Incremental Analysis of Evolving Administrative Role Based Access Control Policies

Silvio Ranise, Anh Truong

Incremental Analysis of Evolving Administrative Role Based Access Control Policies

Silvio Ranise[1] and Anh Truong[1,2]

[1] Security and Trust Unit, FBK-Irst, Trento, Italia
[2] DISI, Università degli Studi di Trento, Italia

Abstract. We consider the safety problem for Administrative Role-Based Access Control (ARBAC) policies, i.e. detecting whether sequences of administrative actions can result in policies by which a user can acquire permissions that may compromise some security goals. In particular, we are interested in sequences of safety problems generated by modifications (namely, adding/deleting an element to/from the set of possible actions) to an ARBAC policy accommodating the evolving needs of an organization. or resulting from fixing some safety issues. Since problems in such sequences share almost all administrative actions, we propose an incremental technique that avoids the re-computation of the solution to the current problem by re-using much of the work done on the previous problem in a sequence. An experimental evaluation shows the better performances of an implementation of our technique with respect to the only available approach to solve safety problems for evolving ARBAC policies proposed by Gofman, Luo, and Yang.

1 Introduction

Today, the administration of access control policies is key to the security of many IT systems that need to evolve in rapidly changing environments and dynamically finding the best trade-off among a variety of needs. Permissions to perform administrative actions must be restricted since security officers can only be partially trusted. In fact, some of them may collude to—inadvertently or maliciously—modify the policies so that untrusted users can get security-sensitive permissions. A way to restrict administrative permissions is to specify a set of administrative actions, each one identifying conditions about which administrators can modify certain policies. Despite this restriction, taking into consideration the effect of all possible sequences of administrative actions turns out to be a difficult task because of the huge number of ways in which actions can be interleaved and the resulting explosion in the generated policies. Thus, push-button analysis techniques are needed to identify *safety* issues, i.e. administrative actions generating policies by which a user can acquire permissions that may compromise some security goals. This is known as the *safety problem*, which amounts to establish whether there exists a (finite) sequence of administrative actions, selected from a set of available ones, that applied to a given initial policy, yields a policy in which a user gets certain permissions.

To further complicate the problem, administrative actions tend to evolve over time in order to correct potential security issues revealed after solving safety problems or to accommodate the changing needs of an organization. After each change, administrators may wish to solve safety problems to check if a security issue has been fixed or if the change introduced a new security issue. Since typical changes have the form of sequences of simple operations that add/delete an action to/from the available set of administrative actions, the available technique for safety analysis is invoked on "similar" problems—i.e. safety problems that share almost all administrative actions. It would be thus good to compute the result for the new problem instance by re-using as much as possible the computations performed for the previous problem instance and possibly performing only those computations needed to take into account the change in the set of administrative actions. This is an example of *incremental* computation [15].

In this paper, we propose an incremental technique capable of solving sequences of safety problems for Role-Based Access Control (RBAC) policies [20]; the administrative actions we consider are those of the Administrative RBAC model (ARBAC) [19]. We derive an incremental version of our procedure for analyzing single instances of the ARBAC safety; see, e.g., [18]. This amounts to computing a symbolic representation of the set of RBAC policies from which the goal is reachable, called the set of *backward* reachable states. We do this by re-using the set of backward reachable states computed to solve a "similar" instance of the ARBAC safety problem. In some situations, this is easy: consider adding a new administrative action when the answer to the previous instance of the problem was "reachable." The answer to the new instance of the problem is obviously again "reachable" since the administrative actions used in the previous instance to show reachability can still be used to solve the new instance, that simply contains one more action. In other situations, finding an answer to a similar safety problem is more complex: consider adding a new administrative action when the answer to the previous instance of the problem was "unreachable." The answer to the new instance of the problem requires additional computations in order to understand if the set of backward reachable states has been enlarged and the new action may turn the answer from "unreachable" to "reachable" or not. As we will see, our procedure tries to recognize situations in which it is easy to infer an answer from previous computations while deferring additional computations when this cannot be avoided.

As observed in [15], the theoretical criteria commonly used to evaluate the performances of non-incremental procedures can be unsatisfactory when considering incremental changes. (Recall that solving single instances of ARBAC safety problems is PSPACE-complete; see, e.g., [21].) Additionally, "from a practical standpoint incremental algorithms that do not have "good" theoretical performance [...] can give satisfactory performance in practice" [15]. For these reasons, we have performed an experimental evaluation of an implementation of our incremental procedure on randomly generated safety problems. We have compared our procedure with those in [8, 9]: the results clearly show the advantages of our approach.

Related work. Deriving incremental versions of batch algorithms is a much studied topic in several fields of Computer Science; we point the reader to [15] for a general overview on incremental computation.

There is a long line of works on the safety analysis of access control policies started with the seminal work in [11]. The idea underlying such works is to reduce safety analysis to graph manipulation [13, 3, 22] or fix-point computation performed either by Logic Programming—as in [14]—or model checking—as in [14, 23, 4, 2, 5, 12, 6, 18]!. All these works do not consider incremental analysis.

The first work to propose the analysis of evolving ARBAC policies is [8, 9]. Besides arguing the importance of incremental analysis, [8, 9] propose incremental versions of the algorithms for analyzing ARBAC policies of [23]. This work is our main source of inspiration: we share the same motivations and take a similar approach by proposing an incremental version of our technique for the automated analysis of ARBAC policies [18]. The main difference is in the underlying model checking procedure: we use an implicit (symbolic) representation of the set of RBAC policies obtained by applying administrative actions whereas [8, 9] use the explicit-state model checking technique of [23]. More recently, [10] presents a symbolic analysis procedure—based on a sophisticated Logic Programming technique, called abduction—for rule-based administrative policies, which are also capable of expressing changes to the policy rules. A comparison with our approach or that of [8, 9] is complex because of the differences in expressive power. On the one hand, the rule-based administrative policies of [10] can express only a sub-set of the ARBAC policies since they do not support, e.g., negations in the conditions to apply an action. On the other hand, they can express modifications of RBAC policies that cannot be expressed by ARBAC policies (the interested reader is pointed to [10] for details). For this reason, we compare our technique with those of [8, 9] only (see Section 5 below).

Plan of the paper. Section 2 introduces the background on the safety of ARBAC policies. Section 3 summarizes our symbolic procedure to solve safety problems, that is made incremental in Section 4. Our implementation and its experimental evaluation are discussed in Section 5. The paper concludes in Section 6.

2 RBAC and ARBAC

In *Role-Based Access Control (RBAC)* [20], 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. For the purpose of safety analysis, without loss of generality (see, e.g., [21]), we ignore role hierarchies (a remark on this assumption is at the end of this section). 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 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 $(r, p) \in PA$ and u is a member of r. A *RBAC policy* is a tuple (U, R, P, UA, PA).

Administrative RBAC (ARBAC) [19] controls how RBAC policies may evolve through administrative actions that assign or revoke user memberships into roles. Usually (see, e.g., [23]), administrators may only update the relation UA while PA is fixed. Thus, a RBAC policy is a tuple (U, R, UA) or simply UA when U and R are clear from the context. This simplification is assumed in several works in the literature and we also adopt it in this paper (a remark on the significance of this assumption is at the end of this section).

The set of possible administrative actions is defined by rules specifying the which roles an administrator should or should not have—this is also called the *administrative domain* of the rule—and which roles a user should have to get a role assigned or revoked. An administrative domain is specified by a *pre-condition*, i.e. a finite set of expressions of the forms r or $\overline{r}$ (for $r \in R$). 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 [23].

The semantics of the administrative actions in the set ψ of rules obtained by the disjoint union of rules in *can_assign* and *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 sequence $UA_0, UA_1, ..., UA_n, ...$ such that $UA_i \rightarrow_\psi UA_{i+1}$ for $i \geq 0$.

The safety problem for ARBAC policies. 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*, identified by the tuple $\langle UA, \psi, (u_g, R_g)\rangle$, consists of checking 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) u_g is a member of each role of R_g in UA_n.

Sometimes, to simplify the solution of user-role reachability problems, *separate administration* has been assumed (see, e.g., [23]), which amounts to requiring that administrative and regular roles are disjoint. This permits to consider just one user, omit administrative users and roles so that the tuples in *can_assign* are pairs composed of a simple pre-condition and a target role and the pairs in

can_revoke reduce to target roles only. Although our approach does not need such an assumption, we make use of it in the examples to simplify the technical development and in the experiments of Section 5 to streamline the comparison with the approach in [8, 9].

Remark. In [19], modifications to role hierarchies are allowed. Since they closely reflect the structure of the organizations in which the policies are used, we believe that their modifications should be rare as they imply substantial changes to the organizations themselves. Thus, we decided to disregard administrative actions modifying the role hierarchy of RBAC policies.

In [19], it is also possible to modify the permission-role assignment relation PA by administrative actions similar to those in ψ for UA (obtained by simply replacing users with permissions). There exists a reduction [21] for the problem of checking if a user can be assigned a given set of permissions—called the *user-permission reachability problem*—into a (finite) set of independent user-role and permission-role reachability problems. A *permission-role reachability problem* consists of answering questions of the form: can a set of permissions be assigned to a given set of roles by applying a finite sequence of actions modifying the relation PA? Given the similarities between the actions modifying PA and UA, analysis techniques for the user-role reachability problem can be easily adapted to the permission-role reachability problem [21]. As a consequence, the incremental analysis techniques described in the following can be extended to take care of administrative actions modifying also PA.

3 Solving User-Role Reachability Problems

Our approach [4, 2, 5, 18] to solve single instances of user-role reachability problems is based on a symbolically representing user-role reachability problems and then invoking a symbolic model checking procedure. More precisely, we represent RBAC policies, administrative actions, and goals by formulae of first-order logic. Then, we use simple logical manipulations and theorem proving techniques to iteratively compute the symbolic representation of sets of backward reachable states. Technically, we use the definitions and results in [17].

We assume a class $\mathcal{L}$ of first-order formulae and define an $\mathcal{L}$-based *symbolic transition system* S ($\mathcal{L}$-STS, for short) to be a triple $\langle V, In, Tr\rangle$ where V is the (finite) set of system variables of S, In is an assertion of $\mathcal{L}$ describing all the initial states of S and Tr is a (finite) set of assertions of $\mathcal{L}$. The assertion In contains some or all the system variables V; we also write $In(V)$ to emphasize this. Each member tr of Tr is an assertion of $\mathcal{L}$ containing some or all the system variables in V and the primed system variables in $V' = \{v' | v \in V\}$ where v' indicates the values of v after the execution of the transition; we also write $tr(V, V')$ to emphasize this.

We take $\mathcal{L}$ to be the *Bernays-Shönfinkel-Ramsey* (*BSR*) [16] fragment of first-order logic; sometimes called Effectively-Propositional Logic (EPR), see, e.g., [17]. An assertion of BSR has the form $\exists X.\forall Y.\varphi(X, Y)$ where X and Y are (disjoint) sets of variables and φ is a quantifier-free formula—called the *matrix*—

Procedure 1 The translation procedure TR

Require: $P = \langle\langle U, R, UA\rangle, \psi, (u_g, R_g)\rangle$ is a user-role reachability problem
Ensure: $S = \langle V, In, Tr\rangle$ is a $\mathcal{L}$-based STS and G is an assertion of $\mathcal{L}_G$
1: $V \leftarrow R$ { Roles are considered as unary predicates }
2: $In \leftarrow \forall x. \bigwedge_{r\in R}(r(x) \Leftrightarrow \bigvee_{(u,r)\in UA} x = u)$
3: $Tr \leftarrow \{[\alpha] | \alpha \in \psi\}$
4: $G \leftarrow \exists x. x = u_g \wedge \bigwedge_{r_g \in R_g} r_g(x)$

$$[\alpha] := \begin{cases} \exists a, x. \forall y. \begin{pmatrix} \bigwedge_{r\in C_a} r(a) \wedge \bigwedge_{\overline{r}\in C_a} \neg r(a) \wedge \\ \bigwedge_{r\in C} r(x) \wedge \bigwedge_{\overline{r}\in C} \neg r(x) \wedge \\ r_t'(y) \Leftrightarrow (y = x \vee r_t(y)) \wedge Id(R \setminus \{r_t\}) \end{pmatrix} & \text{if } \alpha = (C_a, C, r_t) \\ \exists a, x. \forall y. \begin{pmatrix} \bigwedge_{r\in C_a} r(a) \wedge \bigwedge_{\overline{r}\in C_a} \neg r(a) \wedge \\ r_t(x) \wedge \\ r_t'(y) \Leftrightarrow (y \neq x \wedge r_t(y)) \wedge Id(R \setminus \{r_t\}) \end{pmatrix} & \text{if } \alpha = (C_a, r_t) \end{cases}$$

containing equality, predicates, and constants but no functions. The translation procedure 1 shows how to obtain a symbolic representation of a user-role reachability problem. The idea is to consider the roles in R as unary predicates, so that $r(u)$ can be interpreted as $(u, r) \in UA$. The auxiliary translation function $[\cdot]$ maps rules in *can_assign* or *can_revoke* to BSR formulae, where $Id(R^*)$ abbreviates $\bigwedge_{r\in R^*} r'(y) \Leftrightarrow r(y)$. Intuitively, $Id(R^*)$ means that all the roles in $R^* = R \setminus \{r_t\}$ are left unchanged by the administrative action with target role r_t. The BSR formula In does not contain existential quantifiers and it is called a *universal BSR formula*. Dually, the BSR formula G, representing the goal of the user-role reachability problem, does not contain universal quantifiers and it is called an *existential BSR formula*.

Example 1. We illustrate how TR works by means of an example taken from [23]. For simplicity, we assume separate administration (recall the definition at the end of Section 2). Let $U = \{u_1\}$, $R = \{r_1, ..., r_8\}$, initially $UA := \{(u_1, r_1), (u_1, r_4), (u_1, r_7)\}$, rules $(\{r_1\}, r_2)$, $(\{r_2\}, r_3)$, $(\{r_3, \overline{r_4}\}, r_5)$, $(\{r_5\}, r_6)$, $(\{\overline{r_2}\}, r_7)$, and $(\{r_7\}, r_8)$ are in *can_assign*, rules (r_1), (r_2), (r_3), (r_5), (r_6), and (r_7) are in *can_revoke*, and the goal be $(u_1, \{r_6\})$.

TR works by first introducing the unary predicates $r_1, ..., r_8$ in V. Then, it forms the following universal formula In from UA:

$$\forall x. \begin{pmatrix} (r_1(x) \Leftrightarrow x = u_1) \wedge (r_4(x) \Leftrightarrow x = u_1) \wedge (r_7(x) \Leftrightarrow x = u_1) \wedge \\ \neg r_2(x) \wedge \neg r_3(x) \wedge \neg r_5(x) \wedge \neg r_6(x) \wedge \neg r_8(x) \end{pmatrix}.$$

The translation of $(\{r_5\}, r_6)$ in *can_assign* is $\exists x. \forall y. (r_5(x) \wedge (r_6'(y) \Leftrightarrow (y = x \vee r_6(y))) \wedge Id(R \setminus \{r_6\}))$, that of (r_1) in *can_revoke* is $\exists x. \forall y. (r_1(x) \wedge (r_1'(y) \Leftrightarrow (y \neq x \wedge r_1(y))) \wedge Id(R \setminus \{r_1\}))$, and that of the goal is $G := \exists x. (r_6(x) \wedge x = u_1)$. □

Given the translation procedure 1, we can now explain how solving the reachability problem for BSR-based STS is equivalent to solving user-role reachability problems. First, define a *state* of a BSR-based STS $S = \langle V, In, Tr\rangle$ to be a mapping from the set V to a user-role assignment relation such that $(u, r) \in UA$ when

Procedure 2 The backward reachability procedure `BR`

Input: $S = \langle V, In, Tr \rangle$ is a $\mathcal{L}$-based STS and G is an assertion of $\mathcal{L}$
Output: answer to the reachability problem for S and G

1: $P \leftarrow G$; $B \leftarrow \mathit{false}$;
2: **while** ($P \wedge \neg B$ is satisfiable) **do**
3: **if** ($In \wedge P$ is satisfiable) **then**
4: **return** `reachable`
5: **end if**
6: $B \leftarrow P \vee B$;
7: $P \leftarrow \mathit{Pre}(Tr, P)$;
8: **end while**
9: **return** `unreachable`

$r(u)$ holds. Then, define a run of S as a (possibly infinite) sequence $s_0, s_1, ...$ of states such that s_0 satisfies (in the sense of first-order logic) In and for every $i = 0, 1, ...$, we have that s_i, s_{i+1} satisfy tr for some tr in Tr. Let $s_0, ..., s_n, ...$ be a run of S, we say that s_n is *reachable* from s_0 for $n \geq 1$. The *reachability problem* for a STS S and a *goal* assertion G of BSR amounts to establish if there exists a run $s_0, ..., s_n, ...$ of S such that s_n satisfies (in the sense of first-order logic) G. By definition of states of a BSR-STS, it is easy to transform sequences of states to sequences of RBAC policies. Thus, the answer to a user-role reachability problem P is the same to that of the reachability problem for a BSR-STS S and goal G when $(S, G) = \mathtt{TR}(P)$.

We are now left with the problem of solving reachability problems for BSR-STSs. We use the approach in [17], summarized in Procedure 2. `BR` tries to compute an assertion of $\mathcal{L}$, stored in B, characterizing all states from which those satisfying G are reachable by executing finitely many transitions in Tr. If the loop terminates at iteration n, then B stores such an assertion since it contains the disjunction of the n pre-images of G (stored in P). The pre-image of an assertion K of $\mathcal{L}$ containing the system variables in V relative to a transition tr in Tr is defined as follows:

$$\mathit{Pre}(tr, K) := \exists V'.(tr(V, V') \wedge K(V'))$$

and characterizes the set of states from which those satisfying assertion K are reachable. By abuse of notation, we write $\mathit{Pre}(Tr, K)$ instead of $\mathit{Pre}(\bigvee_{tr \in Tr} tr, K)$. Simple logical manipulations show that Pre is *monotonic in its first argument*, i.e. if $Tr_1 \subseteq Tr_2$ then $\mathit{Pre}(Tr_1, K) \Rightarrow \mathit{Pre}(Tr_2, K)$ is valid.

Let us consider the n-iteration of the loop in `BR`, P stores $\mathit{Pre}^n(Tr, G)$ and B stores $\bigvee_{i=0}^{n} \mathit{Pre}^i(Tr, G)$, where $\mathit{Pre}^0(Tr, G) := G$ and $\mathit{Pre}^{k+1}(Tr, G) := \mathit{Pre}(Tr, \mathit{Pre}^k(Tr, G))$ for $k = 0, ..., n-1$. Upon termination of `BR`, the assertion in B is the set of *backward reachable states (from G)*. The formulae checked for satisfiability at lines 2 and 3 are

$$\mathit{Pre}^n(Tr, G) \wedge \neg \bigvee_{i=0}^{n-1} \mathit{Pre}^i(Tr, G) \tag{1}$$

$$In \wedge Pre^n(Tr, G)\,, \tag{2}$$

respectively. Checking the satisfiability of (1) is the *fix-point test* since it is equivalent (by refutation) to verifying the validity of $Pre^n(Tr, G) \Rightarrow \bigvee_{i=0}^{n-1} Pre^i(Tr, G)$ while the converse, i.e. the validity of $\bigvee_{i=0}^{n-1} Pre^i(Tr, G) \Rightarrow Pre^n(Tr, G)$ holds by definition of pre-image. The un-satisfiability of (2) at every iteration $i \leq n$ implies the un-satisfiability of $In \wedge \bigvee_{i=0}^{n} Pre^i(Tr, G)$. This means that checking that none of the states characterized by the assertion in B are allowed as initial states of S is equivalent to verify that there is no state satisfying G that is also reachable from an initial state and G is unreachable (cf. line 9 of Procedure 2).

Theorem 1 ([17]). `BR` *is a decision procedure for the reachability problem of BSR-STSs.*

The main argument underlying the proof of this result is two-fold. First, the satisfiability checks at line 2 and 3 of Procedure 2 are decidable. To see this, observe that formulae (1) and (2) can be rewritten to logically equivalent BSR formulae, whose satisfiability is well-known to be decidable (see, e.g., [16]). Efficient theorem proving techniques (see, e.g., [17])—available in Satisfiability Modulo Theories (SMT) solvers—can be used in practice to tackle such satisfiability checks. Second, the procedure always terminates by using a standard technique for proving termination (see [17] for details).

Let $\texttt{Solve}(P) = \texttt{BR}(\texttt{TR}(P))$ for a user-role reachability problem P.

Corollary 1. `Solve` *decides the user-role reachability problem.*

This follows from Theorem 1 and the fact that $r(u)$ is interpreted as $(u, r) \in UA$.

Example 2. Let us consider again the user-role reachability problem in Example 1. The BSR formula representing the (fix-point) set of backward reachable states obtained by invoking `BR` is

$$\exists x.\begin{pmatrix}(r_6(x) \wedge x = u_1) \vee (r_5(x) \wedge x = u_1) \vee (r_3(x) \wedge \neg r_4(x) \wedge x = u_1) \vee \\ (r_2(x) \wedge \neg r_4(x) \wedge x = u_1) \vee (r_1(x) \wedge \neg r_4(x) \wedge x = u_1)\end{pmatrix} \tag{3}$$

It is not difficult to verify that the initial RBAC policy UA (see again Example 1) is not in the set of states represented by (3), by checking that the conjunction of In—representing UA and (3) is unsatisfiable (recall that this check can be done automatically because of the decidability of BSR formulae). We are thus entitled to conclude that the goal $(u_1, \{r_6\})$ is unreachable. □

Remark. By properties stated and proved in [17] (from which Corollary 1 derives), we observe that `Solve` is capable of solving user-role reachability problem for a *finite but unknown number of users*. This means that our technique for safety analysis is capable of coping with dynamic situations in which users may join or leave the organization in which the RBAC policies are administered. This dramatically enlarges the scope of applicability of the analysis and thus the usefulness of its results. For lack of space, we cannot discuss the reasons of this; the interested reader is pointed to [17]. The incremental version of `Solve` developed below inherits this feature.

4 Incremental Analysis of Evolving ARBAC Policies

An *evolving ARBAC policy* is a pair $(\psi;\omega)$ where ψ is the "original" set of rules and ω is a finite sequence of operations of the form $\mathsf{add}(\alpha)$ or $\mathsf{delete}(\alpha)$. Applying $\mathsf{add}(\alpha)$ ($\mathsf{delete}(\alpha)$, respectively) to ψ generates a new set of rules $\psi \cup \{\alpha\}$ ($\psi \setminus \{\alpha\}$, respectively). The sequence $P_0, ..., P_n$ of user-role reachability problems induced by an evolving ARBAC policy $(\psi; op_1, ..., op_n)$ and an initial user-role reachability problem $P = \langle UA, \psi, (u_g, R_g)\rangle$ is such that $P_0 = P$ and $P_i = \langle UA, \psi_i, (u_g, R_g)\rangle$ where ψ_i is obtained from ψ_{i-1} by applying the operation op_i for $i = 1, ..., n$. We now derive an incremental version of the procedure `Solve` capable of re-using the set of backward reachable states computed to solve P_{i-1} to infer an answer for P_i.

The following two observations are the basis of our approach. First, under certain conditions, it is possible to derive the answer to P_i from that of P_{i-1} without invoking the symbolic reachability procedure `BR` (Procedure 2). Second, it is possible to design an "incremental" version of `BR`, denoted `iBR`, capable of re-using a symbolic representation of the set of backward reachable states computed in a previous invocation. While such a procedure—called, `iBR`—will be described in Section 4.1, for the time being, it is sufficient to know that it takes as input a BSR-STS $S = \langle V, In, Tr\rangle$ and a goal formula G together with the reference B to a formula representing the set of backward reachable states computed in a previous invocation of `iBR`. It then returns either $(\texttt{unreachable}, \epsilon)$, with ϵ being the empty sequence, iff G is unreachable from In by applying a finite sequence of transitions in Tr or $(\texttt{reachable}, \sigma)$ with σ being a non-empty sequence of transitions in Tr leading from In to G. The pre-condition of `iBR` is that

$$\text{there exist } n \geq 0 \text{ and } \widehat{Tr} \subseteq Tr \text{ such that } B \text{ refers to } \textstyle\bigvee_{i=0}^{n} Pre^i(\widehat{Tr}, G)\,. \quad (4)$$

We emphasize that, when invoking $\texttt{iBR}(S, G, B)$, the parameters S and G are passed by value whereas B by reference so that, upon returning, B refers to the newly computed set of backward reachable states. We assume that after an invocation to `iBR`, (4) holds again, i.e. (4) is an invariant of `iBR`. If $n = 0$, then the formula above reduces to *false* and $\texttt{iBR}(S, G, \mathit{false})$ is equivalent to $\texttt{BR}(S, G)$, except for the capability of returning the sequence of transitions leading from an initial state to one satisfying the goal G together with `reachable`.

We are now ready to describe the incremental version `SolveEvolving` of `Solve`, shown in Procedure 3. The original user-role reachability problem $\langle UA, \psi, (u_g, R_g)\rangle$ is translated (by invoking `TR`, c.f. Procedure 1), solved from scratch (the third parameter of `iBR` refers to *false* meaning that there is no previous knowledge about the set of backward reachable states), and the user is notified of the result via `notify` (line 1). Such a procedure simply prints out a message reporting if the user-role reachability problem under consideration (identified with `original` or the operation *op* that has been applied) is reachable or unreachable and, in the first case, also shows the sequence σ of administrative actions leading from the initial RBAC policy to one satisfying the goal. Afterwards (lines 2-25), the processing of the sequence ω of operations for adding or deleting administrative actions is started until none is left. At each iteration, an operation *op* in ω is

Procedure 3 `SolveEvolving`

Input: Original user-role reachability problem $\langle UA, \psi, (u_g, R_g)\rangle$ and sequence ω of operations

1: $(S, G) \leftarrow \texttt{TR}(\langle UA, \psi, (u_g, R_g)\rangle)$; $(res, \sigma) \leftarrow \texttt{iBR}(S, G, false)$; $\texttt{notify}(\textsf{original}, res, \sigma)$;
2: **while** $(\omega \neq \epsilon)$ **do**
3: $op \leftarrow \texttt{first}(\omega)$; $\omega \leftarrow \texttt{rest}(\omega)$;
4: **if** $(res = \texttt{reachable})$ **then**
5: **if** $(op = \textsf{add}(\alpha))$ **then**
6: $S \leftarrow S \oplus [\alpha]$;
7: **else if** $(op = \textsf{delete}(\alpha))$ **then**
8: $S \leftarrow S \ominus [\alpha]$; $B \leftarrow \texttt{Filter}(B, \alpha)$;
9: **if** $(\alpha \in \sigma)$ **then**
10: $(res, \sigma) \leftarrow \texttt{iBR}(S, G, B)$;
11: **end if**
12: **end if**
13: **else if** $(res = \texttt{unreachable})$ **then**
14: **if** $(op = \textsf{add}(\alpha))$ **then**
15: $S \leftarrow S \oplus [\alpha]$;
16: Let r_t be the target role of α and $F \leftarrow \begin{cases} \exists x.r_t(x) & \text{if } \alpha \in can\_assign \\ \exists x.\neg r_t(x) & \text{if } \alpha \in can\_revoke \end{cases}$
17: **if** $(F \Rightarrow B$ is valid$)$ **and** $(Pre([\alpha], B) \Rightarrow B$ is invalid$)$ **then**
18: $(res, \sigma) \leftarrow \texttt{iBR}(S, G, B)$;
19: **end if**
20: **else if** $(op = \textsf{delete}(\alpha))$ **then**
21: $S \leftarrow S \ominus [\alpha]$; $B \leftarrow \texttt{Filter}(B, \alpha)$;
22: **end if**
23: **end if**
24: $\texttt{notify}(op, res, \sigma)$;
25: **end while**

considered (line 3) and one among the following four cases (lines 5-6, 7-12, 14-19, or 20-22) is executed depending on the answer *res* to the previous problem and the type of the operation *op*. In each, the translation of a rule α is added (line 6 or 15) to or deleted (line 8 or 21) from S, where $S \oplus [\alpha] = \langle V, In, Tr \cup \{[\alpha]\}\rangle$ and $S \ominus [\alpha] = \langle V, In, Tr \setminus \{[\alpha]\}\rangle$ when $S = \langle V, In, Tr\rangle$. Let $P_0, ..., P_n$ be the sequence of user-role reachability problems induced by the evolving ARBAC policy $(\psi; \omega)$, it is easy to see that $(S, G) = \texttt{TR}(P_i)$ at iteration $i = 0, ..., n$ of the loop when executing $\texttt{SolveEvolving}(P, \omega)$. We now describe each case in detail.

Let $\overline{P} = \langle UA, \overline{\psi}, (u_g, R_g)\rangle$ and $\overline{S} = \texttt{TR}(\overline{P}) = \langle V, In, \overline{Tr}\rangle$ be the user-role reachability problem and the content of the variable S at the previous iteration of the loop, respectively, together with $\overline{res}$ and $\overline{\sigma}$ be the values stored in the variables *res* and σ, respectively, at the previous iteration of the loop.

(lines 5-6) $\overline{res}$ is `reachable` and *op* is $\textsf{add}(\alpha)$: indeed, the goal (u_g, R_g) is still reachable since all the actions in $\overline{\sigma}$ are in $\overline{\psi} \cup \{[\alpha]\}$, i.e. are still available in the new user-role reachability problem. So, there is no need to invoke `iBR` and `SolveEvolving` can notify the user that the answer is again $(\overline{res}, \overline{\sigma})$ (line 24). It

is easy to see that the invariant (4) of `iBR` still holds at the end of the current iteration of the loop.

(lines 7-11) $\overline{res}$ is `reachable` and *op* is $\mathsf{delete}(\alpha)$: after removing α from S, we invoke the function `Filter` so that B now contains the formula

$$\bigvee_{j=0}^{m} Pre^j(\overline{Tr} \setminus \{[\alpha]\}, G) \tag{5}$$

describing the sub-set of backward reachable states of those computed by the last invocation of `iBR` not taking into account the action α being deleted (for some $m \geq 0$). I.e. (5) is the post-condition of the function `Filter`, whose efficient implementation will be described in Section 4.1. Since (5) holds after the invocation of `Filter`, it is easy to see that also the invariant (4) of `iBR` holds at the end of the current iteration of the loop. Then, we consider two cases. If the deleted action α is not in the sequence $\overline{\sigma}$ of administrative actions leading from the initial policy to one satisfying the goal, then `SolveEvolving` can immediately notify the user that the answer is again $(\overline{res}, \overline{\sigma})$ (line 24). Otherwise (i.e. α is in $\overline{\sigma}$), we invoke `iBR` on the new reachability problem $\overline{S} \ominus [\alpha]$ and the user is notified of the newly computed values of *res* and σ (line 24).

(lines 14-19) $\overline{res}$ is `unreachable` and *op* is $\mathsf{add}(\alpha)$: we consider depending on the fact that the set $\overline{B}$ of backward reachable states is affected or not by the addition of the administrative action α. To verify this, we first build the formula F (line 16) which is satisfied by any set of states in which the target role r_t of α is assigned (if $\alpha \in$ *can_assign*) or not (if $\alpha \in$ *can_revoke*) to some user. Then, it is checked whether F is contained in $\overline{B}$ ($F \Rightarrow B$ is valid or, by refutation, $F \wedge \neg B$ is unsatisfiable) but the set of states described by the pre-image of $\overline{B}$ with respect to $[\alpha]$ is not included in $\overline{B}$ ($Pre([\alpha], \overline{B}) \Rightarrow \overline{B}$ is invalid or, by refutation, $Pre([\alpha], \overline{B}) \wedge \neg \overline{B}$ is satisfiable). Notice that both checks are decidable because of the decidability of the satisfiability of BSR formulae. If $\alpha \in$ *can_assign* and the check fails, then $\overline{B}$ is unaffected by the addition of α since either the target role r_t is assigned to no user in $\overline{B}$ or the pre-image of $\overline{B}$ with respect to $[\alpha]$ is already in the fix-point; similarly, when $\alpha \in$ *can_revoke*. If the check succeeds, `iBR` is invoked on the updated set $\overline{S} \oplus [\alpha]$ of transition formulae and the goal G while considering the previously computed set $\overline{B}$ of backward reachable states. In both cases, it is not difficult to see that that the invariant (4) of `iBR` is maintained at the end of the current iteration.

Example 3. Let $\overline{P}$ be the user-role reachability problem in Example 1. Recall that the set of backward reachable states is represented by (3) in Example 2. Now, consider the following three problems all derived from $\overline{P}$ by adding action
– $(\{r_3\}, r_7)$: $F = \exists x.r_7(x)$, $F \Rightarrow (3)$ is invalid, `SolveEvolving` can notify that the answer is `unreachable`;
– $(\{r_1\}, r_3)$: $F = \exists x.r_3(x)$, $F \Rightarrow (3)$ is valid, $Pre([(\{r_1\}, r_3)], (3))$ is $\exists x.(r_1(x) \wedge \neg r_4(x) \wedge x = u_1)$, which is the last disjunct in (3) and thus $Pre([(\{r_1\}, r_3)], (3)) \Rightarrow (3)$ is trivially valid, `SolveEvolving` can notify that the answer is `unreachable`;

$-$ $(\{r_1\}, r_5)$: $F = \exists x.r_5(x)$, $F \Rightarrow (3)$ is valid, $Pre([(\{r_1\}, r_5)],\ (3))$ is $\exists x.(r_1(x) \wedge x = u_1)$ that is not in (3), and thus $Pre([(\{r_1\}, r_5)], (3)) \Rightarrow (3)$ is invalid, and we need to invoke `iBR` on the new reachability problem. □

(lines 20-22) $\overline{res}$ is `unreachable` and *op* is delete(α): indeed, the goal (u_g, R_g) is still unreachable since, at the previous iteration of the loop, we had that $In \wedge \bigvee_{i=0}^{n} Pre^i(\overline{Tr}, G)$ was unsatisfiable as $\overline{res}$ is `unreachable`. Now, observe that $\bigvee_{j=0}^{m} Pre^j(\overline{Tr} \setminus \{[\alpha]\}, G) \Rightarrow \bigvee_{i=0}^{n} Pre^i(\overline{Tr}, G)$ is valid since *Pre* is monotonic with respect to its first argument (as stated in Section 3). From these, by simple Boolean reasoning, we can conclude that $In \wedge \bigvee_{j=1}^{m} Pre^j(\overline{Tr} \setminus \{[\alpha]\}, G)$ is also unsatisfiable. Because of the post-condition (5) of `Filter`, it is easy to see that the invariant (4) of `iBR` is maintained.

Example 4. Let $\overline{P}$ be again the user-role reachability problem in Example 1. Consider removing action $(\{r_2\}, r_3)$ from $\overline{P}$. It is easy to see that the goal is still unreachable. By invoking the function `Filter`, variable B refers to the formula

$$\exists x.\ \big((r_6(x) \wedge x = u_1) \vee (r_5(x) \wedge x = u_1) \vee (r_3(x) \wedge \neg r_4(x) \wedge x = u_1) \big)\,, \quad (6)$$

which is obtained by deleting the pre-image of the goal w.r.t. $(\{r_2\}, r_3)$, namely $\exists x.(r_2(x) \wedge \neg r_4(x) \wedge x = u_1)$ and $\exists x.(r_1(x) \wedge \neg r_4(x) \wedge x = u_1)$. `SolveEvolving` can consider new operations with the new set of backward reachable states. □

4.1 `iBR` and `Filter`

Procedures `iBR` and `Filter` use a decorated version of formulae. Let ϵ denote the empty sequence and σ be a (finite, possibly empty) sequence of administrative actions, i.e. either $\sigma = \epsilon$ or there exists $l \geq 1$ such that $\sigma = \alpha_1; \ldots; \alpha_l$ for α_i an administrative action with $i = 1, ..., l$. If K is a BSR formula, then K^σ is a *decorated BSR formula* for σ a sequence of administrative actions. The logical reading of the decorated BSR formulae K^σ is simply the BSR formula K; for instance, $K_1^{\sigma_1} \Rightarrow K_2^{\sigma_2}$ is equivalent to $K_1 \Rightarrow K_2$. We will also use Boolean combinations of standard and decorated BSR formulae, its logical meaning is simply obtained by forgetting the decorations; for example, the meaning of $K_1 \wedge K_2^\sigma$ is simply $K_1 \wedge K_2$ with σ a sequence of administrative actions.

We derive the incremental version `iBR` of `BR` by performing the following two modifications on the code of Procedure 1. First, we replace line 1 with

if $(B = \mathit{false})$ **then** $P \leftarrow G^\epsilon$; **else** $P \leftarrow B$; **end if**

i.e. variable P contains the goal G decorated by the empty sequence ϵ when there is no information about previous reachability problems; otherwise, P contains the set of states computed in a previous invocation of `iBR` (recall that B is a parameter passed by reference). The second modification concerns the computation of pre-images: $Pre([\alpha], K^\sigma)$ is the BSR formula $Pre([\alpha], K)$ decorated by the sequence $\alpha; \sigma$ of administrative actions. It is thus not difficult to see that variables P and B contain decorated BSR formulae of the form

$$\bigvee_{i=0}^{m} K_i^{\sigma_i} \quad (7)$$

for some $m \geq 0$, K_i is a decorated BSR formula for $i = 0, ..., m$ (as before, the formula reduces to *false* when $m = 0$). The last modification to the code of Procedure 1 refers to the conditional at lines 3-5. Recalling (7), we can assume that the decorated version of (2) has the form $In \wedge (7)$. Such a formula is satisfiable iff there exists $i^* \in \{0, ..., m\}$ such that $In \wedge K_i^{\sigma_i}$ is so. Thus, we replace lines 3-5 with the following

for $i = 0$ **to** m **do**
  if ($In \wedge K_i^{\sigma_i}$ is satisfiable) **then** (`reachable`, σ_i) **end if**
end for

which allows `iBR` to return the sequence of administrative actions making a goal reachable, by simply reading the decoration σ_i of a disjunct $K_i^{\sigma_i}$ of the decorated BSR formula stored in P. The fact that (4) is an invariant of `iBR` can be shown by a case-analysis on the result returned by the previous invocation of `iBR`.

`Filter` also exploits decorated BSR formulae. Let (7) be the formula in B and α an administrative action, `Filter`(B, α) returns the decorated BSR formula $\bigvee_{j \in J} K_j^{\sigma_j}$ for $J = \{j \in \{0, ..., m\} | \alpha \notin \sigma_j\}$. It is easy to verify that the post-condition (5) holds.

5 Implementation and Experiments

We have used Python to implement `SolveEvolving` in a system called iASASP. The tool is an evolution of ASASP [1, 18], which can be seen as an implementation of `Solve` (introduced immediately before Corollary 1, towards the end of Section 3). The implementation of `iBR` is done by invoking the model checker MCMT [7] to re-use its capability of saving the symbolic representation of the set of backward reachable states to a file and consider it for later invocations. Below, we describe an experimental evaluation comparing iASASP with an implementation of the approach in [8, 9] on a set of randomly generated benchmark problems. We consider four versions of the technique in [8, 9]: *IncFwd1*, *IncFwd2*, and *LazyInc* are based on the forward reachability algorithm of [23] and assume separate administration whereas *IncFwdWSA* is an incremental forward algorithm for user-role reachability problems not assuming separate administration. The reader interested in the description of these algorithms is pointed to [8, 9]; for this paper, it is sufficient to know that these algorithms incorporate ideas (of increasing degree of sophistication) to permit the re-use of previously computed sets of reachable states.

We consider six sets of benchmarks whose characteristics are shown in tables `T1`, `T2`, and `T3` of Figure 1. The user-role reachability problems in $B_1, ..., B_4$ assume separate administration whereas those in B_5 and B_6 do not (first column of `T1`). The initial user-role reachability problems in $B_1, ..., B_4$ share the same (empty) initial RBAC policy and the same set ψ_1 of administrative actions (second column of `T1` under 'Initial problem'). The problems in B_5 and B_6 have two distinct (non-empty) initial RBAC policies (UA_1 and UA_2, respectively) and share the same set ψ_2 of administrative actions. The answer to the initial user-role reachability problem is shown in column 'Answer' of `T1` and the time t_1 taken by our tool and that t_2 taken by the tool of [8, 9] are in column 'Time' with

T1	Separate Administration	Initial Problem	Answer	Time	$\|\omega\|$	Number of instances
B_1	Yes	$\langle \emptyset, \psi_1, g_1 \rangle$	Reach.	43.28/78.16	1	32
B_2	Yes	$\langle \emptyset, \psi_1, g_2 \rangle$	Unreach.	41.15/80.22	1	32
B_3	Yes	$\langle \emptyset, \psi_1, g_1 \rangle$	Reach.	43.28/78.16	3, 5, 7, 10, 15, 20	15
B_4	Yes	$\langle \emptyset, \psi_1, g_2 \rangle$	Unreach.	41.15/80.22	3, 5, 7, 10, 15, 20	15
B_5	No	$\langle UA_1, \psi_2, g_3 \rangle$	Reach.	45.19/83.65	1	32
B_6	No	$\langle UA_2, \psi_2, g_4 \rangle$	Unreach.	61.42/116.73	1	32

T2	$\|can\_assign\|$	$\|can\_revoke\|$	Total
ψ_1	313	64	377
ψ_2	296	55	351

T3	g_1	g_2	g_3	g_4
Size	5	2	3	1

Fig. 1. Characteristics of the 6 benchmark sets

the format t_1/t_2 (both in seconds). The actions in ψ_1 are randomly generated following the approach in [23] while the actions in ψ_2 are those of the university ARBAC policy in [23]. The number of elements in *can_assign* and *can_revoke* (with their sum) are show in T2. The problems in B_1 and B_3 (B_2 and B_4, respectively) share the same goal g_1 (g_2, respectively). The size of the goals (i.e. the number of roles) in the problems are in T3. As shown in column '$|\omega|$' of T1, the length of the sequences ω of "add" and "delete" actions in B_1, B_2, B_5, and B_6 is 1; this means that the tools need to solve just one user-role reachability problem besides the initial one for instances in these benchmark sets. The length of the sequences ω of "add" and "delete" actions in B_3 and B_4 is $\ell = 3, 5, 7, 10, 15, 20$; this means that the tools need to solve ℓ user-role reachability problems besides the initial one. For problems in B_1, B_2, B_5, and B_6, we consider 32 distinct instances of sequences of actions of length 1 while for those in B_3 and B_4, we consider 15 distinct instances of sequences of actions of increasing length $\ell = 3, 5, 7, 10, 15, 20$ (see column 'Number of instances' of T1).

All the experiments were performed on an Intel QuadCore (3.6 GHz) CPU with 16 GB Ram running Ubuntu 11.10. The timings of the tools on the problems in the six benchmarks are reported in Figure 2; they are in seconds, are obtained by averaging the times taken over the number of instances indicated in the last column of T1 (cf. Figure 1), and measure the performance of processing the sequence ω of operations being considered, not including the time used to solve the initial user-role reachability problem since we want to compare the performances of the two approaches in handling changes to the set of administrative operations, not in solving single instance problems. However, notice how ıASASP is better than the tool of [8, 9] on the initial (single instance) problems in the benchmarks $B_1, ..., B_6$ by considering the column 'Time' in Table T1 of Figure 1. Any operation can affect the reachability of the goal in the benchmarks that we consider. The table on the upper-left corner and the two plots refer to benchmark sets B_1, B_2, B_3, and B_4 under separation administration whereas the table on the upper-right corner to B_5 and B_6 that do not assume separate

	Operation in ω	Time			
		IncFwd1	*IncFwd2*	*LazyInc*	iASASP
B_1	add can_assign	**0**	10.31	**0**	0.01
	delete can_assign	69.72	11.14	11.14	**1.75**
	add can_revoke	**0**	1.07	**0**	0.01
	delete can_revoke	12.15	1.72	1.72	**0.47**
B_2	add can_assign	132.93	68.79	68.79	**5.23**
	delete can_assign	**0**	12.09	**0**	0.01
	add can_revoke	19.67	1.25	1.25	**0.69**
	delete can_revoke	**0**	6.44	**0**	0.01

	Operation in ω	Time		
		IncFwd	*WSA*	iASASP
B_5	add can_assign	10.07		**0.03**
	delete can_assign	8.62		**4.84**
	add can_revoke	5.14		**0.03**
	delete can_revoke	2.35		**1.31**
B_6	add can_assign	32.27		**5.35**
	delete can_assign	6.76		**0.03**
	add can_revoke	11.2		**1.05**
	delete can_revoke	0.46		**0.03**

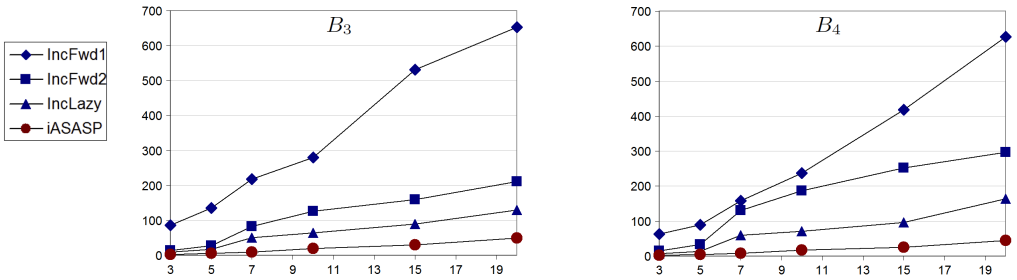

Fig. 2. Comparison of our approach with that of [8, 9] on the six benchmark sets

administration. In almost cases, iASASP performs and scales better than the techniques of [8, 9] as shown by the two plots in Figure 2. iASASP performs better than the best version (*IncLazy*) of the techniques in [8, 9] and clearly outperforms the worse version (*IncFwd1*). For instance in B_4, iASASP processes sequences of length 20 of operations in around 50 seconds whereas *IncLazy* takes more than 150 seconds, *IncFwd2* around 300 seconds, and *IncFwd1* takes more than 600 seconds. The better performances of our approach are due to the sophisticated techniques put in place in `SolveEvolving` to detect when the addition or deletion of an administrative action does not change the answer to the new instance of the reachability problem.

6 Conclusion

The paper discusses an algorithm for the automated analysis of evolving ARBAC policies. The idea is to re-use the previously computed sets of backward reachable states in order to infer the answer to "similar" user-role reachability problems. An experimental evaluation shows that our incremental procedure performs better than the state-of-the-art techniques in [8, 9]. As future work, we plan to extend our experiments on problems under non-separate administration and to compare iASASP also with the backward reachability algorithm in [9].

Acknowledgments. We thank the authors of [8, 9] for making the code of their tool available to us and the help in using it. We also thank the anonymous reviewers for their constructive criticisms.

References

1. F. Alberti, A. Armando, and S. Ranise. ASASP: Automated Symbolic Analysis of Security Policies. In *CADE*, volume 6803 of *LNCS*, pages 26–34. Springer, 2011.
2. F. Alberti, A. Armando, and S. Ranise. Efficient Symbolic Automated Analysis of Administrative Role Based Access Control Policies. In *ASIACCS*. ACM Pr., 2011.
3. P. Ammann, R. Lipton, and R. Sandhu. The expressive power of multi-parent creation in monotonic access control models. *JCS*, 4(2&3):149–196, 1996.
4. A. Armando and S. Ranise. Automated Symbolic Analysis of ARBAC Policies. In *6th STM Workshop*, volume 6710 of *LNCS*, pages 17–33. Springer, 2010.
5. A. Armando and S. Ranise. Scalable Automated Symbolic Analysis of ARBAC Policies by SMT Solving. *JCS*, 20(4):309–352, 2012.
6. A. L. Ferrara, P. Madhusudan, and G. Parlato. Policy Analysis for Self-Administrated Role-based Access Control. In *TACAS*. Springer, 2013.
7. S. Ghilardi and S. Ranise. MCMT: a Model Checker Modulo Theories. In *Proc. of IJCAR'10*, LNCS, 2010.
8. M. Gofman and P. Yang. Efficient Policy Analysis for Evolving Administrative Role Based Access Control. *Int. J. of Software and Informatics*, 2014. To appear.
9. M.I. Gofman, R. Luo, and P. Yang. User-role reachability analysis of evolving administrative role based access control. In *ESORICS*, volume 6345 of *LNCS*, pages 455–471. 2010.
10. P. Gupta, S. D. Stoller, and Z. Xu. Abductive analysis of administrative policies in rule-based access control. In *ICISS*, volume 7093 of *LNCS*, pages 116–130, 2011.
11. M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in Operating Systems. *Communications of ACM*, 19(8):461–471, 1976.
12. K. Jayaraman, V. Ganesh, M. Tripunitara, M. Rinard, and S. Chapin. Automatic Error Finding for Access-Control Policies. In *CCS*. ACM, 2011.
13. M. Koch, L. V. Mancini, and F. Parisi-Presicce. Decidability of Safety in Graph-Based Models for Access Control. In *ESORICS*, volume 2502 of *LNCS*, pages 229–244. 2002.
14. N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM TISSEC*, 9(4):391–420, 2006.
15. G. Ramalingam and T. Reps. A Categorized Bibliography on Incremental Computation. In *Proc. of POPL*, pages 502–510. ACM, 1993.
16. F. P. Ramsey. On a Problem of Formal Logic. *Proceedings of the London Mathematical Society*, s2-30(1):264–286, 1930.
17. S. Ranise. Symbolic Backward Reachability with Effectively Propositional Logic—Applications to Security Policy Analysis. *FMSD*, 42(1):24–45, 2013.
18. S. Ranise, T. A. Truong, and A. Armando. Boosting Model Checking to Analyse Large ARBAC Policies. In *STM Ws*, volume 7783 of *LNCS*, pages 273–288, 2012.
19. R. Sandhu, V. Bhamidipati, and Q. Munawer. The ARBAC97 model for role-based control administration of roles. *ACM TISSEC*, 1(2):105–135, 1999.
20. R. Sandhu, E. Coyne, H. Feinstein, and C. Youmann. Role-Based Access Control Models. *IEEE Computer*, 2(29):38–47, 1996.
21. A. Sasturkar, P. Yang, S. D. Stoller, and C.R. Ramakrishnan. Policy Analysis for Administrative Role Based Access Control. *TCS*, 412(44):6208–6234, 2011.
22. M. Soshi, M. Maekawa, and E. Okamoto. The Dynamic-Typed Access Matrix Model and Decidability of the Safety Problem. *IEICE-TF*, (1):1–14, 2004.
23. S. D. Stoller, P. Yang, C.R. Ramakrishnan, and M. I. Gofman. Efficient policy analysis for administrative role based access control. In *CCS*. ACM Press, 2007.