The Role of Abduction in Declarative Authorization Policies

Moritz Y. Becker[1] and Sebastian Nanz[2]

[1] Microsoft Research, Cambridge, CB3 0FB, UK
moritzb@microsoft.com
Tel: +44-1223-479826 Fax: +44-1223-479999
[2] Informatics and Mathematical Modelling
Technical University of Denmark, Denmark
nanz@imm.dtu.dk

Abstract. Declarative authorization languages promise to simplify the administration of access control systems by allowing the authorization policy to be factored out of the implementation of the resource guard. However, writing a correct policy is an error-prone task by itself, and little attention has been given to tools and techniques facilitating the analysis of complex policies, especially in the context of access denials. We propose the use of abduction for policy analysis, for explaining access denials and for automated delegation. We show how a deductive policy evaluation algorithm can be conservatively extended to perform abduction on Datalog-based authorization policies, and present soundness, completeness and termination results.

Keywords: access control, abduction, authorization language, Datalog

1 Introduction

Authorization is the task of granting or denying access to a system's resources according to a policy. Traditionally, authorization policies have been implemented by access control lists (ACL) or capabilities provided by the operating system, sometimes augmented by groups or roles. However, there are many applications for which these mechanisms are too inflexible, not sufficiently expressive and provide the wrong level of abstraction. For example, access to electronic health records is regulated by a huge number of laws that are both complex and prone to change. Decentralized applications such as grid systems require support for delegation of authority and attribute-based constraints.

Such requirements have led to the development of trust management systems and declarative authorization languages for flexible, expressive application-level access control (e.g. [1,2,3,4,5]). An authorization policy is then written as a set of rules that are both human readable and machine enforceable. This approach aims to increase the usability and scalability of access control systems: policies written in such languages are more concise and have a lower viscosity than

ACLs, and provide a much higher level of abstraction, thus facilitating a closer representation of the intended policy.

However, comprehending and predicting the consequences of a policy is difficult, as policies can be complex and contain hundreds of rules, and each access grant is based on the construction of a logical proof of compliance with respect to this policy. Thus, writing a correct policy is still a highly error-prone task. We conjecture that the current lack of tools for analyzing policies remains a major obstacle to a wider adoption of authorization languages.

In this paper, we develop algorithms for analyzing the consequences of declarative authorization policies. Many existing authorization languages are based on negation-free Datalog or are translated into Datalog for evaluating access requests. (A Datalog clause is a first-order definite Horn clause without function symbols.) Datalog is sufficiently expressive for a wide range of policies, including delegation, which requires recursion. Furthermore, Datalog is decidable and can be evaluated efficiently. Hence, to maximize generality, our algorithms work on policies specified in Datalog.

In particular, we focus on the tasks of *explaining* access grants and access denials. In the former case, the question we are trying to answer is "*why is a given request granted?*". It is easy to see that the proof graph contains exactly the necessary information for constructing the (possibly textual) explanation. The basic evaluation algorithm that is used for deciding access requests can be easily extended to construct a copy of the proof graph during evaluation.

In the case of access denial, the question is "*which authorization facts or credentials were missing that would have led to an access grant?*". This turns out to be a harder question, as the failed partial proof does not contain enough information to answer it. Moreover, there are in general infinitely many different answers, but often only finitely many "interesting" or "useful" ones. We propose to apply *abductive* techniques for finding the set of meaningful answers. Abduction [6] is a reasoning paradigm that has been used for planning, fault diagnosis and other areas in AI, but has not previously been considered for analyzing authorization policies. Many algorithms have been developed for various variants of abduction (see [7] for an extensive survey); in this paper, we show that a deductive evaluation algorithm that is used for deciding access requests can be conservatively extended to perform abduction of authorization facts and credentials. Thus we show that existing implementations of Datalog-based authorization engines can be leveraged and extended with little effort to facilitate this kind of analysis. Moreover, we show that this algorithm can be used for multiple purposes: (1) as the basis of a tool helping security administrators to write and to debug policies, (2) for providing users with an answer in the case of an access denial that is more helpful than a mere "no", and (3) to compute sets of missing credentials in automated distributed delegation scenarios.

The remainder of the paper is structured as follows: Section 2 first presents a non-deterministic terminating algorithm for evaluating policies. The algorithm is extended to construct proof graphs. A second extension is developed that computes sets of missing facts that, if added, would lead to a positive access

decision. Section 3 presents techniques for guaranteeing that the algorithm terminates. Section 4 discusses three application scenarios to illustrate the different ways in which the algorithm could be used. In Section 5 we discuss related work and conclude. A technical report [8] contains full proofs.

2 Adding Abduction to Policy Evaluation

The basic authorization problem consists of deciding whether an access request complies with an authorization policy and a set of credentials. For Datalog-based policies, this amounts to deductive query evaluation. Tabling resolution algorithms are proposed in [9] and [5] for evaluating queries against Datalog-based authorization policies. These algorithms are easy to implement and are guaranteed to terminate due to tabling [10,11].

Here we provide a generalization of these algorithms, presented as state transition systems. The non-deterministic presentation lends itself to parallel implementations; moreover, it also leads to simpler soundness, completeness and termination proofs. Section 2.2 instantiates this generalized tabling scheme to a purely deductive policy evaluation algorithm. The evaluation algorithms in [9] and [5] can be seen as straightforward deterministic implementations of this general scheme. Section 2.3 illustrates a simple extension of the first one, which facilitates the construction of proof graphs, e.g. for explaining positive access decisions. Finally, Section 2.4 extends it further to perform *abduction*, which computes the *dual* of the basic authorization problem, namely the sets of facts or credentials which, according to the policy, would grant an authorization request. As shown in Section 4, this algorithm can be applied to explain access denials, to analyze policies and to provide automated distributed delegation.

2.1 An Extensible Scheme for Policy Evaluation

Preliminaries. We use the terms *groundness*, *substitution*, *unifier*, *most general unifier (mgu)* and *(fresh) variable renaming* in their standard meanings. We assume a denumerable set of variables $\mathcal{X}$ and a first-order signature with a countable (possibly infinite) set of constants $\mathcal{C}$ and a finite set of predicate names (but no function symbols). An *atom* P consists of a predicate name applied to an ordered list of terms, each of which is either a variable or a constant. *Clauses* are of the form $P_0 \leftarrow \vec{P}$. The atom P_0 is referred to as the *head*, and the (possibly empty) finite sequence of atoms $\vec{P}$ as the *body* of the clause. A clause with an empty body is also called a *fact*. A *policy* $\mathcal{P}$ is a finite set of clauses.

The semantics of a policy $\mathcal{P}$ is given by the least fixed point of the *immediate consequence operator* $T_{\mathcal{P}}$ [12]:

$$T_{\mathcal{P}}(I) = \{P_0\theta \ : \ (P_0 \leftarrow P_1, \ldots, P_n) \in \mathcal{P},\ P_i\theta \in I \text{ for each } i,\ P_0\theta \text{ ground}\}$$

We denote the least fixed point of $T_{\mathcal{P}}$ by $T_{\mathcal{P}}^{\omega}(\emptyset)$. Intuitively, it contains all ground atoms that are deducible from the policy. The most general unifier of atoms P

and Q is denoted by $mgu(P, Q)$. We say that P *is subsumed by* Q (also written $P \preceq Q$) iff $P = Q\theta$ for some substitution θ.

In our examples, we write variables in *italics*, constants in `typewriter` font, and predicate names in sans serif.

An authorization policy defines authorization-relevant predicates such as canRead, canWrite etc. Upon an access request, the resource guard issues a query (e.g. "canRead(Alice,Foo)?") to be evaluated against the policy. Usually, the policy is composed of the locally stored policy plus a set of facts obtained from (user-submitted or fetched) *credentials* that may support the request. We will later show examples of policies and their uses.

Description of the Scheme. The algorithms described in the following subsections are instantiations of a state transition system that processes nodes of the following form:

Definition 2.1 (Nodes). A *node* is either a *root node* $\langle P \rangle$ where P is an atom, or a *tuple* (with at least 3 fields) of the form $\langle P; \vec{Q}; S; ... \rangle$, where the atom P is called the *index*, the (possibly empty) sequence of atoms $\vec{Q}$ the *subgoals*, and the atom S the *partial answer*. If the list of subgoals $\vec{Q}$ is empty, a node is called an *answer node* with *answer* S. Otherwise it is called a *goal node*, and the first atom in $\vec{Q}$ is its *current subgoal*.

Intuitively, the list of subgoals $\vec{Q}$ contains the atoms that still have to be solved for the goal P. The subgoals are solved from left to right, hence the head of the list is the current subgoal. The current subgoal can be resolved against another answer node, which may entail instantiations of variables which will narrow down the partial answer S. The partial answer may eventually become a proper answer if all subgoals have been solved.

Note also that tuples may have more than three fields, which allows us to instantiate the algorithm scheme to perform various computational tasks. In its standard form, it implements ordinary deduction as outlined in Section 2.2. We may add an additional field $\vec{n}$ containing the nodes which justify the derivation of the current node; this will allow us to reconstruct proof graphs in Section 2.3. Lastly, we will introduce a field Δ containing atoms which were just assumed to hold when deriving the current node, and thus yield an abductive algorithm in Section 2.4.

Furthermore, the algorithm makes use of two tables:

Definition 2.2 (Answer and Wait Tables). An *answer table* is a partial function from atoms to sets of answer nodes. A *wait table* is a partial function from atoms to sets of goal nodes.

We denote the answer and wait tables in the algorithm by *Ans* and *Wait*, respectively. The set $Ans(P)$ contains all answer nodes pertaining to the goal $\langle P \rangle$ found so far. The set $Wait(P)$ contains all those nodes whose current subgoal is waiting for answers from $\langle P \rangle$. Whenever a new answer for $\langle P \rangle$ is produced, the computation of these waiting nodes is resumed.

$$\text{(root)}\quad (\{\langle P\rangle\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow (\mathcal{N} \cup \mathcal{N}', \mathit{Ans}, \mathit{Wait})$$
$$\text{if}\quad \mathcal{N}' = \mathbf{generate}_{\mathcal{P}}(P)$$

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

$$\text{(goal}_1\text{)}\quad (\{n\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow (\mathcal{N} \cup \mathcal{N}', \mathit{Ans}, \mathit{Wait}[Q' \mapsto \mathit{Wait}(Q') \cup \{n\}])$$
$$\text{if}\quad n \text{ is a goal node with current subgoal } Q$$
$$\exists\, Q' \in \mathit{dom}(\mathit{Ans})\ :\ Q \preceq Q'$$
$$\mathcal{N}' = \textstyle\bigcup_{n' \in \mathit{Ans}(Q')} \mathbf{resolve}(n, n')$$

$$\text{(goal}_2\text{)}\quad (\{n\} \uplus \mathcal{N}, \mathit{Ans}, \mathit{Wait}) \rightarrow (\mathcal{N} \cup \{\langle Q\rangle\}, \mathit{Ans}[Q \mapsto \emptyset], \mathit{Wait}[Q \mapsto \{n\}])$$
$$\text{if}\quad n \text{ is a goal node with current subgoal } Q$$
$$\forall\, Q' \in \mathit{dom}(\mathit{Ans})\ :\ Q \not\preceq Q'$$

Table 1. Generic tabling algorithm

The algorithm is given in Table 1 as a transition system defined by a relation $\rightarrow$ on states of the following form:

Definition 2.3 (States). A *state* is a triple $(\mathcal{N}, \mathit{Ans}, \mathit{Wait})$ where $\mathcal{N}$ is a set of nodes, *Ans* is an answer table, and *Wait* is a wait table.

A state of the form $(\{\langle P\rangle\}, \{P \mapsto \emptyset\}, \{P \mapsto \emptyset\})$ is an *initial state*. A state $\mathcal{S}$ is a *final state* iff there is no state $\mathcal{S}'$ and such that $\mathcal{S} \rightarrow \mathcal{S}'$.

We have left the description of the algorithm generic with respect to the following choices:

1. the structure of tuples (beyond the first three fields)
2. the subsumption relation $\preceq$ on answer nodes
3. the procedure $\mathbf{resolve}(n, n')$
4. the procedure $\mathbf{generate}_{\mathcal{P}}(P)$

Intuitively, if $n \preceq n'$ (n *is subsumed by* n') holds, then the answer node n provides no more information than n'; in the algorithm, we can thus discard n and potentially ensure that the answer set is kept finite. The procedure $\mathbf{resolve}(n, n')$ is intended to take a goal node n and an answer node n' and combine the current subgoal of n with the answer provided by n' to get a new node with a simpler subgoal. The procedure $\mathbf{generate}_{\mathcal{P}}(P)$ is intended to generate a set of tuples for a given query $\langle P\rangle$ by resolving P against the rules of program $\mathcal{P}$.

Starting in an initial state, rule (root) generates answer and goal nodes for a query $\langle P\rangle$. Answer nodes are processed by (ans) which inserts them into $\mathit{Ans}(P)$ if they are not subsumed by the answers already present there; likewise they are resolved against all nodes currently waiting for an answer to P. Goal nodes are either handled by (goal_1) or (goal_2), depending on whether the current subgoal

is subsumed by an atom in the domain of the answer table. If it is, the already existing answers to that atom can be reused for the current subgoal, and the goal node is added to the wait table in (goal_1). Otherwise, (goal_2) spawns a new root node, and initializes the answer and wait tables.

2.2 Deductive Policy Evaluation

In its simplest instantiation, the algorithm performs ordinary deduction, i.e. starting in an initial state with root node $\langle P \rangle$ it will terminate in a state where $Ans(P)$ represents all instantiations of P which are deducible from the policy $\mathcal{P}$. We obtain this instantiation by defining:

1. Tuples are of the form $\langle P; \vec{Q}; S \rangle$.
2. $\langle P; [\,]; S \rangle \preceq_{\mathtt{D}} \langle P; [\,]; S' \rangle$ iff $S \preceq S'$.
3. Let $n = \langle \_; [\,]; Q' \rangle$ be an answer node, and Q'' a fresh renaming of Q'.
   $\mathbf{resolve}^{\mathtt{D}}(\langle P; [Q, \vec{Q}]; S \rangle, n) = \begin{cases} \{\langle P; \vec{Q}\theta; S\theta \rangle\} & \text{if } \theta = mgu(Q, Q'') \text{ exists,} \\ \emptyset & \text{otherwise} \end{cases}$
4. $\mathbf{generate}^{\mathtt{D}}_{\mathcal{P}}(P) = \bigcup_{(Q \leftarrow \vec{Q}) \in \mathcal{P}} \mathbf{resolve}^{\mathtt{D}}(\langle P; [Q, \vec{Q}]; Q \rangle, \langle P; [\,]; P' \rangle)$
   where P' is a fresh renaming of P.

The subsumption relation $\preceq_{\mathtt{D}}$ causes all answer nodes to be discarded whose partial solutions are "more instantiated" and therefore less general than already existing answers.

Example 2.4. The following policy allows the file `Foo` to be read by `Bob` and every employee who is associated with any work group (in particular, `Alice`):

$$\mathsf{canRead}(x, \mathtt{Foo}) \leftarrow \mathsf{isEmployee}(x), \mathsf{inWorkgroup}(x, y).$$
$$\mathsf{canRead}(\mathtt{Bob}, \mathtt{Foo}).$$
$$\mathsf{isEmployee}(\mathtt{Alice}).$$
$$\mathsf{inWorkgroup}(\mathtt{Alice}, \mathtt{WG23}).$$

Suppose the algorithm is started in an initial state with query $\langle \mathsf{canRead}(z, \mathtt{Foo}) \rangle$. The only possible start transition is (root), thus $\mathbf{generate}^{\mathtt{D}}_{\mathcal{P}}(\mathsf{canRead}(z, \mathtt{Foo}))$ is called and produces a goal node

$$n_0 \equiv \langle \mathsf{canRead}(z, \mathtt{Foo}); [\mathsf{isEmployee}(x), \mathsf{inWorkgroup}(x, y)]; \mathsf{canRead}(x, \mathtt{Foo}) \rangle$$

and an answer node $\langle \mathsf{canRead}(z, \mathtt{Foo}); [\,]; \mathsf{canRead}(\mathtt{Bob}, \mathtt{Foo}) \rangle$. Eventually, the goal node will be resolved against the last two facts in the policy to yield a second answer $\langle \mathsf{canRead}(z, \mathtt{Foo}); [\,]; \mathsf{canRead}(\mathtt{Alice}, \mathtt{Foo}) \rangle$. The algorithm terminates with no further answers.

2.3 Constructing Proof Graphs

A simple extension of the above instantiation reconstructs the proof graphs for every answer in $Ans(P)$:

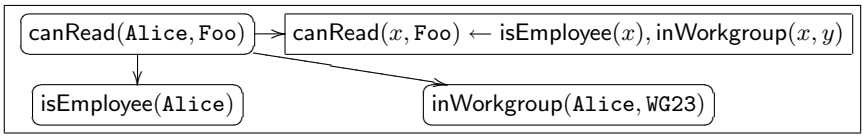

Fig. 1. Proof graph related to Example 2.4

1. Tuples are of the form $\langle P; \vec{Q}; S; \vec{n}; Cl \rangle$, where Cl is a clause in $\mathcal{P}$ and $\vec{n}$ is a sequence of answer nodes called *child nodes*.
2. $\langle P; [\,]; S; \_; \_ \rangle \preceq_{\mathtt{G}} \langle P; [\,]; S'; \_; \_ \rangle$ iff $S \preceq S'$.
3. Let $n = \langle \_; [\,]; Q'; \_; \_ \rangle$ be an answer node, and Q'' a fresh renaming of Q'.
$$\mathbf{resolve}^{\mathtt{G}}(\langle P; [Q, \vec{Q}]; S; \vec{n}; Cl \rangle, n) = \begin{cases} \{\langle P; \vec{Q}\theta; S\theta; [\vec{n}, n]; Cl \rangle\} \\ \qquad\qquad \text{if } \theta = mgu(Q, Q'') \text{ exists,} \\ \emptyset \qquad\qquad \text{otherwise} \end{cases}$$
4. Let P' be a fresh renaming of P.
$\mathbf{generate}^{\mathtt{G}}_{\mathcal{P}}(P) = \bigcup_{(Q \leftarrow \vec{Q}) \in \mathcal{P}} \mathbf{resolve}^{\mathtt{G}}(\langle P; [Q, \vec{Q}]; Q; [\,]; Cl \rangle, \langle P; [\,]; P'; [\,]; Cl \rangle)$

When resolving a goal against an answer node, the answer node is inserted as a new child node, as justification for the resolution step. In order to reconstruct the proof graph, an answer node is interpreted to have edges pointing to each of its child nodes and an edge pointing to the rule Rl which has been used to derive that particular answer. Figure 1 shows a proof graph for the derivation of canRead(Alice, Foo) in Example 2.4.

Proof graphs are useful for auditing and explaining positive access decisions. If the predicates are associated with meta-information on how they can be translated into natural language, the proof graph could also be represented as a sentence such as "Alice can read Foo because Alice is an employee and Alice is in workgroup WG23".

2.4 Abductive Policy Evaluation

In our setting, the term abduction relates to the following problem. Given an atom P and a policy $\mathcal{P}$, find all sets A of atoms such that P is deducible from $\mathcal{P}$ augmented by A. The set A is called an *abductive solution* for query P, and we require that the predicate names occurring in A are from a given set of *abducible* predicate names. The choice of the abducibles usually depends on the application domain and the kind of analysis we want to perform. In many cases, we are interested in all possible abductive solutions, so we specify all predicate names in $\mathcal{P}$ to be abducible. We define $\mathcal{A}$ to be the set of all ground instantiations of abducible predicates.

In the context of decentralized authorization, the parameters of the abductive solution may be unknown and may thus have to be left uninstantiated. For example, the solution could specify a delegation chain where the identities of the intermediate delegators cannot be fixed a priori. Therefore, the abductive solutions we are interested in may contain variables that can be arbitrarily instantiated; this is sometimes referred to as *floundering* abduction. This way,

each solution can represent an infinite number of ground solutions. We can provide a relatively simple algorithm to solve the floundering abduction problem (compared to e.g. [13,14]) mainly because our policies are monotonic.

The generic tabling scheme is instantiated as follows:

1. Tuples are of the form $\langle P; \vec{Q}; S; \Delta \rangle$, where Δ is a set of atoms called the *residue*.
2. $\langle P; [\,]; S; \Delta \rangle \preceq_{\mathtt{A}} \langle P; [\,]; S'; \Delta' \rangle$ iff $|\Delta| \geq |\Delta'|$ and there exists a substitution θ such that $S = S'\theta$ and $\Delta \supseteq \Delta'\theta$.
3. Let $n = \langle \_; [\,]; Q'; \Delta' \rangle$ be an answer node, and Q'', Δ'' fresh renamings of Q', Δ'.
$$\mathbf{resolve}^{\mathtt{A}}(\langle P; [Q, \vec{Q}]; S; \Delta \rangle, n) = \begin{cases} \{\langle P; \vec{Q}\theta; S\theta; \Delta\theta \cup \Delta''\theta \rangle\} & \\ & \text{if } \theta = mgu(Q, Q'') \text{ exists,} \\ \emptyset & \text{otherwise} \end{cases}$$
4. $\mathbf{generate}^{\mathtt{A}}_{\mathcal{P},\mathcal{A}}(P) = \bigcup_{(Q \leftarrow \vec{Q}) \in \mathcal{P}} \mathbf{resolve}^{\mathtt{A}}(\langle P; [Q, \vec{Q}]; Q; \emptyset \rangle, \langle \_; [\,]; P; \emptyset \rangle) \cup$
$\{\langle P; [\,]; P; \{P\} \rangle \ : \ P \text{ is abducible}\}$

The main idea is thus to extend tuples with a *residue* Δ, containing atoms which are just *assumed* to hold in the process of the algorithm. Such atoms are initially inserted into the residue by $\mathbf{generate}^{\mathtt{A}}_{\mathcal{P},\mathcal{A}}$ whenever an abducible goal $\langle P \rangle$ is encountered. They are then propagated using $\mathbf{resolve}^{\mathtt{A}}$ such that for each hypothetical answer S obtained, Δ expresses which atoms must be added to $\mathcal{P}$ in order to be able to deduce it.

In general, there are infinitely many abductive solutions: by monotonicity, any extension of an abductive solution is trivially also an abductive solution. Clearly, the abductive algorithm should only consider solutions that are not simple supersets of already existing ones. Similarly, we are not interested in a new solution that is an instantiation of (and thus less general than) an already existing one. The subsumption relation $\preceq_{\mathtt{A}}$ makes sure that such "uninteresting" answers are not considered.

The correctness of the algorithm is formalized by the following theorems.

Theorem 2.5 (Soundness). *If $(\mathcal{N}, Ans, Wait)$ is reachable from an initial state $\mathcal{S}_0$ then for all $P \in dom(Ans)$: $\langle P'; [\,]; S; \Delta \rangle \in Ans(P)$ implies that for all substitutions ϑ such that all elements of $\Delta\vartheta$ are ground it holds that $P = P'$, $S\vartheta \preceq P$, and $S\vartheta \in T^{\omega}_{\mathcal{P} \cup \Delta\vartheta}(\emptyset)$.*

Theorem 2.6 (Completeness). *If $\mathcal{S}_f \equiv (\mathcal{N}, Ans, Wait)$ is a final state reachable from an initial state $\mathcal{S}_0$ then for all $P \in dom(Ans)$: $S \in T^{\omega}_{\mathcal{P} \cup \mathcal{A}}(\emptyset)$ and $S \preceq P$ implies that there exists a substitution ϑ, an atom S', and a residue Δ such that $S'\vartheta = S$, $\Delta\vartheta \subseteq \mathcal{A}$, and $\langle P; [\,]; S'; \Delta \rangle \in Ans(P)$.*

Example 2.7. Consider again the program of Example 2.4, but assume that it no longer contains the atom $\mathsf{inWorkgroup}(\mathtt{Alice}, \mathtt{WG23})$. Furthermore, suppose that both $\mathsf{isEmployee}$ and $\mathsf{inWorkgroup}$ are abducible predicate names.

For query $\langle \mathsf{canRead}(z, \mathtt{Foo}) \rangle$ the procedure $\mathbf{generate}^{\mathtt{A}}_{\mathcal{P},\mathcal{A}}$ executes as the one described in Example 2.4, and produces in particular the goal node n_0. Using

(goal_2), a new root $\langle \mathsf{isEmployee}(x) \rangle$ will be inserted. The call to the function $\mathbf{generate}^{\mathtt{A}}_{\mathcal{P},\mathcal{A}}(\mathsf{isEmployee}(x))$ produces answer nodes

$\langle \mathsf{isEmployee}(x)); [\,]; \mathsf{isEmployee}(\mathtt{Alice}); \emptyset \rangle$
$\langle \mathsf{isEmployee}(x); [\,]; \mathsf{isEmployee}(x); \{\mathsf{isEmployee}(x)\} \rangle$

Upon termination of the algorithm, $\mathit{Ans}(\mathsf{canRead}(z, \mathtt{Foo}))$ contains answer node which exhibit the following answer/residue pairs:

$(\mathsf{canRead}(\mathtt{Bob}, \mathtt{Foo}), \quad \emptyset)$
$(\mathsf{canRead}(\mathtt{Alice}, \mathtt{Foo}), \{\mathsf{inWorkgroup}(\mathtt{Alice}, y)\})$
$(\mathsf{canRead}(x, \mathtt{Foo}), \quad \{\mathsf{isEmployee}(x), \mathsf{inWorkgroup}(x, y)\})$

The first one does not require any assumptions, and the other two give sets of hypothetical assumptions in order for answers of a particular shape to hold. For example, to grant read access for `Alice`, she would have to show that she is member of some work group y. All other possible abductive solutions are subsumed by these three answers.

3 Termination Conditions

The abduction algorithm from Section 2.4 is guaranteed to terminate if there is a finite set of answers such that every valid answer would be subsumed by some element in the set. However, there are cases in which every complete set of answers is infinite and the algorithm does not terminate.

Example 3.1. Consider the policy

$$\mathsf{canRead}(\mathit{user}, \mathit{file}) \leftarrow \mathsf{deleg}(\mathit{delegator}, \mathit{user}, \mathit{file}), \mathsf{canRead}(\mathit{delegator}, \mathit{file}).$$

In this example, $\mathsf{canRead}(x, y)$ indicates that principal x has read access to resource y, and $\mathsf{deleg}(x, y, z)$ indicates that principal x delegates read access for resource z to principal y. This policy implements a simple variant of discretionary access control: users can delegate read access if they have read access themselves. The abductive query $\mathsf{canRead}(\mathtt{Alice}, \mathtt{Foo})$ has an infinite set of answers with growing residues:

$\{\mathsf{canRead}(\mathtt{Alice}, \mathtt{Foo})\}$
$\{\mathsf{deleg}(x_1, \mathtt{Alice}, \mathtt{Foo}), \mathsf{canRead}(x_1, \mathtt{Foo})\}$
$\{\mathsf{deleg}(x_1, \mathtt{Alice}, \mathtt{Foo}), \mathsf{deleg}(x_2, x_1, \mathtt{Foo}), \mathsf{canRead}(x_2, \mathtt{Foo})\}$
$\{\mathsf{deleg}(x_1, \mathtt{Alice}, \mathtt{Foo}), \mathsf{deleg}(x_2, x_1, \mathtt{Foo}), \mathsf{deleg}(x_3, x_2, \mathtt{Foo}), \mathsf{canRead}(x_3, \mathtt{Foo})\}$
$\ldots$

The answers do not subsume each other: being able to provide the missing facts corresponding to one of these answer does not imply being able to provide the facts corresponding to any other answer. Clearly, the algorithm does not terminate.

There are different ways to approach this problem. If the algorithm is used for debugging policies or for explaining access denials to users, non-termination may not be a serious problem, if the answers can be returned one by one. Ideally, the answers would be returned in some meaningful order, e.g. sorted by simplicity. This can be achieved by serializing the non-deterministic algorithm into a deterministic one with a fixed order of transitions.

Sometimes, however, it is important to ensure both termination and completeness. For example, the algorithm could be used to verify that there is *no* abductive answer of a certain form; this would require a complete set of abductive solutions. This section discusses various strategies of ensuring termination.

3.1 Subsumption Weakening

One such strategy is to replace the subsumption relation $\preceq_{\mathtt{A}}$ by a weaker relation. Intuitively, a weaker subsumption relation has the effect of filtering out more answers in the (ans)-transition; in other words, fewer answers are deemed "relevant" or "interesting". As long as this alternative subsumption relation enjoys a sort of compactness property (essentially that it is not possible to indefinitely keep adding new answers that are not subsumed) then the algorithm is guaranteed to terminate.

Theorem 3.2. *Let $\sqsubseteq$ be a partial order on nodes such that in every infinite sequence of nodes $n_1, n_2, \ldots$ containing only a finite number of distinct constants, there are nodes n_i and n_j with $i < j$ and $n_j \sqsubseteq n_i$. If the subsumption relation $\preceq_{\mathtt{A}}$ on nodes in the algorithm is replaced by $\sqsubseteq$ then all transition paths starting from an initial state are of finite length.*

The following definition specifies two examples of subsumption relations that satisfy the condition of the theorem.

Definition 3.3. Let $n = \langle P; [\,]; S; \Delta\rangle$ and $n' = \langle P; [\,]; S'; \Delta'\rangle$. Then $n \sqsubseteq_0 n'$ iff $S \preceq S'$ and the predicate names occurring in Δ' are a subset of the predicate names occurring in Δ. Let M be a positive integer. Then $n \sqsubseteq_M n'$ iff $|\Delta| > M$ or $n \preceq_{\mathtt{A}} n'$.

The first relation, $\sqsubseteq_0$, is useful if one is only interested in the predicate names of the missing facts, not their parameters. For example, a security administrator may be interested in the question "*is it possible for Alice to gain read access to Foo if someone, no matter who, is granted write access?*". Here, the administrator is only interested in whether an abductive answer containing canWrite exists.

The second relation, $\sqsubseteq_M$, is parameterized on a constant M and filters out answers with more than M missing facts. This method could be used in the non-termination example above to ensure termination by cutting off the delegation chain at a certain maximum length.

3.2 Static Termination Analysis

The advantage of weakening the subsumption relation is a strong termination guarantee for all possible policies, queries and sets of abducibles. The downside of this approach is a correspondingly weaker completeness result: completeness only holds with respect to the subsumption relation.

We now develop an alternative approach that guarantees termination under the original subsumption relation, as long as the policy satisfies a certain property. To gain an intuition for this property, consider the necessary conditions for non-termination. The algorithm does not terminate only if there is an infinite sequence of abductive answers, each of which is not subsumed by any previous answer. This is only possible if the residues Δ are growing unboundedly. Since the answers can only contain the (finitely many) predicate names and constants occurring in the policy and the query, it must be the case that there is a predicate name p such that for all integers N there is always a residue in the sequence in which there are more than N occurrences of p. But this is only possible if the policy is recursive.

Verbaeten [15] provides a sufficient termination condition for general abductive logic programs that requires non-recursivity. But recursion plays an important role in authorization policies, for example for specifying delegation. Fortunately, due to the subsumption check, recursion does not always lead to non-termination. In order for the sequence to pass the subsumption condition, the p-atoms in the residues must form increasingly bigger structures that are connected via an unbounded number of shared variables. In Example 3.1, we can see that non-termination stems from the recursive canRead condition, and furthermore the sharing of the variable *delegator* with a second body predicate which is not in the head of the clause: this essentially causes the creation of an increasing linked structure of deleg atoms with newly created, shared variables.

This leads to a necessary condition for non-termination, the negation of which is then a sufficient condition for termination. The following definition is used in the formalization of the condition.

Definition 3.4. A clause $R \leftarrow \vec{R}$ is an *unfolding* of a clause $P \leftarrow P_1, ..., P_i, ..., P_n$ if there exists a clause $Q \leftarrow \vec{Q} \in \mathcal{P}$ such that P_i and Q unify with mgu θ, and $R = P\theta$ and $\vec{R} = (P_1, ..., P_{i-1}, \vec{Q}, P_{i+1}, ..., P_n)\theta$. A clause C can be *unfolded* to yield a clause C' if C' is obtained from C by 0 or more unfolding transformations.

Theorem 3.5. *Let $\mathcal{P}$ be a set of clauses such that no clause can be unfolded to yield a clause with the following property:*

- *The head predicate occurs in a body atom P.*
- *It has a second body atom Q that is abducible and shares a variable with P which does not occur in the head.*

Then all transition paths starting from an initial state are of finite length.

The condition in Theorem 3.5 is decidable; in fact, a static analyzer for checking it can be written in Prolog in less than a hundred lines. The static analyzer

could then be run before an abductive query, and if the condition is not satisfied, the user could be warned that evaluation of the query may not terminate. Preliminary experiments have shown that the condition gives a relatively accurate approximation for non-termination. In particular, many recursive policies can be shown to guarantee termination.

4 Application Scenarios

This section illustrates three possible applications of the abduction algorithm in the area of policy-based access control.

4.1 Explaining Access Denials to Users

In current access control systems, the user is often left without guidance if access is denied. Consider the following policy of some company:

$$\mathsf{canRead}(x, \texttt{/workgroup23/}) \leftarrow \mathsf{isEmployee}(x), \mathsf{inWorkgroup}(x, \texttt{WG23}).$$
$$\mathsf{canRead}(x, \texttt{/workgroup23/}) \leftarrow \mathsf{isManager}(x).$$

Suppose employee `Alice` submits the credential $\mathsf{isEmployee}(\texttt{Alice})$ upon log-on to the authorization system, but forgets to submit the credential that she is member of workgroup `WG23`. If she then tries to access the folder `/workgroup23/`, she will just get the answer "access denied". Using the abduction technique, the produced residues $\{\mathsf{inWorkgroup}(\texttt{Alice}, \texttt{WG23})\}$ and $\{\mathsf{isManager}(\texttt{Alice})\}$ could be used to construct the more helpful message "access would have been granted if you had shown that you are a member of work group WG23 or that you are a manager".

If parts of the policy itself are considered confidential, the abductive solutions can be filtered by a *disclosure* meta-policy. Disclosure policies have been studied extensively in the area of automated trust negotiation (e.g. [16,17,18]). A simpler (but slightly less fine-grained) approach would be to tag particular atoms in clause bodies that must not be abduced.

4.2 Administration of Authorization Policies

The next example illustrates the use of the abduction algorithm in a policy analysis and debugging tool. Consider the following example rules that are part of a policy of an electronic health record (EHR) service:

$$\mathsf{treatingClinician}(\mathit{cli}, \mathit{pat}) \leftarrow \quad (1)$$
$$\mathsf{roleMember}(\mathit{pat}, \texttt{Patient}), \mathsf{roleMember}(\mathit{cli}, \texttt{Clinician}), \mathsf{consent}(\mathit{pat}, \mathit{cli}).$$

$$\mathsf{canReadEHR}(\mathit{cli}, \mathit{pat}, \mathit{subj}) \leftarrow \quad (2)$$
$$\mathsf{treatingClinician}(\mathit{cli}, \mathit{pat}), \mathsf{nonSensitive}(\mathit{subj}).$$

$$\mathsf{canReadEHR}(\mathit{cli}, \mathit{pat}, \mathtt{Psych}) \leftarrow \quad (3)$$
$$\quad \mathsf{treatingClinician}(\mathit{cli}, \mathit{pat}), \mathsf{isCertifiedPsychiatrist}(\mathit{cli}).$$

$$\mathsf{canReadEHR}(\mathit{pat}, \mathit{pat}, \mathit{subj}) \leftarrow \quad (4)$$
$$\quad \mathsf{roleMember}(\mathit{pat}, \mathtt{Patient}), \mathsf{nonSensitive}(\mathit{subj}).$$

Rule (1) specifies that a clinician *cli* is a *treating clinician* of patient *pat* if the patient has given consent to treatment. The predicate $\mathsf{canReadEHR}(x, \mathit{pat}, \mathit{subj})$ is used to check if principal x is permitted to read patient *pat*'s record items on subject matter *subj*, where subject matters range over categories such as psychiatry, cardiology, radiology etc. Some subject matters, such as `Psych`, are deemed sensitive and have stricter access requirements. The predicate $\mathsf{nonSensitive}(\mathit{subj})$ defines the range of subjects that are deemed non-sensitive. Rule (2) allows clinicians to read their patients' record items on non-sensitive subjects. Rule (3) specifies that only psychiatrists are permitted to access their patients' psychiatric data. Finally, Rule (4) permits patients to view their own data, but only the items regarding non-sensitive subject matters.

The rationale behind Rule (4) is that patients should not be allowed to access data that could potentially distress them if read without professional guidance. In particular, they should not be able to autonomously access their psychiatric data. In order to check if the policy really implements the intended behavior, the security administrator can issue the abductive query $\mathsf{canReadEHR}(\mathit{pat}, \mathit{pat}, \mathtt{Psych})$ to see if there is a way for patients to access their own psychiatric data.

Assuming that all predicates apart from canReadEHR and treatingClinician are abducible, we obtain an answer with residues

$$\{\mathsf{roleMember}(\mathit{pat}, \mathtt{Patient}), \mathsf{nonSensitive}(\mathtt{Psych})\}.$$

This is easily dismissed as unproblematic if the administrator can verify that there is no way of inserting a fact $\mathsf{nonSensitive}(\mathtt{Psych})$. But we also get a second answer with residue

$$\{\mathsf{roleMember}(\mathit{pat}, \mathtt{Patient}), \mathsf{roleMember}(\mathit{pat}, \mathtt{Clinician}),$$
$$\mathsf{isCertifiedPsychiatrist}(\mathit{pat}), \mathsf{consent}(\mathit{pat}, \mathit{pat})\}.$$

This answer is more troublesome: a patient can read what her psychiatrist has written if she happens to be a certified psychiatrist too and has given consent to treat herself. This may or may not be regarded as a bug in the policy; but in any case, as there are no further abductive answers, and by completeness and termination of the algorithm, it is guaranteed that there are no further loopholes in the policy.

4.3 Automated Delegation

The following scenario takes place in a multi-domain grid computing environment. Alice is a user who wishes to submit a job to be computed on a grid

compute cluster. She knows that during the execution of her job, a node from the compute cluster will have to access her file `alice.dat` stored on a file server in a different domain. Therefore, at some point, an authorization query of the form $\mathsf{canRead}(\mathtt{Node}, \mathtt{alice.dat})$ will be evaluated on the file server. Suppose the file server's policy contains the rule from Example 3.1 and the fact $\mathsf{canRead}(\mathtt{Alice}, \mathtt{alice.dat})$.

As Alice's job may take many days to complete, she wants to know in advance which delegation credentials she has to submit to the file server, so she sends the abductive query $\mathsf{canRead}(\mathit{node}, \mathtt{alice.dat})$ to the server. Her query contains a variable *node* in place of the node that will eventually access her file, because she cannot know its identity in advance.

The first returned answer is the trivial answer where *node* is instantiated to `Alice` and the residue is empty. The second answer is uninstantiated and has the singleton residue $\{\mathsf{deleg}(\mathtt{Alice}, \mathit{node}, \mathtt{alice.dat})\}$. This would require direct delegation to the node, which is not convenient as its identity is not known to Alice. The third answer has residue

$$\{\mathsf{deleg}(\mathtt{Alice}, x, \mathtt{alice.dat}), \mathsf{deleg}(x, \mathit{node}, \mathtt{alice.dat})\}.$$

This answer represents a delegation chain of depth 2 and is the most useful in this situation, because Alice knows the identity of the compute cluster's scheduling service. Thus she can submit a delegation credential

$$\mathsf{deleg}(\mathtt{Alice}, \mathtt{Scheduler}, \mathtt{alice.dat})$$

to the scheduling service along with her job and a partially instantiated missing-credential "template"

$$\{\mathsf{deleg}(\mathtt{Scheduler}, \mathit{node}, \mathtt{alice.dat})\}.$$

The service will then execute the job on some node, e.g. `Node42`, passing along Alice's delegation credential as well as a newly created (or cached) credential instantiated from the template, namely $\mathsf{deleg}(\mathtt{Scheduler}, \mathtt{Node42}, \mathtt{alice.dat})$. When the node eventually requests access to Alice's file on the file server, it submits both Alice's and the scheduler's delegation credentials to support the request. Access is then guaranteed to be granted as long as the file server's policy has not been changed in the meantime.

5 Discussion

Related work There has been very little research on improving the usability of authorization systems in the case of access denial. The *Know* system [19] can provide helpful feedback to the user in the form of a list of conditions under which the policy allows access. A separate disclosure policy restricts the information revealed by the feedback. However, the authors only consider policies of rather limited expressiveness, namely those that can be written as propositional boolean

formulas; hence the feedback can be computed using Ordered Binary Decision Diagrams (OBDDs).

Bonatti et al. [20] have developed a framework for explaining both positive and negative access decisions in the context of a Datalog-based authorization language. For explaining access denials, they essentially compute a tabled failed proof graph (called explanation graph) for the query. Users can navigate through the graph, which is represented in controlled natural language, to see where the proof failed. To keep the overhead as low as possible, they do not attempt to search for the missing facts that would complete the failed proof. We have shown in this paper that we can compute the sets of missing facts while maintaining a low implementation overhead.

Koshutanski and Massacci [18] employ abduction for interactive credential retrieval: if the credentials presented by the user are not sufficient to allow access, the service computes a set of missing credentials (filtered by a disclosure policy) and returns it to the user who can either supply the required credentials or decline, in which case the process iterates. This process terminates because it is assumed that the set of constants that could be used as credential parameters is finite and known to the service in advance. The policy can then be reduced to propositional (variable-free) formulas. However, we believe that this is an unreasonable assumption, particularly in decentralized applications where authorization is based on attributes as the identities of principals in delegation chains are not known a priori.

Becker and Nanz [21] have developed an algorithm for analyzing authorization policies in which facts (such as current role activations) can be added and removed dynamically by commands (such as activating or deactivating a role). The paper presupposes a function for computing sufficient preconditions for executing such commands, but does not explain how it can be implemented. The abductive procedure presented in this paper could be used to implement the required function.

Implementation. Prototypes of the abductive algorithm and the static termination analysis have been implemented in OCaml and in XSB Prolog, respectively. The SecPAL system [5] supports proof graph generation based on the algorithm presented here. Current and future work on the SecPAL system will include integration of tools based on our abduction algorithm. As SecPAL is compiled into Datalog with constraints, the implementation will also have to handle constraints. Extending the abduction algorithm with constraints is relatively simple, assuming the existence of satisfiability and subsumption checking operations. In essence, tuples are extended to include a constraint on variables occurring in the predicate and the residue. In the resolution step, the conjunction of the constraints from the two input nodes is computed and checked for satisfiability. Furthermore, the subsumption check is extended to also check if the constraint from one node is subsumed by the constraint from the other node.

Conclusion. Declarative authorization languages can increase the flexibility, scalability, and manageability of access control systems. However, they are not with-

out their own usability problems, as many authorization policies are intrinsically complex. To alleviate this complexity, tools are needed for facilitating access auditing and review, meaningful user feedback, policy diagnosis and debugging, and automated credential retrieval. In this paper, we have shown how a tabled resolution algorithm for policy evaluation can be extended to create proof graphs and to compute sets of missing proof facts using abduction. For the abductive algorithm, we have explored methods for guaranteeing termination and complete answers. The algorithms are general enough to be applicable to a wide range of existing systems. We believe that tools based on these algorithms will help the declarative policy approach gain wider adoption in the industry.

References

1. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management framework. In: Symposium on Security and Privacy. (2002) 114–130
2. DeTreville, J.: Binder, a logic-based security language. In: IEEE Symposium on Security and Privacy. (2002) 105–113
3. Li, N., Mitchell, J.C.: Datalog with constraints: A foundation for trust management languages. In: Practical Aspects of Declarative Languages. (2003) 58–73
4. Becker, M.Y., Sewell, P.: Cassandra: Flexible trust management, applied to electronic health records. In: IEEE Computer Security Foundations Workshop. (2004)
5. Becker, M.Y., Fournet, C., Gordon, A.D.: Design and semantics of a decentralized authorization language. In: IEEE Computer Security Foundations Symposium. (2007)
6. Peirce, C.: Abduction and Induction. In: Philosophical Writings of Peirce. (1955)
7. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In Gabbay, D.M., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 5. (1998) 235–324
8. Becker, M.Y., Nanz, S.: The role of abduction in declarative authorization policies. Technical Report MSR-TR-2007-105, Microsoft Research (2007) http://research.microsoft.com/research/pubs/view.aspx?tr_id=1348.
9. Becker, M.Y., Sewell, P.: Cassandra: distributed access control policies with tunable expressiveness. In: IEEE International Workshop on Policies for Distributed Systems and Networks. (2004) 159–168
10. Tamaki, H., Sato, T.: OLD resolution with tabulation. In: International Conference on Logic Programming, Springer-Verlag (1986) 84–98
11. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM **43**(1) (1996) 20–74
12. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)
13. Alferes, J.J., Pereira, L.M., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. Theory and Practice of Logic Programming **4**(4) (2004) 383–428
14. Denecker, M., Schreye, D.D.: SLDNFA: An abductive procedure for abductive logic programs. Journal of Logic Programming **34**(2) (1998) 111–167
15. Verbaeten, S.: Termination analysis for abductive general logic programs. In: International Conference on Logic Programming. (1999) 365–379
16. Winsborough, W.H., Seamons, K.E., Jones, V.E.: Automated trust negotiation. In: DARPA Information Survivability Conference and Exposition. Volume 1. (2000)

17. Winsborough, W.H., Li, N.: Towards practical automated trust negotiation. In: IEEE International Workshop on Policies for Distributed Systems and Networks. (2002)
18. Koshutanski, H., Massacci, F.: Interactive access control for web services. In: International Information Security Conference. (2004) 151–166
19. Kapadia, A., Sampemane, G., Campbell, R.H.: Know why your access was denied: regulating feedback for usable security. In: ACM conference on computer and communications security. (2004) 52–61
20. Bonatti, P.A., Olmedilla, D., Peer, J.: Advanced policy explanations on the web. In: European Conference on Artificial Intelligence. (2006) 200–204
21. Becker, M.Y., Nanz, S.: A logic for state-modifying authorization policies. In: European Symposium on Research in Computer Security. (2007)