Abductive Analysis of Administrative Policies in Rule-based Access Control

Puneet Gupta, Scott D. Stoller, and Zhongyuan Xu

Department of Computer Science, Stony Brook University, USA

In large organizations, the access control policy is managed by multiple users (administrators). An administrative policy specifies how each user may change the policy. The consequences of an administrative policy are often non-obvious, because sequences of changes by different users may interact in unexpected ways. Administrative policy analysis helps by answering questions such as user-permission reachability, which asks whether specified users can together change the policy in a way that achieves a specified goal, namely, granting a specified permission to a specified user.

This paper presents a rule-based access control policy language, a rule-based administrative policy model that controls addition and removal of rules and facts, and a symbolic analysis algorithm for answering reachability queries. The algorithm can analyze policy rules that refer to sets of facts (e.g., information about users) that are not known at analysis time. The algorithm does this by computing conditions on the initial set of facts under which the specified goal is reachable by actions of the specified users.

1 Introduction

The increasingly complex security policies required by applications in large organizations cannot be expressed in a precise and robust way using access-control lists or role-based access control (RBAC). This has led to the development of attribute-based access control frameworks with rule-based policy languages. These frameworks allow policies to be expressed at a higher level of abstraction, making the policies more concise and easier to administer.

In large organizations, access control policies are managed by multiple users (administrators). An *administrative framework* (also called *administrative model*) is used to express policies that specify how each user may change the access control policy. Several administrative frameworks have been proposed for role-based access control, starting with the classic ARBAC97 model [6]. Fully understanding the implications of an administrative policy can be difficult, due to unanticipated interactions between interleaved sequences of changes by different users. This motivated research on analysis of administrative policies. For example, analysis algorithms for ARBAC97 and variants thereof can answer questions such as user-permission reachability, which asks whether changes by specified users can lead to a policy in which a specified user has a specified permission [5, 4, 7].

There is little work on administrative frameworks for rule-based access control [1, 3], and it considers only addition and removal of facts, not rules.

This paper defines a rule-based access control policy language, with a rule-based administrative framework that controls addition and removal of both facts and rules. We call this policy framework ACAR (Access Control and Administration using Rules). It allows administrative policies to be expressed concisely and at a desirable level of abstraction. Nevertheless, fully understanding the implications of a rule-based administrative policy in ACAR is even more difficult than fully understanding the implications of an ARBAC policy, because in addition to considering interactions between interleaved sequences of changes by different administrators, one must consider possible chains of inferences using rules in each intermediate policy. This paper presents a symbolic analysis algorithm for answering atom-reachability queries for ACAR policies, i.e., for determining whether changes by specified administrators can lead to a policy in which some instance of a specified atom, called the goal, is derivable. To the best of our knowledge, this is the first analysis algorithm for a rule-based policy framework that considers changes to the rules as well as changes to the facts.

It is often desirable to be able to analyze rule-based policies with incomplete knowledge of the facts in the initial policy. For example, a database containing those facts might not exist yet (if the policy is part of a system that has not been deployed), or it might be unavailable to the policy engineer due to confidentiality restrictions. Even if some database of initial facts exists and is available to the policy engineer, more general analysis results that hold under limited assumptions about the initial facts are often preferable to results that hold for only one given set of initial facts, e.g., because the policy might be deployed in many systems with different initial facts.

There are two ways to handle this. In the deductive approach, the user specifies assumptions—in the form of constraints—about the initial policy, and the analysis algorithm determines whether the goal is reachable under those constraints. However, formulating appropriate constraints might be difficult, and might require multiple iterations of analysis and feedback. We adopt an abductive approach, in which the analysis algorithm determines conditions on the set of facts in the initial policy under which the given goal is reachable. This approach is inspired by Becker and Nanz's abductive policy analysis for a rule-based policy language [2], and our algorithm builds on their tabling-based policy evaluation algorithm. The main difference between their work and ours is that they analyze a fixed access control policy: they do not consider any administrative framework or any changes to the rules or facts in the access control policy. Also, they do not consider constructors or negation, while our policy language allows constructors and allows negation applied to extensional predicates.

Our analysis algorithm may diverge on some policies. This is expected, because Becker and Nanz's abductive algorithm (which solves a simpler problem) may diverge, and because reachability for ACAR is undecidable. Undecidability of this problem is a corollary of our proof in [7] that user-permission reachability is undecidable for ARBAC97 extended with parameters, since ARBAC97 policies can be encoded in ACAR in a straightforward way. Identifying classes of policies for which the algorithm is guaranteed to terminate is a direction for

future work; for now, we note that the algorithm terminates for the case studies we have considered so far, including a non-trivial fragment of a policy for a healthcare network. Also, the current version of the algorithm is incomplete (it "gives up" in some cases), but the algorithm can be extended to eliminate this incompleteness, following the approach sketched in Section 5.3; for now, we note that the current version of the algorithm already succeeds (does not "give up") for some non-trivial policies, including our healthcare network case study.

2 Policy Framework

Policy Language The policy language is a rule-based language with constructors (functors) and negation (denoted "`!`"). Predicates are classified as intensional or extensional. Intensional predicates are defined by rules. Extensional predicates are defined by facts. Constructors are used to construct terms representing operations, rules (being added or removed by administrative operations), parameterized roles, etc. The grammar ensures that negation is applied only to extensional predicates; this is why we distinguish intensional and extensional predicates. The grammar appears below. p ranges over predicates, c ranges over constructors (functors), and v ranges over variables. The grammar is parameterized by the sets of predicates, variables, and constructors; these sets may be finite or countable. Predicates and constructors start with a lowercase letter; variables start with an uppercase letter. Constants are represented as constructors with arity zero; the empty parentheses are elided. Subscripts *in* and *ex* are mnemonic for intensional and extensional, respectively. A term or atom is *ground* if it does not contain any variables. A substitution θ is *ground*, denoted ground(θ), if it maps variables to ground terms. A *policy* is a set of rules and facts.

$$
\begin{array}{lll}
\mathit{term} & ::= v \mid c(\mathit{term}^*) & \mathit{literal} ::= \mathit{atom}_{ex} \mid !\mathit{atom}_{ex} \mid \mathit{atom}_{in} \\
\mathit{atom}_{ex} & ::= p_{ex}(\mathit{term}^*) & \mathit{rule} ::= \mathit{atom}_{in} \texttt{ :- } \mathit{literal}^* \\
\mathit{atom}_{in} & ::= p_{in}(\mathit{term}^*) & \mathit{fact} ::= \text{ground instance of } \mathit{atom}_{ex}
\end{array}
$$

The distinguished predicate `permit(user, operation)` specifies permissions, including permissions for administrative operations, as discussed below.

Administrative Framework The administrative framework defines an API for modifying policies. Specifically, the operations in the API are `addRule`($rule$), `removeRule`($rule$), `addFact`($atom_{ex}$), and `removeFact`($atom_{ex}$). Let AdminOp = {`addRule`, `removeRule`, `addFact`, `removeFact`}. In addition, the framework defines how permission to perform those operations are controlled. These permissions are expressed using the `permit` predicate but given a special interpretation, as described below.

A *permission rule* is a rule whose conclusion has the form `permit(...)`. For an operation *op*, an *op* permission rule is a rule whose conclusion has the form `permit(..., op(...))` . An *administrative permission rule* is an *op* permission rule with $op \in$ AdminOp. In a well-formed policy, the argument to `addFact` and `removeFact` must be an extensional atom (it does not need to be ground).

A rule is *safe* if it satisfies the following conditions. (1) Every variable that appears in the conclusion outside the arguments of `addRule` and `removeRule` also appears in a positive premise. (2) Every variable that appears in a negative premise also appears in a positive premise. (3) In every occurrence of permit, the second argument is a constructor term, not a variable. (4) `addRule` and `removeRule` do not appear outside the second argument of `permit` in the conclusion. A policy is safe if all rules in the policy are safe.

Policy Semantics. The semantics $[\![P]\!]$ of a policy P is the least fixed-point of F_P, defined by $F_P(I) = \{a\theta \mid (a \texttt{ :- } a_1, \ldots, a_m, !b_1, \ldots, !b_n) \in P \wedge (\forall i \in [1..m] : \ a_i\theta \in I) \wedge (\forall i \in [1..n] : \ b_i\theta \notin I)\}$. To simplify notation, this definition assumes that the positive premises appear before the negative premises; this does not affect the semantics. Intuitively, $[\![P]\!]$ contains all atoms deducible from P. Atoms in the semantics are ground except that arguments of `addRule` and `removeRule` may contain variables. Limiting negation to extensional predicates ensures that F_P is monotonic. By the Knaster-Tarski theorem, the least fixed point of F_P can be calculated by repeatedly applying F_P starting from the empty set. Safety of the policy implies that, during this calculation, whenever $b_i \notin I$ is evaluated, b_i is ground; this simplifies the semantics of negation. We sometimes write $P \vdash a$ (read "P derives a") to mean $a \in [\![P]\!]$.

Fixed Administrative Policy. Our goal in this paper is to analyze a changing access control policy subject to a fixed administrative policy. Therefore, we consider policies that satisfy the *fixed administrative policy requirement*, which says that administrative permission rules cannot be added or removed, except that `addFact` administrative permission rules can be added. We allow this exception because it is useful in practice and can be accommodated easily.

We formalize this requirement as follows. A *higher-order* administrative permission rule is an administrative permission rule whose conclusion has the form $\texttt{permit}(\ldots, op(\texttt{permit}(\ldots, op'(\ldots))))$ with $op \in$ AdminOp and $op' \in$ AdminOp; in other words, it is a rule that permits addition and removal of administrative permission rules. A rule satisfies the fixed administrative policy requirement if either it is not a higher-order administrative permission rule or it is an administrative permission rule having the above form with $op =$ `addRule` and $op' =$ `addFact`. A policy satisfies this requirement if all of the rules in it do.

Even in a policy with no higher-order administrative permission rules, the available administrative permissions may vary, because addition and removal of other rules and facts may change the truth values of the premises of administrative permission rules.

Administrative Policy Semantics. The above semantics is for a fixed policy. We specify the semantics of administrative operations and administrative permissions by defining a transition relation T between policies, such that $\langle P, U\colon Op, P'\rangle \in T$ iff policy P permits user U to perform administrative operation Op thereby changing the policy from P to P'.

Rule R is *at least as strict as* rule R' if (1) R and R' have the same conclusion, and (2) the set of premises of R is a superset of the set of premises of R'.

$\langle P, U : \texttt{addRule}(R), P \cup \{R\}\rangle \in T$ if there exists a rule R' such that (1) R is at least as strict as R', (2) $P \vdash \texttt{permit}(U, \texttt{addRule}(R'))$, (3) $R \notin P$, (4) R satisfies the fixed administrative policy requirement, and (5) R satisfies the safe policy requirement. Note that R' may be a partially or completely instantiated version of the argument of `addRule` in the `addRule` permission rule used to satisfy condition (2); this follows from the definition of $\vdash$. Thus, an administrator adding a rule may specialize the "rule pattern" in the argument of `addRule` by instantiating some of the variables in it and by adding premises to it. We call the argument of `addRule` or `removeRule` a "rule pattern", even though it is generated by the same grammar as rules, to emphasize that it can be specialized in these ways, giving the administrator significant flexibility to customize the rules, without giving the administrator additional authority.

$\langle P, U : \texttt{removeRule}(R), P \setminus \{R\}\rangle \in T$ if there exists a rule R' such that R is as least as strict as R', $P \vdash \texttt{permit}(U, \texttt{removeRule}(R'))$, and $R \in P$.

$\langle (P, U : \texttt{addFact}(a), P \cup \{a\}\rangle \in T$ if $P \vdash \texttt{permit}(U, \texttt{addFact}(a))$ and $a \notin P$.

$\langle (P, U : \texttt{removeFact}(a), P \setminus \{a\}\rangle \in T$ if $P \vdash \texttt{permit}(U, \texttt{removeFact}(a))$ and $a \in P$.

Case Study: Healthcare Network. As a case study, we wrote a policy with about 50 rules for a healthcare network (HCN). The HCN policy defines a HCN policy officer (`hcn_po`) role that can add rules that give the facility policy officer (`facility_po`) role for each constituent healthcare facility appropriate permissions to manage the facility's policy. We consider policies for two kinds of facilities: hospitals and substance abuse facilities.

For example, the rule below allows the `facility_po` to add rules that allow the `hr_manager` to appoint a member of a workgroup as head of that workgroup:

```
permit(User, addRule(permit(HRM,
                            addFact(memberOf(Head, wgHead(WG, Fac))))
                     :- hasActivated(HRM, hr_manager(Fac)),
                        !memberOf(HRM, workgroup(WG, Fac, WGtype))
                        memberOf(Head, workgroup(WG, Fac, WGtype))))
  :- hasActivated(User, facility_po(Fac))
```

where $\texttt{memberOf}(U, R)$ holds if user U is a member of role R, $\texttt{hasActivated}(U, R)$ holds if U has activated R, `WG` is the workgroup name, `Fac` is the facility name, and `WGtype` is `team` or `ward`. The negative premise prevents a `hr_manager` from appointing a head for a workgroup to which he or she belongs.

At Stony Brook Hospital (`sb_hosp`), a member of `facility_po(sb_hosp)` might use this permission to add the following rule, which allows `hr_manager(sb_hosp)` to appoint a team member as the team head, with the additional premise that the user is a clinician at `sb_hosp` with any specialty `Spcty`.

```
permit(HRM, addFact(memberOf(Head, wgHead(WG, sb_hosp))))
  :- hasActivated(HRM, hr_manager(sb_hosp)),
     !memberOf(HRM, workgroup(WG, sb_hosp, team))
     memberOf(Head, workgroup(WG, sb_hosp, team)),
     memberOf(Head, clinician(sb_hosp, Spcty))
```

At Stony Brook Substance Abuse Facility (`sb_saf`), `facility_po(sb_saf)` might add a similar rule except with a stricter additional premise, requiring the team head to have specialty `psychiatry`.

3 Abductive Reachability

This section defines abductive atom-reachability queries and their solutions.

Let a and b denote atoms, L denote a literal, and $\vec{L}$ denote a sequence of literals. An atom a is *subsumed* by an atom b, denoted $a \preceq b$, iff there exists a substitution θ such that $a = b\theta$. For an atom a and a set A of atoms, let $[\![a]\!] = \{a' \mid a' \preceq a\}$ and $[\![A]\!] = \bigcup_{a \in A} [\![a]\!]$.

A *specification of abducible atoms* is a set A of extensional atoms. An atom a is abducible with respect to A if $a \in [\![A]\!]$.

A *goal* is an atom.

Given an initial policy P_0 and a set U_0 of users (the active administrators), the *state graph* for P_0 and U_0, denoted $\mathrm{SG}(P_0, U_0)$, contains policies reachable from P_0 by actions of users in U_0. Specifically, $\mathrm{SG}(P_0, U_0)$ is the least graph (N, E) such that (1) $P_0 \in N$ and (2) $\langle P, U : Op, P' \rangle \in E$ and $P' \in N$ if $P \in N \wedge U \in U_0 \wedge \langle P, U : Op, P' \rangle \in T$.

An *abductive atom-reachability query* is a tuple $\langle P_0, U_0, A, G_0 \rangle$, where P_0 is a policy (the initial policy), U_0 is a set of users (the users trying to reach the goal), A is a specification of abducible atoms, and G_0 is a goal. Informally, P_0 contains rules and facts that are definitely present in the initial state, and $[\![A]\!]$ contains facts that might be present in the initial state. Other facts are definitely not present in the initial state and, since we make the closed world assumption, are considered to be false.

A *ground solution* to an abductive atom-reachability query $\langle P_0, U_0, A, G_0 \rangle$ is a tuple $\langle \Delta, G \rangle$ such that Δ is a ground subset of $[\![A]\!]$, G is a ground instance of G_0, and $\mathrm{SG}(P_0 \cup \Delta, U_0)$ contains a policy P such that $P \vdash G$. Informally, a ground solution $\langle \Delta, G \rangle$ indicates that a policy P in which G holds is reachable from $P_0 \cup \Delta$ through administrative actions of users in U_0.

A *minimal-residue ground solution* to a query is a ground solution $\langle \Delta, G \rangle$ such that, for all $\Delta' \subset \Delta$, $\langle \Delta', G \rangle$ is not a ground solution to the query.

A *tuple disequality* has the form $\langle t_1 \ldots, t_n \rangle \neq \langle t'_1, \ldots, t'_n \rangle$, where the t_i and t'_i are terms.

A *complete solution* to an abductive atom-reachability query $\langle P_0, U_0, A, G_0 \rangle$ is a set S of tuples of the form $\langle \Delta, G, D \rangle$, where Δ is a set of atoms (not necessarily ground), G is an atom (not necessarily ground), and D is a set (interpreted as a conjunction) of tuple disequalities over the variables in Δ and G, such that (1) Soundness: S represents ground solutions to the query, i.e., $\bigcup_{s \in S} [\![s]\!] \subseteq S_{gnd}$, where $[\![\langle \Delta, G, D \rangle]\!] = \{\langle \Delta\theta, G\theta \rangle \mid \mathrm{ground}(\theta) \wedge D\theta = \mathrm{true}\}$ and S_{gnd} is the set of ground solutions to the query, and (2) Completeness: S represents all minimal-residue ground solutions to the query, i.e., $\bigcup_{s \in S} [\![s]\!] \supseteq S_{min\text{-}gnd}$, where $S_{min\text{-}gnd}$ is the set of minimal-residue ground solutions to the query.

Transition Rules:

(root) $(\{\langle G\rangle\} \uplus N, \mathit{Ans}, \mathit{Wait}) \to (N \cup N', \mathit{Ans}, \mathit{Wait})$
if $N' = \mathbf{generate}_P(G)$

(ans) $(\{n\} \uplus N, \mathit{Ans}, \mathit{Wait}) \to (N \cup N', \mathit{Ans}[G \mapsto \mathit{Ans}(G) \cup \{n\}], \mathit{Wait})$
if n is an answer node with index G
$\nexists n' \in \mathit{Ans}(G) : n \preceq n'$
$N' = \bigcup_{n'' \in \mathit{Wait}(G)} \mathbf{resolve}(n'', n)$

(goal_1) $(\{n\} \uplus N, \mathit{Ans}, \mathit{Wait}) \to (N \cup N', \mathit{Ans}, \mathit{Wait}[Q' \mapsto \mathit{Wait}(Q') \cup \{n\}])$
if n is a goal node with current subgoal Q
$\exists Q' \in \mathit{dom}(\mathit{Ans}) \ : \ Q \preceq Q'$
$N' = \bigcup_{n' \in \mathit{Ans}(Q')} \mathbf{resolve}(n, n')$

(goal_2) $(\{n\} \uplus N, \mathit{Ans}, \mathit{Wait}) \to (N \cup \{\langle Q\rangle\}, \mathit{Ans}[Q \mapsto \emptyset], \mathit{Wait}[Q \mapsto \{n\}])$
if n is a goal node with current subgoal Q
$\forall Q' \in \mathit{dom}(\mathit{Ans}) \ : \ Q \not\preceq Q'$

Auxiliary Definitions:

$\langle G; []; S; \vec{c}; R; \Delta\rangle \preceq \langle G; []; S'; \vec{c}'; R'; \Delta'\rangle$ iff $|\Delta| \geq |\Delta'| \wedge (\exists \theta \,.\, S = S'\theta \wedge \Delta \supseteq \Delta'\theta)$

for an answer node $n = \langle \_; []; Q'; \_; \_; \Delta'\rangle$, and Q'' and Δ'' fresh renamings of Q' and Δ',

$$\mathbf{resolve}(\langle G; [Q, \vec{Q}]; S; \vec{c}; R; \Delta\rangle, n) = \begin{cases} \{n'\} \text{ if unifiable}(Q, Q'') \\ \qquad \text{where } \theta = \text{mostGeneralUnifier}(Q, Q'') \\ \qquad\qquad n' = \langle G; \vec{Q}\theta; S\theta; [\vec{c}; n]; R; \Delta\theta \cup \Delta''\theta\rangle \\ \emptyset \quad \text{otherwise} \end{cases}$$

$$\mathbf{generate}_{P,A}(G) = \bigcup_{(Q \leftarrow \vec{Q}) \in P} \mathbf{resolve}(\langle G; [Q, \vec{Q}]; Q; []; Q \leftarrow \vec{Q}; \emptyset\rangle, \langle G; []; G; []; \_; \emptyset\rangle) \\ \cup (\text{if } G \in [\![A]\!] \text{ then } \{\langle G; []; G; []; \text{abduction}; \{G\}\rangle\} \text{ else } \emptyset)$$

Fig. 1. Becker and Nanz's algorithm for tabled policy evaluation with proof construction and abduction.

4 Becker and Nanz's Algorithm for Tabled Policy Evaluation with Proof Construction and Abduction

This section briefly presents Becker and Nanz's algorithm for tabled policy evaluation extended with proof construction and abduction [2]. This section is based closely on the presentation in their paper.

Becker and Nanz's algorithm appears in Figure 1. It defines a transition system, in which each state is a tuple of the form $(N, \mathit{Ans}, \mathit{Wait})$, where N is a set of nodes, Ans is an answer table, and Wait is a wait table, as defined below, and where the transitions between states are defined by the transition rules in the upper half of the figure. Disjoint union $\uplus$ is used for pattern matching on sets: N matches $N_1 \uplus N_2$ iff $N = N_1 \cup N_2$ and $N_1 \cap N_2 = \emptyset$.

A *node* is either a *root node* $\langle G\rangle$, where G is an atom, or a tuple of the form $\langle G; \vec{Q}; S; \vec{c}; R; \Delta\rangle$, where G is an atom called the *index* (the goal whose derivation this node is part of), $\vec{Q}$ is a list of subgoals that remain to be solved

in the derivation of the goal, S is the partial answer (the instance of G that can be derived using the derivation that this node is part of), $\vec{c}$ is the list of child nodes of this node, R is the rule used to derive this node from its children in the derivation of S, and Δ is the residue (the set of atoms abduced in this derivation). Note that, in the definition of **generate**, we use "abduction" as the name of the rule used to derive an abduced fact. If the list Q of subgoals is empty, the node is called an *answer node* with answer S. Otherwise, it is called a *goal node*, and the first atom in Q is its *current subgoal*. Each answer node is the root of a proof tree; goal nodes (representing queries) are not in proof trees. Selectors for components of nodes are: for $n = \langle G; \vec{Q}; S; \vec{c}; R; \Delta \rangle$, index$(n) = G$, subgoals$(n) = \vec{Q}$, pAns$(n) = S$, children$(n) = \vec{c}$, rule$(n) = R$, and residue$(n) = \Delta$.

An *answer table* is a partial function from atoms to sets of answer nodes. The set $Ans(G)$ contains all answer nodes for the goal G found so far.

A *wait table* is a partial function from atoms to sets of goal nodes. The set $Wait(G)$ contains all those nodes whose current subgoal is waiting for answers from $\langle G \rangle$. Whenever a new answer for $\langle G \rangle$ is produced, the computation of these waiting nodes is resumed.

The auxiliary definitions in the lower half of Figure 1 define the subsumption relation $\preceq$ on nodes and the **resolve** and **generate** functions. Intuitively, if $n \preceq n'$ (read "n is subsumed by n'"), then the answer node n provides no more information than n', so n can be discarded. **resolve**(n, n') takes a goal node n and an answer node n' and combines the current subgoal of n with the answer provided by n' to produce a new node with fewer subgoals. **generate**$_{P,A}(G)$ generates a set of nodes for a query $\langle G \rangle$ by resolving G against the rules of policy P, and by abducing G if G is abducible with respect to A. Constructors are not considered in [2], but the algorithm can handle them if the functions for matching and unification are extended appropriately. .

The *initial state* for goal G is $(\{\langle G \rangle\}, \{G \mapsto \emptyset\}, \{G \mapsto \emptyset\})$. A state S is a *final state* iff there is no state S' and such that $S \to S'$. Given a goal G, start with the initial state for G and apply transition rules repeatedly until a final state is reached. In that final state, $Ans(G)$ represents all derivable instances of G.

5 Analysis Algorithm

The algorithm has three phases. Phase 1 transforms the policy to eliminate `addRule` and `removeRule`. Phase 2 is a modified version of Becker and Nanz's tabling algorithm described above; it produces candidate solutions. Recall that their algorithm attempts to derive a goal from a fixed policy. We modify the tabling algorithm, and transform its input, to enable it to compute sets of policy updates (i.e., administrative operations) needed to derive the goal. However, modifying the tabling algorithm to incorporate a notion of time (i.e., a notion of the order in which updates to the policy are performed, and of the resulting sequence of intermediate policies) would require extensive changes, so we do not do that. Instead, we introduce a third phase that checks all conditions that depend on the order in which administrative operations are performed. These

conditions relate to negation, because in the absence of negation, removals are unnecessary, and additions can be done in any order consistent with the logical dependencies that the tabling algorithm already takes into account.

5.1 Phase 1: Elimination of `addRule` and `removeRule`

The policy P' obtained by elimination of `addRule` and `removeRule` from a policy P is not completely equivalent to P—in particular, the state graphs $\mathrm{SG}(P, U_0)$ and $\mathrm{SG}(P', U_0)$ differ, and some kinds of properties, such as availability of permissions, are not preserved. However, P' is equivalent to P in the weaker sense that using P' in place of P in an abductive atom-reachability query does not change the answer to the query. Informally, this is because the answer to such a query depends only on the "upper bounds" of the derivable facts in reachable policies, not on the exact sets of derivable facts in each reachable policy, and this transformation preserves those upper bounds.

Elimination of `removeRule`. The policy $\mathrm{elimRmRule}(P)$ is obtained from P by simply deleting all `removeRule` permission rules (recall that safety allows `removeRule` to appear only in the conclusion of such rules). This eliminates transitions that remove rules defining intensional predicates, and hence eliminates transitions that make intensional predicates smaller. Since negation cannot be applied to intensional predicates, making intensional predicates smaller never makes more facts (including instances of the goal) derivable. Therefore, every instance of the goal that is derivable in some policy reachable from P is derivable in some policy reachable from $\mathrm{elimRmRule}(P)$. Conversely, since $\mathrm{SG}(\mathrm{elimRmRule}(P_0), U_0)$ is a subgraph of $\mathrm{SG}(P_0, U_0)$, every instance of the goal that is derivable in some policy reachable from $\mathrm{elimRmRule}(P)$ is derivable in some policy reachable from P. Therefore, the elimRmRule transformation does not affect the answer to abductive atom-reachability queries.

Elimination of `addRule`. We eliminate `addRule` by replacing `addRule` permission rules (recall that safety allows `addRule` to appear only in the conclusion of such rules) with new rules that use `addFact` to "simulate" the effect of `addRule`. Specifically, the policy $\mathrm{elimAddRule}(P)$ is obtained from P as follows. Let R be an `addRule` permission rule $\texttt{permit}(U, \texttt{addRule}(L \texttt{ :- } \vec{L}_1)) \texttt{ :- } \vec{L}_2$ in P. Rule R is replaced with two rules. One rule is the rule pattern in the argument of `addRule`, extended with an additional premise using a fresh extensional predicate $\texttt{aux}_R$ that is unique to the rule: $L \texttt{ :- } \vec{L}_1, \texttt{aux}_R(\vec{X})$, where the vector of variables $\vec{X}$ is $\vec{X} = \mathrm{vars}(L \texttt{ :- } \vec{L}_1) \cap (\mathrm{vars}(\{U\}) \cup \mathrm{vars}(\vec{L}_2))$. The other is an `addFact` permission rule that allows the user to add facts to this new predicate: $\texttt{permit}(U, \texttt{addFact}(\texttt{aux}_R(\vec{X}))) \texttt{ :- } \vec{L}_2$. The auxiliary predicate $\texttt{aux}_R$ keeps track of which instances of the rule pattern have been added. Recall from Section 2 that users are permitted to instantiate variables in the rule pattern when adding a rule. Note that users must instantiate variables that appear in the rest of the `addRule` permission rule, i.e., in $\mathrm{vars}(\{U\}) \cup \mathrm{vars}(\vec{L}_2)$, because if those variables are not grounded, the `permit` fact necessary to add the rule will not be derivable using rule R. Therefore, each fact in $\texttt{aux}_R$ records the values of those variables.

In other words, an `addRule` transition t in $\mathrm{SG}(P_0, U_0)$ in which the user adds an instance of the rule pattern with $\vec{X}$ instantiated with $\vec{c}$ is "simulated" in $\mathrm{SG}(\mathrm{elimAddRule}(P_0), U_0)$ by an `addFact` transition t that adds $\mathtt{aux}_R(\vec{c})$.

Note that $\mathrm{SG}(P_0, U_0)$ also contains transitions t' that are similar to t except that the user performs additional specialization of the rule pattern by instantiating additional variables in the rule pattern or adding premises to it. Those transitions are eliminated by this transformation, in other words, there are no corresponding transitions in $\mathrm{SG}(\mathrm{elimAddRule}(P_0), U_0)$. This is sound, because those transitions lead to policies in which the intensional predicate p that appears in literal L (i.e., L is $p(\ldots)$) is smaller, and as argued above, since negation cannot be applied to intensional predicates, eliminating transitions that lead to smaller intensional predicates does not affect the answer to abductive atom-reachability queries.

Applying this transformation to a policy satisfying the fixed administrative policy requirement produces a policy containing no higher-order administrative permission rules.

From the above arguments, we conclude: For every policy P_0, set U_0 of users, and atom a not in excludedAtoms, $\mathrm{SG}(P_0, U_0)$ contains a policy P with $a \in [\![P]\!]$ iff $\mathrm{SG}(\mathrm{elimAddRule}(\mathrm{elimRmRule}(P_0)), U_0)$ contains a policy P' with $a \in [\![P']\!]$, where excludedAtoms is the set of atoms of the form $\mathtt{permit}(\ldots, \mathtt{addRule}(\ldots))$, $\mathtt{permit}(\ldots, \mathtt{removeRule}(\ldots))$, $\mathtt{aux}_R(\ldots)$, or $\mathtt{permit}(\ldots, \mathtt{addFact}(\mathtt{aux}_R(\ldots)))$. From this, it is easy to show that answers to abductive atom-reachability queries are preserved by this transformation. Subsequent phases of the algorithm analyze the policy $\mathrm{elimAddRule}(\mathrm{elimRmRule}(P_0))$.

5.2 Phase 2: Tabled Policy Evaluation

Phase 2 is a modified version of Becker and Nanz's algorithm. It considers three ways to satisfy a positive subgoal: through an inference rule, through addition of a fact (using an `addFact` permission rule), and through abduction (i.e., by assumption that the subgoal holds in the initial policy and still holds when the rule containing it as a premise is evaluated).

To allow the algorithm to explore addition of facts as a way to satisfy positive subgoals, without directly modifying the algorithm, we transform `addFact` permission rules into ordinary inference rules. Specifically, each `addFact` permission rule $\mathtt{permit}(U, \mathtt{addFact}(a))$ `:-` $\vec{L}$ is replaced with the rule a `:-` $\vec{L}, \mathtt{u0}(U)$. The transformation also introduces a new extensional predicate `u0` and, for each $u \in U_0$, the fact $\mathtt{u0}(u)$ is added to the policy. This transformation changes the meaning of the policy: the transformed rule means that a necessarily holds when $\vec{L}$ holds, while the original `addFact` permission rule means that a might (or might not) be added by an administrator when $\vec{L}$ holds. This difference is significant if a appears negatively in a premise of some rule. This change in meaning is acceptable in phase 2, because phase 2 does not attempt to detect conflicts between negative subgoals and added facts. This change in the meanings of rules used in phase 2 does not affect the detection of such conflicts in phase 3.

The algorithm considers two ways to satisfy a negative subgoal: through removal of a fact (using a `removeFact` permission rule) and through abduction (i.e., by assumption that the negative subgoal holds in the initial policy and still holds when the rule containing it as a premise is evaluated).

To allow the algorithm to explore removal of facts as a way to satisfy negative subgoals, `removeFact` permission rules are transformed into ordinary inference rules with negative conclusions. Specifically, each `removeFact` permission rule $\texttt{permit}(U, \texttt{removeFact}(a))$ `:-` $\vec{L}$ is replaced with the rule $!a$ `:-` $\vec{L}, \texttt{u0}(U)$.

Let elimAddRmFact(P) denote the policy obtained from P by transforming `addFact` and `removeFact` rules as described above. An *administrative node* (or "admin node", for short) is a node n such that rule(n) is a transformed `addFact` or `removeFact` permission rule. isAdmin(n) holds iff n is an administrative node.

The algorithm can abduce a negative extensional literal $!a$ when this is consistent with the initial policy, in other words, when a is not in P_0. To enable this, in the definition of **generate**, we replace "$G \in [\![A]\!]$" with "$G \in [\![A]\!] \vee (\exists a \in \text{Atom}_{ex}.\ a \notin P_0 \wedge G \text{ is } !a)$", where Atom_{ex} is the set of extensional atoms. If a is not ground, disequalities in d_{init} in phase 3 will ensure that the solution includes only instances of a that are not in P_0.

The tabling algorithm treats the negation symbol "!" as part of the predicate name; in other words, it treats p and $!p$ as unrelated predicates. Phase 3 interprets "!" as negation and checks appropriate consistency conditions.

The tabling algorithm explores all derivations for a goal except that the subsumption check in transition rule (ans) in Figure 1 prevents use of the derivation represented by answer node n from being added to the answer table and thereby used as a sub-derivation of a larger derivation if the partial answer in n is subsumed by the partial answer in an answer node n' that is already in the answer table. However, the larger derivation that uses n' as a derivation of a subgoal might turn out to be infeasible (i.e., have unsatisfiable ordering constraints) in phase 3, while the larger derivation that uses n as a derivation of that subgoal might turn out to be feasible. We adopt the simplest approach to overcome this problem: we replace the subsumption relation $\preceq$ in transition rule (ans) with the α-equality relation $=_\alpha$, causing the tabling algorithm to explore all derivations of goals. α-equality means equality modulo renaming of variables that do not appear in the top-level goal G_0.

An undesired side-effect of this change is that the algorithm may get stuck in a cycle in which it repeatedly uses some derivation of a goal as a sub-derivation of a larger derivation of the same goal. Exploring the latter derivation is unnecessary, because it will be subjected in phase 3 to the same or more constraints as the former derivation. Therefore, we modify the definition of **resolve** as follows, so that the algorithm does not generate a node n' corresponding to the latter derivation: we replace unifiable(Q, Q'') with unifiable(Q, Q'') $\wedge$ noCyclicDeriv(n'), where

$$\begin{aligned}\text{noCyclicDeriv}(n') = {} & \nexists d \in \text{proof}(n').\ \text{isAns}(d) \\ & \wedge \langle \text{index}(d), \text{pAns}(d), \text{residue}(d) \rangle =_\alpha \langle \text{index}(n'), \text{pAns}(n'), \text{residue}(n') \rangle\end{aligned}$$

where the *proof* of a node n, denoted proof(n), is the set of nodes in the proof graph rooted at node n, i.e., $\text{proof}(n) = \{n\} \cup \bigcup_{n' \in \text{children}(n)} \text{proof}(n')$, and

where isAns(n) holds iff n is an answer node. noCyclicDeriv(n') does not check whether rule(n') = rule(d) or subgoals(n') = subgoals(d), because exploration of n' is unnecessary, by the above argument, regardless of the values of rule(n') and subgoals(n').

We extend the algorithm to store the *partial answer substitution*, denoted $\theta_{\text{pa}}(n)$, in each node n. This is the substitution that, when applied to index(n), produces pAns(n). In the **generate** function, the θ_{pa} component in both nodes passed to **resolve** is the empty substitution. In the **resolve** function, $\theta_{\text{pa}}(n')$ is $\theta \circ \theta_{\text{fr}} \circ \theta_{\text{pa}}(n_1)$, where θ_{fr} is the substitution that performs the fresh renaming of Q' and Δ', and n_1 denotes the first argument to **resolve**.

In summary, given an abductive atom-reachability query of the form in Section 3, phase 2 applies the tabling algorithm, modified as described above, to the policy elimAddRmFact(elimAddRule(elimRmRule(P_0))) with the given goal G_0 and specification A of abducible atoms.

5.3 Phase 3: Ordering Constraints

Phase 3 considers constraints on the execution order of administrative operations. The ordering must ensure that, for each administrative node or goal node n, (a) each administrative operation n' used to derive n occurs before n (this is a "dependence constraint") and its effect is not undone by a conflicting operation that occurs between n' and n (this is an "interference-freedom constraint"), and (b) each assumption about the initial policy on which n relies is not undone by an operation that occurs before n (this is an "interference-freedom constraint").

The overall ordering constraint is represented as a Boolean combination of labeled ordering edges. A labeled ordering edge is a tuple $\langle n, n', D\rangle$, where the label D is a conjunction of tuple disequalities or false, with the interpretation: n must precede n', unless D holds. if D holds, then n and n' operate on distinct atoms, so they commute, so the relative order of n and n' is unimportant.

Pseudocode for phase 3 appears in Figures 2 and 3. The algorithm generates the overall ordering constraint, puts the Boolean expression in DNF, and checks, for each clause c, whether the generated ordering constraints can be satisfied, i.e., whether they are acyclic. If so, the disequalities labeling the ordering constraints do not need to be included in the solution. However, if the generated ordering constraints are cyclic, then the algorithm removes a minimal set of ordering constraints to make the remaining ordering constraints acyclic, and includes the disequalities that label the removed ordering constraints in the solution. After constraints have been checked, negative literals are removed from the residue; this is acceptable, because the problem definition asks for a representation of only minimal-residue ground solutions, not all ground solutions The algorithm can easily be extended to return a *plan* (a sequence of administrative operations that leads to the goal) for each solution.

Repeated Administrative Operations. Tabling re-uses nodes, including, in our setting, administrative nodes. This makes the analysis more efficient and avoids unnecessary repetition of administrative operations in plans. However, in some cases, administrative operations need to be repeated; for example, it might

be necessary to add a fact, remove it, and then add it again, in order to reach the goal. The current version of our algorithm cannot generate plans with repeated administrative operations, but it does identify when repeated operations might be necessary, using the function mightNeedRepeatedOp, and returns a message indicating this. Specifically, mightNeedRepeatedOp(c, n_g) returns true if some node n in c is a child of multiple nodes in proof(n_g); in such cases, it might be necessary to replace n with multiple nodes, one for each parent, in order to satisfy the ordering constraints. To achieve this, the algorithm can be modified so that, if mightNeedRepeatedOp returns true, the algorithm re-runs phases 2 and 3 but this time constructs new answer nodes, instead of re-using tabled answers, for the nodes identified by mightNeedRepeatedOp as possibly needing to be repeated.

5.4 Implementation and Experience.

We implemented the analysis algorithm in 5000 lines of OCaml and applied it to part of the policy P_{HCN} for our healthcare network case study with 30 administrative permission rules. We included facts about a few prototypical users in P_{HCN}: `fpo1`, a member of `facility_po(sb_hosp)`; `clin1`, a clinician at `sb_hosp`; and `user1`, a user with no roles. A sample abductive atom-reachability query that we evaluated has $P_0 = P_{\text{HCN}}$, $U_0 = \{$`fpo1`, `user1`$\}$, $A = \{$`memberOf(User, workgroup(WG, sb_hosp, team))`$\}$, and G_0 = `memberOf(GoalUser, wgHead(cardioTeam, sb_hosp))`. The analysis takes about 1.5 seconds, generates 2352 nodes, and returns five solutions. For example, one solution has partial answer `memberOf(GoalUser, wgHead(cardioTeam, sb_hosp))`, residue {`memberOf(GoalUser, workgroup(cardioTeam, sb_hosp, team))`}, and tuple disequality $\langle$`GoalUser`$\rangle \neq \langle$`fpo1`$\rangle$. The disequality reflects that `fpo1` can appoint himself to the `hr_manager(sb_hosp)` role, can then appoint himself and other users as members of `cardioTeam`, and can then appoint other users as team head, but cannot then appoint himself as team head, due to the negative premise in the sample rules at the end of Section 2.

References

1. Becker, M.Y.: Specification and analysis of dynamic authorisation policies. In: Proc. 22nd IEEE Computer Security Foundations Symposium (CSF). pp. 203–217 (2009)
2. Becker, M.Y., Nanz, S.: The role of abduction in declarative authorization policies. In: Proc. 10th International Symposium on Practical Aspects of Declarative Languages (PADL 2008). pp. 84–99. Springer-Verlag (2008)
3. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. ACM Transactions on Information and System Security 13(3) (2010)
4. Jha, S., Li, N., Tripunitara, M., Wang, Q., Winsborough, W.: Towards formal verification of role-based access control policies. IEEE Transactions on Dependable and Secure Computing 5(4), 242–255 (2008)
5. Li, N., Tripunitara, M.V.: Security analysis in role-based access control. ACM Transactions on Information and System Security 9(4), 391–420 (Nov 2006)

$solutions = \emptyset$
for each node $n_g \in Ans(G)$
  // consistency constraint: disequalities that ensure consistency of initial state,
  // i.e., positive literals are distinct from negative literals.
  $d_{\text{init}} = \bigwedge\{\text{args}(a) \neq \text{args}(b) \mid a \in \text{facts}(P_0) \cup \text{residue}(n_g) \wedge !b \in \text{residue}(n_g) \wedge \text{unifiable}(a, b)\}$
  if $\neg$satisfiable(d_{init})
    continue
  endif
  $O = \text{orderingConstraints}(n_g)$
  if ($\exists$ clause c in O. the ordering constraints in c are acyclic)
    // the ordering constraints for n_g are satisfiable without imposing disequalities.
    // intersect residue with Atom_{ex} (the extensional atoms) to remove negative literals.
    $solutions = solutions \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d_{\text{init}} \rangle\}$
  else
    // the ordering constraints for n_g are not satisfiable in general, but might
    // be satisfiable if disequalities are imposed to ensure that some
    // administrative operations operate on distinct atoms and therefore commute.
    for each clause c in O
      if mightNeedRepeatedOp(c, n_g)
        // the current version of the algorithm does not support repeated operations
        return "repeated operations might be needed"
      endif
      $D_{\text{ord}} = \emptyset$
      // c is a conjunction (treated as a set) of labeled ordering constraints.
      // remove some ordering constraints F from c to make the remaining ordering
      // constraints acyclic, and insert in D_{ord} the conjunction d of d_{init} and the
      // disequalities labeling the removed ordering constraints.
      Cyc = set of all cycles in ordering constraints for clause c
      $FAS = \{F \mid F$ contains one edge selected from each cycle in $Cyc\}$
      // $smFAS$ is the set of $\subseteq$-minimal feedback arc sets (FASs) for clause c
      $smFAS = \{F \in FAS \mid \nexists F' \in FAS.\ F' \subset F\}$
      for each F in $smFAS$
        $d = d_{\text{init}} \wedge \bigwedge\{d' \mid \langle n_1, n_2, d' \rangle \in F\}$
        if satisfiable(d) $\wedge \neg(\exists d' \in D_{\text{ord}}.\ d' \subseteq d)$
          $D_{\text{ord}} = D_{\text{ord}} \cup \{d\}$
        endif
      endfor
      $solutions = solutions \cup \{\langle \text{pAns}(n_g), \text{residue}(n_g) \cap \text{Atom}_{ex}, d \rangle \mid d \in D_{\text{ord}}\}$
    endfor
  endif
endfor
return *solutions*

Fig. 2. Pseudo-code for Phase 3.

6. Sandhu, R., Bhamidipati, V., Munawer, Q.: The ARBAC97 model for role-based administration of roles. ACM Transactions on Information and Systems Security 2(1), 105–135 (Feb 1999)

function orderingConstraints(n_g)
$\theta = \theta_{\text{pa}}(n_g)$
// dependence constraint: an admin node n_s that supports n must occur before n.
$O_{\text{dep}} = \bigwedge\{\langle n_s, n, \text{false}\rangle \mid n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_s \in \text{adminSupport}(n)\}$
// all of the constraints below are interference-freedom constraints.
// a `removeFact` node n_r that removes a supporting initial fact of a node n must occur
// after n.
$O_{\text{rm-init}} = \bigwedge\{\langle n, n_r, \text{args}(a)\theta \neq \text{args}(\text{pAns}(n_r))\theta\rangle \mid$
$n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_r \in \text{proof}(n_g) \wedge \text{isRmFact}(n_r)$
$\wedge\, n \neq n_r \wedge a \in \text{supportingInitFact}(n) \wedge \text{unifiable}(!a, \text{pAns}(n_r))\}$
// an `addFact` node n_a that adds a fact whose negation is a supporting initial fact
// of a node n must occur after n.
$O_{\text{add-init}} = \bigwedge\{\langle n, n_a, \text{args}(a)\theta \neq \text{args}(\text{pAns}(n_a))\theta\rangle \mid$
$n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_a \in \text{proof}(n_g) \wedge \text{isAddFact}(n_a)$
$\wedge\, n \neq n_a \wedge !a \in \text{supportingInitFact}(n) \wedge \text{unifiable}(a, \text{pAns}(n_a))\}$
// an `addFact` node n_a that adds a supporting removed fact of a node n must occur
// either before the removal of that fact or after n.
$O_{\text{add-rmvd}} =$
$\bigwedge\{\langle n_a, n_r, \text{args}(\text{pAns}(n_a))\theta \neq \text{args}(\text{pAns}(n_r))\theta\rangle \vee \langle n, n_a, \text{args}(\text{pAns}(n_a))\theta \neq \text{args}(\text{pAns}(n_r))\theta\rangle \mid$
$n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_r \in \text{adminSupport}(n) \wedge \text{isRmFact}(n_r)$
$\wedge\, n_a \in \text{proof}(n_g) \wedge \text{isAddFact}(n_a) \wedge n \neq n_a \wedge \text{unifiable}(!\text{pAns}(n_a), \text{pAns}(n_r))\}$
// a `removeFact` node n_r that removes a supporting added fact of a node n must occur
// either before the addition of that fact or after n
$O_{\text{rm-added}} =$
$\bigwedge\{\langle n_r, n_a, \text{args}(\text{pAns}(n_a))\theta \neq \text{args}(\text{pAns}(n_r))\theta\rangle \vee \langle n, n_r, \text{args}(\text{pAns}(n_a))\theta \neq \text{args}(\text{pAns}(n_r))\theta\rangle \mid$
$n \in \text{proof}(n_g) \wedge (\text{isAdmin}(n) \vee n = n_g) \wedge n_a \in \text{adminSupport}(n) \wedge \text{isAddFact}(n_a)$
$\wedge\, n_r \in \text{proof}(n_g) \wedge \text{isRmFact}(n_r) \wedge n \neq n_r \wedge \text{unifiable}(!\text{pAns}(n_a), \text{pAns}(n_r))\}$
// conjoin all ordering constraints and convert the formula to disjunctive normal form.
$O = \text{DNF}(O_{\text{dep}} \wedge O_{\text{rm-init}} \wedge O_{\text{add-init}} \wedge O_{\text{add-rmvd}} \wedge O_{\text{rm-added}})$
// for each clause c of O, merge labeled ordering constraints for the same pair of nodes.
for each clause c in O
while there exist n_1, n_2, D, D' such that c contains $\langle n_1, n_2, D\rangle$ and $\langle n_1, n_2, D'\rangle$
replace $\langle n_1, n_2, D\rangle$ and $\langle n_1, n_2, D'\rangle$ with $\langle n_1, n_2, D \wedge D'\rangle$ in c
endwhile
endfor
return O

$\text{args}(a)$ = a tuple containing the arguments of atom a
$\text{support}(n) = \{n' \in \text{proof}(n) \mid \text{isAns}(n') \wedge n' \neq n$
$\neg\exists n_a.\text{isAdmin}(n_a) \wedge \text{descendant}(n, n_a) \wedge \text{descendant}(n_a, n')\}$
$\text{adminSupport}(n) = \{n' \in \text{support}(n) \mid \text{isAdmin}(n')\}$
$\text{supportingInitFact}(n) = \{\text{pAns}(n') \mid n' \in \text{support}(n)$
$\wedge\, (\text{rule}(n') \in \text{facts}(P_0) \vee \text{rule}(n') = \text{abduction})\}$

Fig. 3. Ordering constraints for an answer node n_g.

7. Stoller, S.D., Yang, P., Gofman, M., Ramakrishnan, C.R.: Symbolic reachability analysis for parameterized administrative role based access control. Computers & Security 30(2-3), 148–164 (March-May 2011)