthesis. LMCS **6**(4) (2010)
7. Ghilardi, S., Ranise, S.: MCMT: a model checker modulo theories. In: Giesl, J., Hähnle, R. (eds.) IJCAR 2010. LNCS (LNAI), vol. 6173, pp. 22–29. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14203-1_3
8. Gofman, M.I., Luo, R., Solomon, A.C., Zhang, Y., Yang, P., Stoller, S.D.: RBAC-PAT: a policy analysis tool for role based access control. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 46–49. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-00768-2_4
9. Jayaraman, K., Ganesh, V., Tripunitara, M., Rinard, M., Chapin, S.: Automatic error finding for access-control policies. In: Proceedings of the ACM Conference on Computer and Communications Security (CCS). ACM (2011
10. Li, N., Tripunitara, M.V.: Security analysis in role-based access control. ACM Trans. Inf. Syst. Secur. (TISSEC) **9**(4), 391–420 (2006)
11. Ranise, S., Truong, A., Armando, A.: Boosting model checking to analyse large ARBAC policies. In: Jøsang, A., Samarati, P., Petrocchi, M. (eds.) STM 2012. LNCS, vol. 7783, pp. 273–288. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38004-4_18
12. Sandhu, R., Coyne, E., Feinstein, H., Youmann, C.: Role-based access control models. IEEE Comput. **2**(29), 38–47 (1996)
13. Sasturkar, A., Yang, P., Stoller, S.D., Ramakrishnan, C.R.: Policy analysis for administrative role based access control. In: Proceedings of the 19th Computer Security Foundations (CSF) Workshop. IEEE Computer Society Press, July 2006
14. Stoller, S.D., Yang, P., Gofman, M.I., Ramakrishnan, C.R.: Symbolic reachability analysis for parameterized administrative role based access control. In: Proceedings of SACMAT 2009, pp. 445–454 (2007)
15. Stoller, S.D., Yang, P., Ramakrishnan, C.R., Gofman, M.I.: Efficient policy analysis for administrative role based access control. In: Proceedings of the 14th Conference on Computer and Communications Security (CCS). ACM Press (2007)
16. Yang, P., Gofman, M.L., Stoller, S., Yang, Z.: Policy analysis for administrative role based access control without separate administration. J. Comput. Secur. (2014)