SecPAL: Design and Semantics of a Decentralized Authorization Language

Moritz Y. Becker   Cédric Fournet
Andrew D. Gordon

September 2006

Technical Report
MSR–TR–2006–120

Microsoft Research
Roger Needham Building
7 J.J. Thomson Avenue
Cambridge, CB3 0FB
United Kingdom

The SecPAL project is a collaboration between the Advanced Technology Incubation Group of Microsoft's Chief Research and Strategy Officer and Microsoft Research, Cambridge.

The members of the Incubation Group are Blair Dillaway, Gregory Fee, Jason Hogg, Larry Joy, Brian LaMacchia, and Jason Mackay.

SecPAL: Design and Semantics of a Decentralized Authorization Language

Moritz Y. Becker Cédric Fournet Andrew D. Gordon

September 2006

Abstract

We present a declarative authorization language. Policies and credentials are expressed using predicates defined by logical clauses, in the style of constraint logic programming. Access requests are mapped to logical authorization queries, consisting of predicates and constraints combined by conjunctions, disjunctions, and negations. Access is granted if the query succeeds against the current database of clauses. Predicates ascribe rights to particular principals, with flexible support for delegation and revocation. At the discretion of the delegator, delegated rights can be further delegated, either to a fixed depth, or arbitrarily deeply.

Our language strikes a fine balance between semantic simplicity, policy expressiveness, and execution efficiency. The semantics consists of just three deduction rules. The language can express many common policy idioms using constraints, controlled delegation, recursive predicates, and negated queries. We describe an execution strategy based on translation to Datalog with constraints and table-based resolution. We show that this execution strategy is sound, complete, and always terminates, despite recursion and negation, as long as simple syntactic conditions are met.

Contents

1 Introduction **1**

2 Syntax and semantics of SecPAL **4**

3 Authorization queries **8**

4 Policy idioms **11**

5 Translation into constrained Datalog **15**

6 Datalog evaluation with tabling **17**

7 Evaluation of authorization queries **20**

8 Discussion **21**

A Assertion expiration and revocation **27**

B Proof graph generation **28**

C Auxiliary definitions and proofs **28**

C.1 Authorization queries 28

C.2 Translation into constrained Datalog 30

C.3 Datalog evaluation with tabling 32

C.4 Evaluation of authorization queries 36

1 Introduction

Many applications depend on complex and changing authorization criteria. In some domains, such as electronic health records or eGovernment, authorization must comply with evolving legislation. Distributed models such as web services or shared grid computations involve frequent, ad hoc collaborations between entities with no pre-established trust relation, each with their own authorization policies. Hence, these policies must be phrased in terms of principal attributes, asserted by adequate delegation chains, as well as traditional identities. To deploy and maintain such applications, it is essential that all mundane authorization decisions be automated, according to some human readable policy that can be refined and updated, without the need to change (and re-validate) application code.

To this end, several declarative authorization management systems have been proposed; they feature high-level languages dedicated to authorization policies; they aim at improving scalability, maintenance, and availability by separating policy-based access control decisions from their implementation mechanisms. Despite their obvious advantages, these systems are not used much. The poor usability of policy languages remains a major obstacle to their adoption. In this paper, we describe the design and semantics of SecPAL, a new authorization language that improves on usability in several aspects. SecPAL is being implemented and deployed as part of a large system-development project, initially targeted at grid applications [17]. The early application of SecPAL for access control within a prototype distributed computing environment has led to many improvements in its design.

Expressiveness Flexible delegation of authority is a key feature for decentralized systems. SecPAL introduces a novel delegation mechanism that covers a wider spectrum of delegation variants than existing authorization languages, including those specifically designed for flexible delegation such as XrML [12], SPKI/SDSI [18] and Delegation Logic (DL) [29]. Support for domain-specific constraints is another crucial feature, but existing solutions have been very restrictive to preserve decidability and tractability [31, 5], and disallow constraints that are required for expressing idioms commonly used in practice. We introduce a novel set of mild and purely syntactic safety conditions that allow an open choice of constraints without loss of efficiency. SecPAL can thus express a wide range of idioms, including policies with parameterised roles and role hierarchies, separation-of-duties and threshold constraints, expiration requirements, temporal and periodicity constraints, policies on structured resources such as file systems, and revocation.

Clear, readable syntax The syntax of some policy languages, such as XACML [36] and XrML, is defined only via an XML schema; policies expressed directly in XML are verbose and hard to read and write. On the other hand, policy authors are usually unfamiliar with formal logic, and would find it hard to learn the syntax of most logic-based policy languages (e.g. [26, 14, 32, 22, 31, 5]). SecPAL has a concrete syntax consisting of simple, succinct statements close to natural language. (It also has an XML schema for exchanging statements between implementations.)

Intuitive, unambiguous semantics Languages such as XACML, XrML, or SPKI/SDSI [18] are specified by a combination of lengthy descriptions and algorithms that are ambiguous and, in some cases, inconsistent. Post-hoc attempts to formalise these languages are difficult and reveal their semantic ambiguities and complexities (e.g. [24, 23, 1, 30]). Without a formal foundation, it cannot be guaranteed that these languages are decidable or tractable. Logic-based languages have a formal, unambiguous semantics, but that does not mean they are easily comprehensible. In many cases, the semantics is specified only by translation to another formal language, such as Datalog. The semantics of SecPAL assertions is specified by three deduction rules that succinctly capture the intuitive meaning from the syntax. This is far more comprehensible than specifying the semantics via an algorithm or translation.

Another source of complexity is the use of negated conditions. Negation is a required feature for expressing idioms such as separation of duties, but in conjunction with recursion it can cause intractability and semantic ambiguity [45], and it generally makes policies much harder to comprehend. By introducing a level of authorization queries where negation is permitted and separating it from the negation-free assertion language, we avoid these problems and simplify the task of authoring policies with negation.

Effective decision procedures SecPAL query evaluation is decidable and tractable (with polynomial data complexity) by translation into constrained Datalog We describe a tabling resolution algorithm for efficient evaluation of authorization queries and present correctness and complexity theorems for the translation and the algorithm.

Extensibility SecPAL builds on the notion of tunable expressiveness introduced in [4] and defines several extension points at which functionality can be added in a modular and orthogonal way. For example, the parameterized verbs, the environment functions, and the language of constraints can all be extended by the user without affecting our results.

To introduce the main features of SecPAL, we consider an example in the context of a simplified grid system. Assume that Alice wishes to perform some data mining on a computation cluster. To this end, the cluster needs to fetch Alice's dataset from her file server. A priori, the cluster may not know of Alice, and the cluster and the file server may not share any trust relationship. We identify principals by names `Alice`, `Cluster`, `FileServer`, ...; these names stand for public signature-verification keys in the SecPAL implementation.

Alice sends to the cluster a request to run the command `dbgrep file://project/data.db` plus a collection of tokens for the request, expressed as three SecPAL assertions:

`STS` says `Alice` is a researcher (1)

`FileServer` says `Alice` can read `file://project` (2)

`Alice` says `Cluster` can read `file://project/data.db` if
currentTime() $\leq$ 07/09/2006 (3)

Every assertion is XML-encoded and signed by its issuer. Assertion (1) is an identity token issued by STS, some security token server trusted by the cluster. Assertion (2) is a capability for Alice to read her files, issued by FileServer. Assertion (3) delegates to Cluster the right to access a specific file on that server, for a limited period of time; it is specifically issued by Alice to support her request.

Before processing the request, the cluster authenticates Alice as the requester, validates her tokens, and runs the query Cluster says Alice can execute dbgrep against the set of assertions formed by its local policy plus these tokens. (Pragmatically, an authorization query table for the cluster maps any requests to run dbgrep to such queries.) Assume the local policy of the cluster includes the assertions:

$$\texttt{Cluster}\ \textsf{says}\ \texttt{STS}\ \textsf{can say}_0\ x\ \textsf{is a researcher} \tag{4}$$

$$\texttt{Cluster}\ \textsf{says}\ x\ \textsf{can}\ \texttt{execute dbgrep}\ \textsf{if}\ x\ \textsf{is a researcher} \tag{5}$$

Assertions (4) and (5) state that Cluster defers to STS to say who is a researcher, and that any researcher may run dbgrep. (More realistic assertions may also include conditions and constraints.) Here, we deduce that Cluster says Alice is a researcher by (1) and (4), then deduce the target assertion by (5).

The cluster then executes the task, which involves requesting chunks of file://project/data.db hosted on the file server. To support its requests, the cluster forwards Alice's credentials. Before granting access to the data, the file server runs the query Cluster can read file://project/data.db against its local policy plus Alice's tokens. Assume the local policy of the server includes the assertion

$$\begin{array}{l}\texttt{FileServer}\ \textsf{says}\ x\ \textsf{can say}_\infty\ y\ \textsf{can}\ \texttt{read}\ \mathit{file}\ \textsf{if} \\ \quad x\ \textsf{can}\ \texttt{read}\ \mathit{dir},\ \mathit{file} \preceq \mathit{dir},\ \textsf{markedConfidential}(\mathit{file}) \neq \texttt{Yes}\end{array} \tag{6}$$

Assertion (6) is a constrained delegation rule; it states that any principal x may delegate the right to read a file, provided x can read a directory *dir* that includes the file and the file is not marked as confidential. The first condition is a logical fact, whereas the last two conditions are constraints. Here, by (3) and (6) with $x = \texttt{Alice}$ and $y = \texttt{Cluster}$, the first condition follows from (2) and we obtain that FileServer says Cluster can read file://project/data.db provided that FileServer successfully checks the constraints $\textsf{currentTime}() \leq \texttt{07/09/2006}$ and $\textsf{markedConfidential}(\texttt{file://project/data.db}) \neq \texttt{Yes}$.

In the delegation rules (4) and (6), the can says have different subscripts: in (4), $\textsf{can say}_0$ prevents STS from re-delegating the delegated fact; conversely, in (6), $\textsf{can say}_\infty$ indicates that y can re-delegate read access to *file* by issuing adequate can say tokens.

Assume now that the cluster distributes the task to several computation nodes, such as Node23. In order for Node23 to gain access to the data, Cluster may issue its own delegation token, so that the query FileServer says Node23 can read file://project/data.db may be satisfied by applying (6) twice, with $x = \texttt{Alice}$ then $x = \texttt{Cluster}$. Alternatively, FileServer may simply issue the assertion

$$\texttt{FileServer}\ \textsf{says}\ \texttt{Node23}\ \textsf{can act as}\ \texttt{Cluster} \tag{7}$$

With this aliasing mechanism, any assertion `FileServer` says `Node23` *verbphrase* follows from `FileServer` says `Cluster` *verbphrase*.

Contents The rest of the paper is organized as follows. Section 2 defines the syntax and semantics of SecPAL assertions. Section 3 defines SecPAL authorization queries, built as conjunctions, disjunctions and negations of facts and constraints. Section 4 shows how to express a variety of authorization policy idioms in SecPAL. Sections 5, 6, and 7 give our algorithm for evaluating authorization queries and establish its formal soundness and completeness. SecPAL assertions are first translated into constrained Datalog (Section 5); the resulting program is then evaluated using a deterministic variant of resolution with tabling (Section 6) for a series of Datalog queries obtained from the SecPAL query (Section 7). Section 8 compares SecPAL to related languages and concludes.

Appendix A explains how assertions are filtered to remove expired or revoked assertions prior to evaluating authorization queries. Appendix B discusses representations of proof trees. Appendix C provides auxiliary definitions and all proofs.

2 Syntax and semantics of SecPAL

We give a core syntax for SecPAL. (The full SecPAL language provides additional syntax for grouping assertions, for instance to delegate a series of rights in a single assertion; these additions can be reduced to the core syntax. It also enforces a typing discipline for constants, functions, and variables, omitted here.)

Assertions Authorization policies are specified as sets of assertions of the form

$$A \text{ says } fact \text{ if } fact_1, ..., fact_n, c$$

where the facts are sentences that state properties on principals. A is the *issuer*; the $fact_i$ are the *conditional facts*; and c is the *constraint*. Assertions are similar to Horn clauses, with the difference that (1) they are qualified by some principal A who issues and vouches for the asserted claim; (2) facts can be nested, using the keyword can say, by means of which delegation rights are specified; and (3) variables in the assertion are constrained by c, a first-order formula that can express e.g. temporal, inequality, path and regular expression constraints.

$$
\begin{array}{rcll}
e & ::= & x & \text{(variables)} \\
 & | & A & \text{(constants)} \\
\mathit{pred} & \in & \{\mathsf{possesses}, \mathsf{can}, ...\} & \text{(predicates)} \\
D & ::= & 0 & \text{(one-step, non-transitive delegation)} \\
 & | & \infty & \text{(unbounded, transitive delegation)} \\
\mathit{verbphrase} & ::= & \mathit{pred}\ e_1\ ...\ e_n & \text{for } n = \mathit{Arity}(\mathit{pred}) \geq 0 \\
 & | & \mathsf{can\ say}_D\ \mathit{fact} & \text{(delegation with specified depth } D\text{)} \\
 & | & \mathsf{can\ act\ as}\ e & \\
\mathit{fact} & ::= & e\ \mathit{verbphrase} & \\
\mathit{claim} & ::= & \mathit{fact}\ \mathsf{if}\ \mathit{fact}_1, ..., \mathit{fact}_n, c & \text{for some } n \geq 0 \\
\mathit{assertion} & ::= & A\ \mathsf{says}\ \mathit{claim} & \\
\mathit{AC} & ::= & \{\mathit{assertion}_1, ..., \mathit{assertion}_n\} & \text{(assertion context)}
\end{array}
$$

Constants represent data such as IP addresses, DNS names, URLs, dates, and times. Variables only range over the domain of constants — not predicates, facts, claims or assertions. Predicates are verb phrases of fixed arity with holes for their object parameters; holes may appear at any fixed position in verbphrases, as in e.g. has access from $[-]$ till $[-]$.

We say that a fact is *flat* when it does not contain can say, and is *nested* otherwise. Facts are of the general form e_1 can say$_{D_1}$ ... e_n can say$_{D_n}$ *fact*, where $n \geq 0$ and *fact* is flat. For example, the fact `Bob` can `read` f is flat, but `Charlie` can say$_0$ `Bob` can `read` f is not.

Constraints Constraints range over any constraint domain that extends the *basic constraint domain* shown below. Basic constraints include integer inequalities (for e.g. expressing temporal constraints), path constraints (for hierarchical file systems), and regular expressions (for ad hoc filtering):

$$
\begin{array}{rcll}
f & \in & \{+, -, \mathsf{CurrentTime}, ...\} & \text{(built-in functions)} \\
e & ::= & x & \\
 & | & A & \\
 & | & f(e_1, ..., e_n) & \text{for } n = \mathit{Arity}(f) \geq 0 \\
\mathit{pattern} & \in & \text{RegularExpressions} & \\
c & ::= & \mathtt{True} & \\
 & | & e_1 = e_2 & \\
 & | & e_1 \leq e_2 & \text{(integer inequality)} \\
 & | & e_1 \preceq e_2 & \text{(path constraint)} \\
 & | & e\ \mathsf{matches}\ \mathit{pattern} & \text{(regular expression)} \\
 & | & \mathsf{not}(c) & \text{(negation)} \\
 & | & c_1, c_2 & \text{(conjunction)}
\end{array}
$$

Additional constraints can be added without affecting decidability or tractability. In contrast to Cassandra [5] or RT^C [31], the only requirement is that the validity of ground constraints is decidable in polynomial time. (A phrase of syntax is *ground* when it contains no variables.)

We use a sugared notation for constraints that can be derived from the basic ones, e.g. False, $e_1 \neq e_2$ or c_1 or c_2. In assertions, we usually omit the True constraint, and also omit the if when the assertion has no conditional facts.

We write keywords, function names and predicates in $\mathsf{sans\ serif}$, constants in `typewriter` font, and variables in *italics*. We also use A, B, C, and D as meta variables for constants, usually for denoting principals. We use a vector notation to denote a (possibly empty) list (or tuple) of items, e.g. we may write $f(\vec{e})$ for $f(e_1, ..., e_n)$.

For a given constraint c, we write $\models c$ iff c is ground and valid. The following defines ground validity within the basic constraint domain. The denotation of a constant A is simply $[\![A]\!] = A$. The denotation of a function $f(\vec{e})$ is defined if $\vec{e}$ is ground, and is also a constant, but may depend on the system state as well as $[\![\vec{e}]\!]$. For example, $[\![\mathsf{CurrentTime}()]\!]$ returns a different constant when called at different times. However, we assume that a single authorization query evaluation is atomic with respect to system state. That is, even though an expression may be evaluated multiple times, we require that its denotation not vary during a single evaluation.

$\models \mathsf{True}$	
$\models e_1 = e_2$	iff $[\![e_1]\!]$ and $[\![e_2]\!]$ are equal constants
$\models e_1 \leq e_2$	iff $[\![e_1]\!]$ and $[\![e_2]\!]$ are integer constants and $[\![e_1]\!] \leq [\![e_2]\!]$
$\models e_1 \preceq e_2$	iff $[\![e_1]\!]$ and $[\![e_2]\!]$ are path constants and $[\![e_1]\!]$ is a descendant of, or equal to, $[\![e_2]\!]$
$\models e$ $\mathsf{matches}$ *pattern*	iff $[\![e]\!]$ is a string constant that matches *pattern*
$\models \mathsf{not}(c)$	iff $\models c$ does not hold
$\models c_1, c_2$	iff $\models c_1$ and $\models c_2$

Semantics We now give a formal definition of the language semantics. In the rest of this document, we refer to a substitution θ as a function mapping variables to constants and variables. Substitutions are extended to constraints, predicates, facts, claims, assertions etc. in the natural way, and are usually written in postfix notation. We write $vars(X)$ for the set of variables occurring in a phrase of syntax X.

We now present the three deduction rules to capture the semantics of the language. Each rule consists of a set of premises and a single consequence of the form $AC, D \models A$ says *fact* where $vars(fact) = \emptyset$ and the delegation flag D is 0 or ∞. Intuitively, the deduction relation holds if the consequence can be derived from AC. If $D = 0$, no instance of the rule (can say) occurs in the derivation.

$$\text{(cond)}\ \frac{\begin{array}{l}(A \textsf{ says } \mathit{fact} \textsf{ if } \mathit{fact}_1, ..., \mathit{fact}_k, c) \in AC \\ AC, D \models A \textsf{ says } \mathit{fact}_i\theta \text{ for all } i \in \{1..k\} \qquad \models c\theta \qquad \mathit{vars}(\mathit{fact}\theta) = \emptyset\end{array}}{AC, D \models A \textsf{ says } \mathit{fact}\theta}$$

$$\text{(can say)}\ \frac{AC, \infty \models A \textsf{ says } B \textsf{ can say}_D\ \mathit{fact} \qquad AC, D \models B \textsf{ says } \mathit{fact}}{AC, \infty \models A \textsf{ says } \mathit{fact}}$$

$$\text{(can act as)}\ \frac{AC, D \models A \textsf{ says } B \textsf{ can act as } C \qquad AC, D \models A \textsf{ says } C\ \mathit{verbphrase}}{AC, D \models A \textsf{ says } B\ \mathit{verbphrase}}$$

Rule (cond) allows the deduction of matching assertions in AC with all free variables substituted by constants. All conditional facts must be deducible, and the substitution must also make the constraint valid. The delegation flag D is propagated to all conditional facts. Rule (can say) deduces an assertion made by A by combining a can say assertion made by A and a matching assertion made by B. This rule applies only if the delegation flag is ∞. The matching assertion made by B must be proved with the delegation flag D obtained from A's can say assertion. Rule (can say) states that any fact that holds for C also holds for B.

Rule (can act as) asserts that facts applicable to C also apply to B, when B can act as C is derivable. A corollary is that can act as is a transitive relation.

Corollary 2.1. If $AC, D \models A$ says B can act as B' and $AC, D \models A$ says B' can act as B'' then $AC, D \models A$ says B can act as B''.

The following propositions state basic properties of the deduction relation; they are established by induction on the rules.

Proposition 2.2. If $AC, D \models A$ says *fact* then $\mathit{vars}(\mathit{fact}) = \emptyset$.

Proposition 2.3. If $AC, 0 \models A$ says *fact* then $AC, \infty \models A$ says *fact*.

Proposition 2.4. If $AC_1, D \models A$ says *fact* then $AC_1 \cup AC_2, D \models A$ says *fact*.

Proposition 2.5. Let AC_A be the set of all assertions in AC whose issuer is A. $AC, 0 \models A$ says *fact* iff $AC_A, 0 \models A$ says *fact*.

Proposition 2.5 implies that if A says *fact* is deduced from a zero-depth delegation assertion A says B can $\textsf{say}_0$ *fact* then the delegation chain is guaranteed to depend only on assertions issued by B. XrML and DL [29] also support depth restrictions, but these can be defeated as their semantics do not have the stated property. Section 8 discusses this issue in more detail.

Safety The complexity of constraint logic programs depends on the choice of the constraint domain [44]. For this reason, authorization languages either support no constraints at all, or only a limited class of constraints, in order to guarantee decidability and tractability [3, 5, 31]. However, this would prohibit the use of constraints (even just the basic constraint domain introduced above) that are actually needed in practice. SecPAL has

virtually no restriction on the choice of constraint domain and yet remains decidable and tractable. To achieve this, we enforce a simple, purely syntactic safety condition on the variables occurring in assertions while allowing very expressive constraint domains.

Definition 2.6. (Assertion safety) Let A says *fact* if $fact_1, ..., fact_n, c$ be an assertion. A variable $x \in vars(fact)$ is *safe* iff $x \in vars(fact_1) \cup ... \cup vars(fact_n)$.

The assertion A says *fact* if $fact_1, ..., fact_n, c$ is *safe* iff

1. if *fact* is flat, all variables in *vars*(*fact*) are safe;
   otherwise (i.e. *fact* is of the form e can say$_D$ $fact'$) e is either a constant or a safe variable;
2. $vars(c) \subseteq vars(fact) \cup vars(fact_1) \cup ... \cup vars(fact_n)$;
3. $fact_1, ..., fact_n$ are flat.

Examples Here are examples of *safe* assertions:

A says B can read Foo
A says B can read Foo if B can x y
A says B can read Foo if B can x y, $x \neq y$
A says B can x y if B can x y
A says z can x y if z can x Foo, z can read y
A says B can say$_0$ x can y z

These assertions are *unsafe*:

A says B can x Foo
A says z can read Foo if B can x y
A says B can read Foo if B can x y, $w \neq y$
A says B can x y if B can say$_0$ C can x y
A says w can say$_0$ x can y z

The safety condition guarantees that the evaluation of the Datalog translation, as described in Section 6, is complete and terminates in all cases. In essence, it makes sure that constraints become ground (and thus easy to solve) during runtime once all conditional facts have been satisfied.

3 Authorization queries

Authorization requests are decided by querying an assertion context (containing local as well as imported assertions). In SecPAL, *authorization queries* consist of atomic queries of the form A says *fact* and constraints, combined by logical connectives including negation:

$$
\begin{array}{rcl}
q & ::= & e \textsf{ says } \mathit{fact} \\
 & | & q_1, \; q_2 \\
 & | & q_1 \textsf{ or } q_2 \\
 & | & \textsf{not}(q) \\
 & | & c
\end{array}
$$

The resulting query language is more expressive than in other logic-based languages where only atomic queries are considered. For example, separation of duties, threshold and denying policies can be expressed by composing atomic queries with negation and constraints (see Section 4). We do not allow negation in the assertion language, as coupling negation with a recursive language results in semantic ambiguities [45], and often to higher computational complexity or undecidability [38]. By introducing the level of authorization queries and restricting the use of negation to this level, we effectively uncouple negation from recursion, thereby circumventing the problems usually associated with negation.

The semantics of queries is defined by the relation $AC, \theta \vdash q$. In the following, let ϵ be the empty substitution.

$$
\begin{array}{ll}
AC, \theta \vdash e \textsf{ says } \mathit{fact} & \text{if } AC, \infty \models e\theta \textsf{ says } \mathit{fact}\theta, \text{ and } \mathit{dom}(\theta) \subseteq \mathit{vars}(e \textsf{ says } \mathit{fact}) \\
AC, \theta_1\theta_2 \vdash q_1, \; q_2 & \text{if } AC, \theta_1 \vdash q_1 \text{ and } AC, \theta_2 \vdash q_2\theta_1 \\
AC, \theta \vdash q_1 \textsf{ or } q_2 & \text{if } AC, \theta \vdash q_1 \text{ or } AC, \theta \vdash q_2 \\
AC, \epsilon \vdash \textsf{not}(q) & \text{if } AC, \epsilon \nvdash q \text{ and } \mathit{vars}(q) = \emptyset \\
AC, \epsilon \vdash c & \text{if } \models c
\end{array}
$$

One can easily verify that $AC, \theta \vdash q$ implies $\mathit{dom}(\theta) \subseteq \mathit{vars}(q)$. Note that conjunction is defined to be non-commutative, as the second query may be instantiated by the outcome of the first query.

Given a query q and an authorization context AC, an authorization algorithm should return the set of all substitutions θ such that $AC, \theta \vdash q$. If the query is ground, the answer set will be either empty (meaning "no") or be a singleton set containing the empty substitution ϵ (meaning "yes"). In the general case, i.e. if the query contains variables, the substitutions in the answer set are all the variable assignments that make the query true. This is more informative than just returning "yes, the query can be satisfied". Section 7 gives an algorithm for finding this set of substitutions.

Safety We now define a safety condition on authorization queries to guarantee that the set of answer substitutions is finite, given that the assertions in the assertion context are safe. Furthermore, the condition ensures that subqueries of the form $\textsf{not}(q)$ or c will be ground at evaluation time, assuming a left-to-right evaluation rule with propagation for conjunctions, as is defined in Section 7.

We first define a deduction relation $\Vdash$ with judgments of the form $I \Vdash q : O$ where q is a query and I, O are sets of variables. Intuitively, the set I represents the variables that are

grounded by the context of the subquery, and O represents the variables that are grounded by the subquery.

$$\frac{\mathit{fact} \text{ is flat}}{I \Vdash e \textsf{ says } \mathit{fact} : \mathit{vars}(e \textsf{ says } \mathit{fact}) - I} \qquad \frac{I \Vdash q_1 : O_1 \quad I \cup O_1 \Vdash q_2 : O_2}{I \Vdash q_1,\ q_2 : O_1 \cup O_2}$$

$$\frac{I \Vdash q_1 : O_1 \quad I \Vdash q_2 : O_2}{I \Vdash q_1 \textsf{ or } q_2 : O_1 \cap O_2} \qquad \frac{\mathit{vars}(q) \subseteq I}{I \Vdash \textsf{not}(q) : \emptyset} \qquad \frac{\mathit{vars}(c) \subseteq I}{I \Vdash c : \emptyset}$$

Definition 3.1. (Authorization query safety) An authorization query q is *safe* iff there exists a set of variables O such that $\emptyset \Vdash q : O$.

Checking safety can be done by recursively traversing all subqueries and thereby constructing the set O (which is uniquely determined by the query and I).

Examples

Safe queries	**Unsafe queries**
A says C can `read Foo`	A says B can say C can `read Foo`
x says y can a b, $x = A$	$x = A$, x says y can a b
x says y can a b, y says z can a b, $x \neq y$	x says y can a b, y says z can a b, $x \neq w$
(x says y can a b or y says x can a b), $x \neq y$	(x says y can a b or y says z can a b), $x \neq y$
x says y can a b, not(y says x can a b)	x says y can a b, not(y says z can a b)

Authorization query tables Conceptually, authorization queries are part of the local policy and should be kept separate from imperative code. In SecPAL, authorization queries belonging to a local assertion context are kept in a single place, the authorization query table. The table provides an interface to authorization queries by mapping parameterized method names to queries. Upon a request, the resource guard calls a method (instead of issuing a query directly) that gets mapped by the table to an authorization query, which is then used to query the assertion context.

For example, an authorization query table could contain the mapping:

`canAuthorizePayment`(*requester*, *payment*) :
  `Admin` says *requester* possesses `BankManagerID` *id*,
  not(`Admin` says *requester* has initiated *payment*)

If `Alice` attempts to authorize the payment `Payment47`, say, the resource guard calls `canAuthorizePayment(Alice, Payment47)`, which triggers the query

`Admin` says `Alice` possesses `BankManagerID` *id*,
not(`Admin` says `Alice` has initiated `Payment47`).

The resulting answer set (either an empty set if the request should be denied or a variable assignment for *id*) is returned to the resource guard who can then enforce the policy.

4 Policy idioms

In this section, we give examples of both assertions and authorization queries to show how SecPAL can be used to encode a number of well-known policy idioms.

Discretionary Access Control (DAC) The following assertion specifies that users can pass on their access rights to other users at their own discretion.

`Admin` says *user* can say$_\infty$ x can *access resource* if
  user can *access resource*

For example, if it follows from the assertion context that `Admin` says `Alice` can `read` `file://docs/` and `Alice` says `Bob` can `read` `file://docs/`, then `Admin` says `Bob` can `read` `file://docs/`. Even a policy as simple as this one cannot be expressed in languages without recursion such as XACML.

Mandatory Access Control (MAC) We assume a finite set of users U and a finite set of files S, characterised by the verb phrases is a user and is a file. Additionally, every such user and file is associated with a label from an ordered set of security levels. The constraint domain uses the function level to retrieve these labels, and the relation $\leq$ to represent the ordering. The following two assertions implement the Simple Security Property and the *-Property from the Bell-LaPadula model [6], respectively.

`Admin` says x can `read` f if
  x is a user,
  f is a file,
  level(x) $\geq$ level(f)

`Admin` says *user* can `write` *file* if
  x is a user,
  f is a file,
  level(x) $\leq$ level(f)

Role hierarchies The can act as keyword can be used to express role membership as well as role hierarchies in which roles inherit all privileges of less senior roles. The following example models a part of the hierarchy of medical careers in the UK National Health Service.

`HealthService` says `FoundationTrainee` can `read` `file://docs/`

`HealthService` says `SpecialistTrainee` can act as `FoundationTrainee`

`HealthService` says `SeniorMedPractitioner` can act as `SpecialistTrainee`

`HealthService` says `Alice` can act as `SeniorMedPractitioner`

The first assertion assigns a privilege to a role; the second and third establish seniority relations between roles; and the last assertion assigns Alice the role of a Senior Medical Practitioner. From these assertions it follows that `Admin` says `Alice` can `read` `file://docs/`. This example illustrates that SecPAL principals can represent roles as well as individuals; the principal `FoundationTrainee` is a role, while the principal `Alice` is an individual.

Parameterized attributes Parameterized roles can add significant expressiveness to a role-based system and reduce the number of roles [21, 33]. In SecPAL, parameterized roles, attributes and privileges can be encoded by introducing verb phrases. The following example uses the verb phrases can access health record of [−] and is a treating clinician of [−].

> `HealthService` says x can access health record of *patient* if
> x is a treating clinician of *patient*

Separation of duties In this simple example of separation of duties, a payment transaction proceeds in two phases, initiation and authorization, which are required to be executed by two distinct bank managers. The following shows a fragment of the authorization query table. The method `canInitiatePayment`(R, P) is called by the resource guard when a principal R attempts to initialize a payment P. If this is successful, the resource guard adds `Admin` says R has initiated P to the local assertion context. The method `canAuthorizePayment` is called when a principal attempts to authorize a payment. This is mapped to an authorization query that includes a negated atomic query that checks that the requester has not initiated the payment.

> `canInitiatePayment`(*requester*, *payment*) :
> `Admin` says *requester* possesses `BankManagerID` *id*
>
> `canAuthorizePayment`(*requester*, *payment*) :
> `Admin` says *requester* possesses `BankManagerID` *id*,
> not(`Admin` says *requester* has initiated *payment*)

Threshold-constrained trust SPKI/SDSI has the concept of k-of-n threshold subjects (at least k out of n given principals must sign a request) to provide a fault tolerance mechanism. RT^T has the language construct of "threshold structures" for similar purposes [32]. There is no need for a dedicated threshold construct in SecPAL, because threshold constraints can be expressed directly. In the following example, Alice trusts a principal if that principal is trusted by at least three distinct, trusted principals.

> `Alice` says x is trusted by `Alice` if
> x is trusted by a,
> x is trusted by b,
> x is trusted by c,
> $a \neq b,\ b \neq c,\ a \neq c$
>
> `Alice` says x can say$_\infty$ y is trusted by x if
> x is trusted by `Alice`

Hierarchical resources Suppose the assertion

> `Admin` says `Alice` can `read file://docs/`

is supposed to mean that Alice has read access to the path `/docs/` as well as to all descendants of that path. For example, Alice's request to read `/docs/foo/bar.txt` should be granted. To encode this, one might try to write

`Admin` says `Alice` can `read` x if x matches `file://docs/*`

Unfortunately, this assertion is not safe. Instead, we can stick with the original assertion and use the path constraint $\preceq$ inside queries:

`canRead`(*requester*, *path*) :
    `Admin` says *requester* can `read` *path2*, *path* $\preceq$ *path2*

The same technique can be used in conditional facts. With the following assertion, users can not only pass on their access rights, but also access rights to specific descendants; Alice could then for example delegate read access for `file://docs/foo/`.

`Admin` says *user* can say$_\infty$ x can *access path* if
    user can *access path2*,
    path $\preceq$ *path2*

The support of hierarchical resources is a very common requirement in practice, but existing policy languages cannot express the example shown above. For example, in RT^C [31], the $\preceq$ relation cannot take two variable arguments, because it only allows unary constraints in order to preserve tractability. Again, it is SecPAL's safety conditions that allow us to use such expressive constraint domains without losing efficiency.

Attribute-based delegation Attribute-based (as opposed to identity-based) authorization enables collaboration between parties whose identities are initially unknown to each other. The authority to assert that a subject holds an attribute (such as being a student) may then be delegated to other parties, who in turn may be characterised by attributes rather than identity.

In the example below, students are entitled to a discount. The expiration date of the student attribute can be checked with a constraint. The authority over the student attribute is delegated to holders of the university attribute, and authority over the university attribute is delegated to a known principal, the Board of Education.

`Admin` says x is entitled to discount if
    x is a student till *date*,
    currentTime() $\leq$ *date*

`Admin` says *univ* can say$_\infty$ x is a student till *date* if
    univ is a university

`Admin` says `BoardOfEducation` can say$_\infty$ *univ* is a university

Constrained delegation Delegators may wish to restrict the parameters of the delegated fact. In SecPAL, this can be done with constraints. In the example below, a Security Token Server (STS) is given the right to issue tickets for accessing some resource for a specified validity period of no longer than eight hours.

> Admin says STS can say$_\infty$ x has access from $t1$ till $t2$ if
> $t2 - t1 \leq 8$ hours

The delegation depth specified in the assertion above is unlimited, so STS can in turn delegate the same right to some STS2, possibly with additional constraints. With STS's assertion below, Admin will accept tickets issued by STS2 with a validity period of at most eight hours, where the start date is not before 01/01/2007.

> STS says STS2 can say$_0$ x has access from $t1$ till $t2$ if
> $t1 \geq$ 01/01/2007

Depth-bounded delegation The delegation-depth subscript of the can say keyword can only be 0 (no re-delegation) and ∞ (unlimited re-delegation). This might seem restrictive at first sight. However, SecPAL can express any fixed integer delegation depth by nesting can say. In the following example, Alice delegates the authority over is a friend facts to Bob and allows Bob to re-delegate one level further.

> Alice says Bob can say$_0$ x is a friend
>
> Alice says Bob can say$_0$ x can say$_0$ y is a friend

Suppose Bob re-delegates to Charlie with the assertion Bob says Charlie can say$_\infty$ x is a friend. Now, Alice says Eve is a friend follows from Charlie says Eve is a friend. Since Alice does not accept any longer delegation chains, Alice (in contrast to Bob) does not allow Charlie to re-delegate with

> Charlie says Doris can say$_0$ x is a friend

Furthermore, Charlie cannot defeat the delegation depth restriction with the following trick either, because the restriction is propagated to conditional facts.

> Charlie says x is a friend if x is Doris' friend
> Charlie says Doris can say$_0$ x is Doris' friend

If the only assertions by Alice and Bob that mention the predicate is a friend are those listed above, it is guaranteed that the result of the query Alice says x is a friend depends only on Charlie's assertions—not those of Doris for instance. Previous languages cannot express such a policy with this guarantee, either because they do not support delegation depth at all, or their depth restriction can be defeated with the trick above.

Width-bounded delegation Suppose Alice wants to delegate authority over is a friend facts to Bob. She does not care about the length of the delegation chain, but she requires every delegator in the chain to satisfy some property, e.g. to possess an email address from `fabrikam.com`. The following assertions implement this policy by encoding constrained transitive delegation using the can say keyword with a 0 subscript. Principals with the is a delegator attribute are authorized by Alice to assert is a friend facts, and to transitively re-delegate this attribute, but only amongst principals with a matching email address.

> `Alice` says x can say$_0$ y is a friend if
> x is a delegator
>
> `Alice` says Bob is a delegator
>
> `Alice` says x can say$_0$ y is a delegator if
> x is a delegator,
> y possesses `Email` *email*,
> *email* matches `*@fabrikam.com`

If these are the only assertions by `Alice` that mention the predicate is a friend or is a delegator, then any derivation of `Alice` says x is a friend can only depend on `Bob` or principals with a matching email address. As with depth-bounded delegation, this property cannot be enforced in previous languages.

5 Translation into constrained Datalog

The SecPAL assertion semantics is defined by the three deduction rules of Section 2. This semantics is more comprehensible and intuitive than one defined in terms of a translation into some form of formal logic, as in e.g. [29, 26, 32, 14, 31, 22, 5]. Nevertheless, it is useful to be able to efficiently translate SecPAL assertion contexts into equivalent Datalog programs. We can then exploit known complexity results (polynomial data complexity) and use the translated Datalog program for query evaluation, as described in Section 6.

Our terminology for constrained Datalog is as follows. (See [9] or [2] for an introduction to Datalog and [39, 38] for constrained Datalog.) A *literal*, P, consists of a *predicate name* plus an ordered list of *parameters*, each of which is either a variable or a constant. A literal is *ground* if and only if it contains no variables. A *clause*, written $P' \leftarrow P_1, \ldots, P_n, c$, consists of a *head* and a *body*. The head, P', is a literal. The body consists of a list of literals, $P_1, \ldots, P_n$, plus a constraint, c. We assume the sets of variables, constants, and constraints are the same as for SecPAL. A Datalog *program*, $\mathcal{P}$, is a finite set of clauses.

The semantics of a program $\mathcal{P}$ is the least fixed point of the following operator $T_{\mathcal{P}}$.

Definition 5.1. (Consequence operator) The *immediate consequence operator* $T_{\mathcal{P}}$ is a

function between sets of ground literals and is defined as:

$$T_{\mathcal{P}}(I) = \{P'\theta \mid (P' \leftarrow P_1, ..., P_n, c) \in \mathcal{P}, \\ P_i\theta \in I \text{ for each } i, \\ c\theta \text{ is valid}, \\ c\theta \text{ and } P'\theta \text{ are ground }\}$$

The operator $T_{\mathcal{P}}$ is monotonic and continuous, and its least fixed point $T^{\omega}_{\mathcal{P}}(\emptyset)$ contains all ground literals deducible from $\mathcal{P}$.

We now describe an algorithm for translating an assertion context into an equivalent constrained Datalog program. We treat expressions of the form e_1 says_k *fact* as Datalog literals, where k is either a variable or 0 or ∞. This can be seen as a sugared notation for a literal where the predicate name is the string concatenation of all infix operators (says, can say, can act as, and predicates) occurring in the expression, including subscripts for can say. The arguments of the literal are the collected expressions between these infix operators. For example, the expression A says_k x $\mathsf{can\ say}_\infty$ y $\mathsf{can\ say}_0$ B can act as z is shorthand for says_can_say_infinity_can_say_zero_can_act_as(A, k, x, y, B, z).

Algorithm 5.2. The translation of an assertion context *AC* proceeds as follows:

1. If $fact_0$ is flat, then an assertion A says $fact_0$ if $fact_1, ..., fact_n, c$ is translated into the clause A says_k $fact_0 \leftarrow A$ says_k $fact_1, ..., A$ says_k $fact_n, c$ where k is a fresh variable.

2. Otherwise, $fact_0$ is of the form e_0 $\mathsf{can\ say}_{K_0}$ ... e_{n-1} $\mathsf{can\ say}_{K_{n-1}}$ *fact*, for some $n \geq 1$, where *fact* is flat. Let $\hat{fact}_n \equiv fact$ and $\hat{fact}_i \equiv e_i$ $\mathsf{can\ say}_{K_i}$ $\hat{fact}_{i+1}$, for $i \in \{0..n-1\}$. Note that $fact_0 = \hat{fact}_0$. Then the assertion A says $fact_0$ if $fact_1, ..., fact_m, c$ is translated into a set of $n + 1$ Datalog rules as follows.

   (a) We add the Datalog rule A says_k $\hat{fact}_0 \leftarrow A$ says_k $fact_1, ..., A$ says_k $fact_m, c$ where k is a fresh variable.

   (b) For each $i \in \{1..n\}$, we add a Datalog rule

   $$A \ \mathsf{says}_\infty \ \hat{fact}_i \leftarrow \\ x \ \mathsf{says}_{K_{i-1}} \ \hat{fact}_i, \\ A \ \mathsf{says}_\infty \ x \ \mathsf{can\ say}_{K_{i-1}} \ \hat{fact}_i$$

   where x is a fresh variable.

3. Finally, for each Datalog rule created above of the form A says_k e *verbphrase* $\leftarrow$ ... we add a rule

   $$A \ \mathsf{says}_k \ x \ verbphrase \leftarrow \\ A \ \mathsf{says}_k \ x \ \mathsf{can\ act\ as} \ e, \\ A \ \mathsf{says}_k \ e \ verbphrase$$

   where x is a fresh variable. Note that k is not a fresh variable, but either a constant or a variable taken from the original rule.

Example For example, the assertion

$$A \text{ says } B \text{ can say}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \text{ if } y \text{ can } \mathtt{read}\ \mathtt{Foo}$$

is translated into

$$A \text{ says}_k\ B \text{ can say}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \leftarrow A \text{ says}_k\ y \text{ can } \mathtt{read}\ \mathtt{Foo}$$

$$\begin{aligned} & A \text{ says}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \leftarrow \\ & \quad x \text{ says}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z\ , \\ & \quad A \text{ says}_\infty\ x \text{ can say}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \end{aligned}$$

$$\begin{aligned} & A \text{ says}_\infty\ C \text{ can } \mathtt{read}\ z \leftarrow \\ & \quad x \text{ says}_0\ C \text{ can } \mathtt{read}\ z\ , \\ & \quad A \text{ says}_\infty\ x \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \end{aligned}$$

in Steps 2a and 2b. Finally, in Step 3, the following rules are also added:

$$\begin{aligned} & A \text{ says}_k\ x \text{ can say}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \leftarrow \\ & \quad A \text{ says}_k\ x \text{ can act as } B, \\ & \quad A \text{ says}_k\ B \text{ can say}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \end{aligned}$$

$$\begin{aligned} & A \text{ says}_\infty\ x \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \leftarrow \\ & \quad A \text{ says}_k\ x \text{ can act as } y, \\ & \quad A \text{ says}_\infty\ y \text{ can say}_0\ C \text{ can } \mathtt{read}\ z \end{aligned}$$

$$\begin{aligned} & A \text{ says}_\infty\ x \text{ can } \mathtt{read}\ z \leftarrow \\ & \quad A \text{ says}_k\ x \text{ can act as } C, \\ & \quad A \text{ says}_\infty\ C \text{ can } \mathtt{read}\ z \end{aligned}$$

Intuitively, the says subscript keeps track of the delegation depth, just like the D in the three semantic rules in Section 2. This correspondence is reflected in the following theorem that relates the Datalog translation to the SecPAL semantics.

Theorem 5.3. (Soundness and completeness) Let $\mathcal{P}$ be the Datalog translation of the assertion context AC. $A \text{ says}_D\ fact \in T_{\mathcal{P}}^{\omega}(\emptyset)$ iff $AC, D \models A \text{ says } fact$.

6 Datalog evaluation with tabling

In the section above, we showed how a set of SecPAL assertions can be translated into an equivalent constrained Datalog program. This section deals with the problem of evaluating such a program, that is, given a query of the form $e \text{ says}_\infty\ fact$, finding all instances of the query that are in $T_{\mathcal{P}}^{\omega}(\emptyset)$.

In the context of deductive databases, the bottom-up approach is most often used for Datalog evaluation. There, the program's model (i.e., its least fixed-point) is computed once and for all. This has the advantage that it is a complete, terminating procedure, and query evaluation is fast once the fixed-point has been constructed.

However, bottom-up evaluation is not suitable for SecPAL, as the assertion context is not constant. In fact, it may be completely different between different requests. Computing the model for every request is not efficient as it results in the evaluation of irrelevant goals. Furthermore, the queries we are interested in are usually fully or partially instantiated, so a top-down, goal-directed approach seems more appropriate.

The most widely-known top-down evaluation algorithm is SLD resolution, as used for Prolog. Unfortunately, SLD resolution can run into infinite loops even for safe Datalog programs, if some of the predicates have recursive definitions, as is the case for can say or can act as assertions. The problem remains even if instead of a depth-first a breadth-first search strategy is employed: the looping occurs because the SLD search tree is infinite. Tabling, or memoing, is an efficient approach for guaranteeing termination by incorporating some bottom-up techniques into a top-down resolution strategy. The basic idea is to prune infinite search trees by keeping a table of encountered subgoals and their answers, and to compute a subgoal only if it is not already in the table [42, 16, 11].

We present here a deterministic algorithm based on tabling. The algorithm constructs a directed acyclic proof graph by assembling and connecting nodes. A node can either be a *root node* of the form $\langle P \rangle$, where P is a literal, or it can be a sextuple $\langle P; \vec{Q}; c; S; \vec{nd}; Rl \rangle$ where the literal P is its *index*, the list of literals $\vec{Q}$ its *subgoals*, c its *constraint*, the literal S its *partial answer*, the list of nodes $\vec{nd}$ its *child nodes*, and Rl its *rule*. If the list of subgoals is nonempty, its head is the node's *current subgoal*. A node with an empty list of subgoals and a true constraint (i.e., of the form $\langle _; [\,]; \texttt{True}; S; _; _ \rangle$) is an *answer node*, and S is its *answer*. Intuitively, the list of subgoals (originally taken from the body of rule Rl) contains the literals that still have to be solved for the query indexed by P. The subgoals are solved from left to right, hence the head of the list is the current subgoal. When the current subgoal can be resolved against another answer node, the latter is added to the list of child nodes. This may entail instantiations of variables which will narrow down both the constraint c and the partial answer S. The partial answer may eventually become a proper answer if all subgoals have been solved and the constraint is valid.

The algorithm makes use of two tables. The *answer table* Ans is a mapping from literals to sets of answer nodes. The *wait table* $Wait$ is a mapping from literals to sets of nodes with nonempty lists of subgoals. $Ans(P)$ contains all answer nodes pertaining to the query $\langle P \rangle$ found so far. $Wait(P)$ is a list of 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.

Before presenting the algorithm in detail, we define a number of terms. *simplify* denotes a function on constraints whose return value is always an equivalent constraint, and if the argument is ground, the return value is either True or False. The infix operators :: and @ denote the cons and the append operations on lists, respectively. Let P and Q range over literals. A *variable renaming* for P is an injective substitution whose range consists only of variables. A *fresh renaming* of P is a variable renaming for P such that the variables in its range have not appeared anywhere else. A substitution θ is *more general* than θ' iff there exists a substitution ρ such that $\theta' = \theta\rho$. A substitution θ is a *unifier* of P and Q

```
Resolve-Clause(⟨P⟩)
  Ans(P) := ∅;
  foreach (Q ← Q⃗, c) ∈ 𝒫 do
      if nd = resolve(⟨P; Q :: Q⃗; c; Q; [ ]; Rl⟩, P) exists then
        Process-Node(nd)

Process-Answer(nd)
  match nd with ⟨P; [ ]; c; _; _; _⟩ in
    if nd ∉ Ans(P) then
      Ans(P) := Ans(P) ∪ {nd};
    foreach nd′ ∈ Wait(P) do
      if nd″ = resolve(nd′, nd) exists then
        Process-Node(nd″)

Process-Node(nd)
  match nd with ⟨P; Q⃗; c; _; _; _⟩ in
    if Q⃗ = [ ] then
      Process-Answer(nd)
    else match Q⃗ with Q_0 :: _ in
      if there exists Q′_0 ∈ dom(Ans)
            such that Q_0 ⇒ Q′_0 then
        Wait(Q′_0) := Wait(Q′_0) ∪ {nd};
        foreach nd′ ∈ Ans(Q′_0) do
          if nd″ = resolve(nd, nd′) exists then
            Process-Node(nd″)
      else
        Wait(Q_0) := {nd};
        Resolve-Clause(⟨Q_0⟩)
```

Figure 1: A tabled resolution algorithm for evaluating Datalog queries.

iff $P\theta = Q\theta$. A substitution θ is a *most general unifier* of P and Q iff it is more general than any other unifier of P and Q. When P and Q are unifiable, they also have a most general unifier that is unique up to variable renaming. We denote it by $mgu(P, Q)$. Finding the most general unifier is relatively simple (see [27] for an overview) but there are more intricate algorithms that run in linear time, see e.g. [37, 34]. P is an *instance* of Q iff $P = Q\theta$ for some substitution θ, in which case we write $P \Rightarrow Q$.

A node $nd \equiv \langle P; Q :: \vec{Q}; c; S; \vec{nd}; Rl\rangle$ and a literal Q' are *resolvable* iff Q'' is a fresh renaming of Q', $\theta \equiv mgu(Q, Q'')$ exists and $d \equiv simplify(c\theta) \neq \texttt{False}$. Their resolvent is $nd'' \equiv \langle P; \vec{Q}\theta; d; S\theta; \vec{nd}; Rl\rangle$, and θ is their *resolution unifier*. We write $resolve(nd, Q') = nd''$ if nd and Q' are resolvable. By extension, a node $nd \equiv \langle P; Q :: \vec{Q}; c; S; \vec{nd}; Rl\rangle$ and an answer node $nd' \equiv \langle _; [\,]; \texttt{True}; Q'; _; _\rangle$ are *resolvable* iff nd and Q' are resolvable with resolution unifier θ, and their *resolvent* is $nd'' \equiv \langle P; \vec{Q}\theta; d; S\theta; \vec{nd}@[nd']; Rl\rangle$. We write $resolve(nd, nd') = nd''$ if nd and nd' are resolvable.

Figure 1 shows the pseudocode of our Datalog evaluation algorithm. Let P be a literal and Ans be an answer table. $Answers_{\mathcal{P}}(P, Ans)$ is defined as

$$\{\theta \;:\; \langle _; _; _; S; _; _\rangle \in Ans(P'), S = P\theta, dom(\theta) \subseteq vars(P)\}$$

if there exists a literal $P' \in dom(Ans)$ such that $P \Rightarrow P'$. In other words, if the supplied answer table already contains a suitable answer set, we can just return the existing answers. If no such literal exists in the domain of Ans and if the call Resolve-Clause($\langle P\rangle$) terminates with initial answer table Ans and an initially empty wait table, then $Answers_{\mathcal{P}}(P, Ans)$ is defined as

$$\{\theta \;:\; \langle _; _; _; S; _; _\rangle \in Ans'(P), S = P\theta, dom(\theta) \subseteq vars(P)\}$$

where Ans' is the modified answer table after the call. In all other cases $Answers_{\mathcal{P}}(P, Ans)$ is undefined. Theorem 6.2 states the termination, soundness and completeness properties of $Answers_{\mathcal{P}}$. These properties will be exploited in Section 7 where we present an algorithm for evaluating composite authorization queries.

At first sight, completeness with respect to $T^{\omega}_{\mathcal{P}}(\emptyset)$ depends on $\mathcal{P}$ being Datalog-safe, i.e., all variables in the head literal must occur in a body literal. However, the translation of a safe SecPAL assertion context does not always result in a Datalog-safe program. We define an alternative notion of safety for Datalog programs that is satisfied by the SecPAL translation and that still preserves completeness. Every parameter position of a predicate is associated with either IN or OUT. A Datalog clause is IN/OUT-*safe* iff any OUT-variable in the head also occurs as an OUT-variable in the body, and any IN-variable in a body literal also occurs as IN-variable in the head or as OUT-variable in a preceding body literal. A Datalog program $\mathcal{P}$ is IN/OUT-*safe* iff all its clauses are IN/OUT-safe. A query P is IN/OUT-*safe* iff all its IN-parameters are ground.

Theorem 6.1. If *AC* is a safe assertion context and $\mathcal{P}$ its Datalog translation then there exists an IN/OUT assignment to predicate parameters in $\mathcal{P}$ such that $\mathcal{P}$ is IN/OUT-safe.

An answer table *Ans* is *sound* (with respect to some program $\mathcal{P}$) iff for all $P \in dom(Ans)$: $\langle P'; [\,]; \texttt{True}; S; _; _\rangle \in Ans(P)$ implies $P = P'$, $S \Rightarrow P$, and $S \in T^{\omega}_{\mathcal{P}}(\emptyset)$. *Ans* is *complete* (with respect to $\mathcal{P}$) iff for all $P \in dom(Ans)$: $S \in T^{\omega}_{\mathcal{P}}(\emptyset)$ and $S \Rightarrow P$ implies that S is the answer of an answer node in $Ans(P)$. Note, in particular, that the empty answer table is sound and complete.

Theorem 6.2. (soundness, completeness, termination) Let *Ans* be a sound and complete answer table, $\mathcal{P}$ an IN/OUT-safe program and P an IN/OUT-safe query. Then $Answers_{\mathcal{P}}(P, Ans) = \Theta$ is defined, finite and equal to $\{\theta \; : \; P\theta \in T^{\omega}_{\mathcal{P}}(\emptyset), dom(\theta) \subseteq vars(P)\}$.

Actually, the modified answer table after evaluation is still sound and complete and contains enough information to reconstruct the complete proof graph for each answer: an answer node $\langle _; _; _; S; \vec{nd}; Rl\rangle$ is interpreted to have edges pointing to each of its child nodes $\vec{nd}$ and an edge pointing to the rule Rl. Appendix B shows how this Datalog proof graph can be converted back into a corresponding SecPAL proof graph.

7 Evaluation of authorization queries

Based on the algorithm from the previous section, we can now show how to evaluate complex authorization queries as defined in Section 3.

In the following, let *AC* be an assertion context and $\mathcal{P}$ its Datalog translation, and let ϵ be the empty substitution. We define the function $AuthAns_{AC}$ on authorization queries as follows.

$$\begin{array}{rcl}
AuthAns_{AC}(e \text{ says } fact) & = & Answers_{\mathcal{P}}(e \text{ says}_\infty\ fact,\ \emptyset) \\
AuthAns_{AC}(q_1,\ q_2) & = & \{\theta_1\theta_2 \mid \theta_1 \in AuthAns_{AC}(q_1) \text{ and } \theta_2 \in AuthAns_{AC}(q_2\theta_1)\} \\
AuthAns_{AC}(q_1 \text{ or } q_2) & = & AuthAns_{AC}(q_1) \cup AuthAns_{AC}(q_2) \\
AuthAns_{AC}(\text{not}(q)) & = & \begin{cases} \{\epsilon\} & \text{if } vars(q) = \emptyset \text{ and } AuthAns_{AC}(q) = \emptyset \\ \emptyset & \text{if } vars(q) = \emptyset \text{ and } AuthAns_{AC}(q) \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases} \\
AuthAns_{AC}(c) & = & \begin{cases} \{\epsilon\} & \text{if } \models c \\ \emptyset & \text{if } vars(c) = \emptyset \text{ and } \not\models c \\ \text{undefined} & \text{otherwise} \end{cases}
\end{array}$$

The following theorem shows that $AuthAns_{AC}$ is an algorithm for evaluating safe authorization queries.

Theorem 7.1. (Finiteness, soundness, and completeness of authorization query evaluation) For all safe assertion contexts AC and safe authorization queries q,

1. $AuthAns_{AC}(q)$ is defined and finite, and
2. $AC, \theta \vdash q$ iff $\theta \in AuthAns_{AC}(q)$.

The evaluation of the base case e says *fact* calls the function $Answers_{\mathcal{P}}$ with an empty answer table. But since the answer table after each call remains sound and complete (it will just have a larger domain), a more efficient implementation could initialize an empty table only for the first call in the evaluation of an authorization query, and then reuse the existing, and increasingly populated, answer table for each subsequent call to $Answers_{\mathcal{P}}$.

Finally, the following theorem states that SecPAL has polynomial data complexity (see e.g. [2, 13] for a definition and discussion of data versus program complexity). Data complexity is a sensible measure for the tractability of policy languages as the size of the extensional database (the number of "plain facts") typically exceeds the size of the intensional database (the number of "rules") by several orders of magnitude.

Theorem 7.2. Let M be the number of flat atomic assertions (i.e., those without conditional facts) in AC and N be the maximum length of constants occurring in these assertions. The time complexity of computing $AuthAns_{AC}$ is polynomial in M and N.

8 Discussion

Related work The earliest authorization policy languages to support the trust management paradigm were PolicyMaker and Keynote [8, 7]. Quite a few other policy languages have been developed since. SPKI/SDSI [18] is an experimental IETF standard using certificates to specify decentralized authorization. Authorization certificates grant permissions to subjects specified either as public keys, or as names defined via linked local name spaces

[41], or as k-out-of-n threshold subjects. Grants can have validity restrictions and indicate whether they may be delegated.

XrML [12] (and its offspring, MPEG-21 REL) is an XML-based language targeted at specifying licenses for Digital Rights Management. Grants may have validity restrictions and can be conditioned on other existing or deducible grants. A grant can also indicate under which conditions it may be delegated to others.

XACML [36] is another XML-based language for describing access control policies. A policy grants capabilities to subjects that satisfy the specified conditions. Deny policies explicitly state prohibitions. XACML defines a number of policy combination rules for resolving conflicts between permitting and denying policies such as First-Applicable, Deny-Override or Permit-Override. XACML does not support delegation and is thus not well suited for decentralized authorization.

The OASIS SAML [35] standards define XML formats and protocols for exchanging authenticated user identities, attributes, and authorization decisions, such as whether a particular subject has access to a particular object. SAML messages do not themselves express authorization rules or policies.

The original Globus security architecture [20] for grid computing defines a general security policy for subjects and objects belonging to multiple trust domains, with cross-domain delegation of access rights. More recent computational grids rely on specific languages, such as Akenti [43], Permis [10], and XACML, to define fine-grained policies, and exchange SAML or X.509 certificates to convey identity, attribute, and role information. Version 1.1 of XACML has a formal semantics [24] via a purely functional implementation in Haskell. To the best of our knowledge, XACML and SecPAL are the only authorization languages for grid computing that have formal semantics.

Policy languages such as Binder [14], SD3 [26], Delegation Logic (DL) [29] and the RT family of languages [32] use Datalog as basis for their syntax and semantics. To support attribute-based delegation, these languages allow predicates to be qualified by an issuing principal. Cassandra [5, 4] and RT^C [31] are based on Datalog with Constraints [25, 38] for higher expressiveness. The Cassandra framework also defines a transition system for evolving policies and supports automated credential retrieval and automated trust negotiation. Lithium [22] is a language for reasoning about digital rights and is based on a slightly different fragment of first order logic that can express negation, in the conclusion as well as in the premises of policy rules. It was not designed for decentralized trust management, however, and lacks dedicated features for controlling delegation.

Lampson, Burrows, Abadi, and Wobber [28] develop a theory of authentication and access control in which a "speaks for" relation defines when the right to assert a statement is delegated from one principal to another. SecPAL's "can say" relation can be seen as a specialized form of "speaks for" in their theory.

In the following, we discuss in detail how SecPAL compares with other languages.

Syntactic simplicity. Usability is one of the main obstacles in the industrial deployment of policy technology. Purely XML-based languages such as XACML or XrML are difficult

to read due to the verbosity of XML documents. In contrast, logic-based languages such as Cassandra, Binder, Lithium, SD3, or RT tend to have a more concise syntax. However, the users who would be writing the policies (e.g. system administrators) are usually not acquainted with formal logic, and are deterred by the high level of abstraction and the syntax based on formal logic found in these languages [3]. In contrast, the syntax of SecPAL essentially consists of simple, succinct sentences in English.

Semantic simplicity. To be practically usable, a policy language should not only have a simple, readable syntax, but also a simple, intuitive semantics. The XACML rule semantics is simple enough that a natural language specification may be sufficient, but at the expense of limited expressiveness. XrML is specified in terms of a very complex authorization algorithm [12, Section 5.8]. SPKI/SDSI does not have a formal foundation either. The unintended semantic complexities of policy languages without a precise semantics become apparent when a formalization is attempted retroactively, as in the case of post-hoc formalizations for XACML [24], XrML [23] or SPKI/SDSI [1, 30].

Lithium uses exactly the syntax and semantics of (a fragment of) first order logic. Binder, SD3 and RT are defined in terms of their translation into Datalog, and DL in terms of a translation into pure Prolog. Cassandra and RT^C are defined via translations into Datalog with Constraints. In contrast, the SecPAL assertion semantics is defined by the three deduction rules in Section 2 that directly reflect the intuition suggested by the syntax. We believe that our proof-theoretic approach enhances simplicity and clarity, far more than if we had instead taken the Datalog translation in Section 5 as the language specification.

Termination and tractability. A query evaluation algorithm must be sound and complete with respect to the language semantics, and must always terminate and be tractable. Halpern and Weissman have shown for a fragment of XrML that the given evaluation algorithm returns unintended results and is intractable. It was recently proved that the evaluation algorithms of XrML 2.0 and the related MPEG-REL 21 are not guaranteed to terminate[1]. The analysis in [30] shows that the algorithm for SPKI/SDSI is incomplete; in fact, the language is likely to be undecidable due to the complex structure of SPKI's authorization tags.

Binder, SD3 and RT are tractable as they are equivalent to Datalog. RT^C achieves tractability by allowing only simple, unary constraints. In Cassandra, termination and tractability properties depend on the complexity of the chosen constraint domain; establishing these properties formally can be tedious. In SecPAL, tractability is guaranteed by the purely syntactic (and thus simple) safety conditions, even if the chosen constraint domain would have resulted in undecidability or intractability with Cassandra or RT^C. Lithium is a tractable fragment of first order logic that is characterised by syntactic as well semantic conditions. In particular, a bipolarity condition restricts the polarity of predicates

[1] Personal communication, Vicky Weissman. To be published shortly.

in a formula written in conjunctive normal form.

Delegation and local names. Attribute-based delegation of authority between administrative domains is the main building block of authorization policies in distributed systems. XACML does not provide any constructs for expressing delegation. Lithium does not have any delegation constructs either, and it is also not possible in Lithium to encode recursive delegation because of the bipolarity restrictions.

In Binder, SD3, RT and Cassandra, delegation from A to B can be expressed as (in SecPAL-like syntax) "A says *fact* if B says *fact*". These languages also support linked local role name spaces (as in the SDSI part of SPKI/SDSI) that can be regarded as a form of delegation. For example, in RT one could write

Alice.friend $\leftarrow$ Alice.relative.spouse

to express that Alice regards all her relatives' spouses as friends but delegates the definition of the local name spouse to her relatives. This can be expressed in SecPAL as

Alice says x is a friend if
  r is a relative of Alice,
  x is a spouse of r
Alice says r can say$_\infty$ x is a spouse of r if
  r is a relative of Alice

Binder, SD3, RT and Cassandra cannot control re-delegation, e.g. they cannot express depth- or width-restricted delegation. SecPAL makes the delegation step explicit and thus allows for more fine-grained delegation control.

SPKI/SDSI has a boolean delegation depth flag that corresponds to the 0 or ∞ subscript in can say but cannot express any other integer delegation depths. In XrML and DL, the delegation depth can be specified, and can be either an integer or ∞. However, in both languages, the depth restrictions can be defeated. Consider the following three SecPAL assertions:

Alice says Bob can say$_0$ *fact*.
Bob says *fact* if *fact*$'$.
Bob says Charlie can say$_\infty$ *fact*$'$.

Alice delegates authority over *fact* to Bob, but disallows him to re-delegate. In XrML and DL, Bob could use policies corresponding to the second and third assertions to re-delegate *fact* to Charlie via *fact*$'$, thereby circumventing the depth specification. SecPAL semantics prevents this by threading the depth restriction through the entire branch of the proof; this is a corollary of Proposition 2.5. It would be much harder to design a semantics with this guaranteed property if the depth restriction could be any arbitrary integer; this is also why XrML and DL cannot be easily "fixed" to support integer delegation depth that is immune to this kind of attack. In SecPAL, arbitrary integer delegation depths can be safely expressed by nested can say$_0$ facts.

Constraints. Datalog is the basis of many policy languages such as DL, Binder, SD3, and RT. Datalog has many advantages. It closely resembles conditional sentences in natural language. It is recursive, monotonic, decidable and tractable. However, Datalog is not expressive enough in practice as it lacks function symbols and constraints. The more expressive Datalog with Constraints is taken as the basis for RT^C and Cassandra. Adding constraints to a language is nontrivial, as they not only affect the language's expressiveness, but also its computational complexity and decidability. In RT^C, tractability is achieved by restricting the constraint domain to certain subclasses of unary constraints. This is too restrictive in practice: some policies for example require inequality constraints or path constraints on two variables. The benchmark policy for electronic health records in [3] makes use of a much more powerful constraint domain for which evaluation is decidable but intractable in Cassandra. Another problem with Datalog with Constraints is that constraint domains must be equipped with operations such as satisfaction checking or existential quantifier elimination that can be hard to implement. Ease of implementation is crucial for the success of a standard.

The solution adopted for SecPAL is based on the observation that a wide range of constraints are used in authorization policies, but they are used in a very restricted way: they relate to variables whose values always become instantiated during runtime. Therefore, rather than restricting the expressiveness of the constraint domain as is done in previous approaches, we should make sure that constraints will be ground during runtime: this is taken care of by SecPAL's safety conditions. This approach preserves decidability and tractability while allowing more powerful constraints than in Cassandra or RT^C. Moreover, it also greatly simplifies the evaluation algorithm, thus making it much easier to implement.

Negation Some policies such as separation of duties inherently depend on a condition being false. Cassandra supports a very restricted form of negated body predicates (and the related aggregation operation). Lithium is the only policy language that can express real logical negation (as opposed to negation as failure) both as condition and as conclusion, which is useful for analysing merged policies. Most other policy languages with recursion do not allow negated conditions, as the combination of the two can increase the computational complexity and lead to non-unique models. Negation inside policy rules also leads to non-monotonicity; the consequences of such policies can be confusing and are generally hard to comprehend and to foresee.

Our solution is based on the observation that for most policies, the negated conditions can be effectively separated from recursion. We can do the recursive computation first, and then query for negated conditions at the end. Therefore, SecPAL *assertions* may not include negated conditional facts, but SecPAL *authorization queries* can consist of a composition of possibly negated atomic queries. Moving negations into the top-level authorization queries makes for clearer policies. Furthermore, SecPAL authorization queries could easily be extended by even more powerful composition operators such as aggregation (as in Cassandra), restricted universal quantification or threshold operators (as in RT^T [32]) without changing the assertion semantics and without affecting the complexity results.

Authorization queries can also express prohibitions similar to deny policies in XACML. For example, by defining a predicate cannot we could write a Deny-Overrides authorization query as follows:

$$Alice \text{ says } x \texttt{ can read } f, \ \text{not}(Alice \text{ says } x \texttt{ cannot read } f)$$

More elaborate conflict resolution rules such as assertions with different priorities could also be encoded on the level of authorization queries. Just as with negative conditions, prohibition makes policies less comprehensible and should be used sparingly, if at all [19, 15].

Conclusions We presented a policy language that we believe is simpler and more intuitive than existing languages, due to the resemblance of its syntax to natural language, its small semantic specification and its purely syntactic safety conditions. Despite its simplicity, SecPAL supports fine-grained delegation control and highly expressive constraints that are needed in practice but cannot be expressed in other languages. If authorization queries are extended by an aggregation operator (which can be done without modifying the assertion semantics and without sacrificing polynomial data complexity), SecPAL can express the entire benchmark policy in [3].

A SecPAL prototype, including an auditing infrastructure, is being implemented as part of a project investigating access control solutions for large-scale Grid computing environments. A primary focus of this effort is on developing flexible and robust mechanisms for expressing trust relationships and constrained delegation of rights within a uniform authentication and authorization framework. Scenarios, similar to the one described in Section 1, have been demonstrated using the prototype. At the time of writing, the prototype only supports atomic authorization queries.

Acknowledgements Blair Dillaway and Brian LaMacchia authored the original SecPAL design; the current definition is the result of many fruitful discussions with them. In a separate whitepaper, Dillaway [17] presents the design goals and introduces the language informally. Gregory Fee and Jason Mackay implemented the SecPAL prototype. Blair Dillaway suggested improvements to a draft of this paper. We also thank Sebastian Nanz for valuable discussions.

A Assertion expiration and revocation

In SecPAL, expiration dates can be expressed as ordinary verb phrase parameters:

$\texttt{UCambridge}$ says $\texttt{Alice}$ is a student till 31/12/2007 if currentTime() $\leq$ 31/12/2007

Sometimes it should be up to the acceptor to specify an expiration date or set their own recency requirements [40]. In this case, the assertion could just contain the date without enforcing it:

$\texttt{UCambridge}$ says $\texttt{Alice}$ is a student till 31/12/2007

An acceptor can then use the date to enforce their own recency requirements:

$\texttt{Admin}$ says x is entitled to discount if
  x is a student till $date$,
  currentTime() $\leq date$,
  $date -$ currentTime() ≤ 1 year

Assertions may have to be revoked before their scheduled expiration date. To deal with compromise of an issuer's key, we can use existing key revocation mechanisms. But sometimes the issuer needs to revoke their own assertions. For instance, the assertion in the example above has to be revoked if Alice drops out of her university.

We assume that every assertion M is associated with an identifier (e.g., a serial number) ID_M. Revocation (and delegation of revocation) can then be expressed in SecPAL by *revocation assertions* with the verb phrase revokes ID_M. For example, the revocation assertion

A says A revokes ID if currentTime() $>$ 31/7/2007

revokes all assertions that are issued by A and have identifier ID, but only after 31 July 2007.

Definition A.1. (revocation assertion) An assertion is a *revocation assertion* if it is safe and of the form

A says A revokes ID if c, or
A says B_1 can say$_{D_1}$ ... B_n can say$_{D_n}$ A revokes ID if c.

Given an assertion context AC and a set of revocation assertions AC_{rev} where $AC \cap AC_{rev} = \emptyset$, we remove all assertions revoked by AC_{rev} in AC before an authorization query is evaluated. The filtered assertion context is defined by

$$AC - \{M \mid M \in AC,\ A \text{ is the issuer of } M, \text{ and } AC_{rev}, \infty \models A \textsf{ says } A \textsf{ revokes } ID_M\}$$

The condition that AC and AC_{rev} must be disjoint means that revocation assertions cannot be revoked (at least not within the language). Allowing revocation assertions to be revoked by each other causes the same problems and semantic ambiguities as negated body predicates in logic programming. Although these problems could be formally overcome, for example by only allowing stratifiable revocation sets or by computing the well-founded model, these approaches are not simple enough for users to cope with in practice.

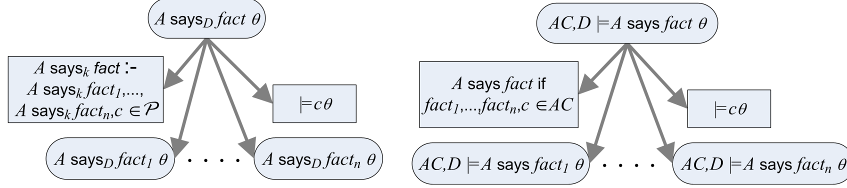

Figure 2: Left: Datalog proof node with parent from Translation Step 1 or 2a. Right: corresponding SecPAL proof node using Rule (cond).

B Proof graph generation

When testing and troubleshooting policies, it is useful to be able to see a justification of an authorization decision. This could be some visual or textual representation of a corresponding proof graph constructed according to the rule system in Section 2.

Given a Datalog program $\mathcal{P}$, a *proof graph* for $\mathcal{P}$ is a directed acyclic graph with the following properties. Leaf nodes are either Datalog clauses in $\mathcal{P}$ or ground constraints that are valid. Every non-leaf node is a ground instance $P'\theta$ of the head of a clause $P' \leftarrow P_1, \ldots, P_n, c$, where θ substitutes a constant for each variable occurring in the clause; the node has as child nodes the clause, the ground instances $P_1\theta, \ldots, P_n\theta$ of the body literals, and the ground instance $c\theta$ of the body constaint. A ground literal P occurs in $T^{\omega}_{\mathcal{P}}(\emptyset)$ if and only if there is a proof graph for $\mathcal{P}$ with P as a root node. The algorithm in Figure 1 constructs such a Datalog proof graph during query evaluation of the Datalog program $\mathcal{P}$ obtained by translating an assertion context AC. Each answer to a query is a root node of the graph. Every non-leaf node is a ground Datalog literal of the form A says_D fact. Leaf nodes are either Datalog clauses in the program $\mathcal{P}$, or ground constraints that are valid. (See left panels of Figures 2, 3 and 4.)

Similarly, we can define a notion of proof graph for SecPAL such that there is a derivation of $AC, \infty \models A$ says fact according to the three deduction rules of Section 2 if and only if there is a SecPAL proof graph with $AC, \infty \models A$ says fact as a root node.

If during execution of Algorithm 5.2, each generated Datalog clause is labelled with the algorithm step at which it was generated (i.e., 1, 2a, 2b, or 3), the Datalog proof graph contains enough information to be easily converted into the corresponding SecPAL proof graph. The conversion is illustrated in Figures 2, 3 and 4.

C Auxiliary definitions and proofs

C.1 Authorization queries

Lemma C.1. If $AC, \theta \vdash q$ then $\mathit{dom}(\theta) \subseteq \mathit{vars}(q)$, and θ grounds all $x \in \mathit{dom}(\theta)$.

Proof. By induction on the definition of $\vdash$. ☐

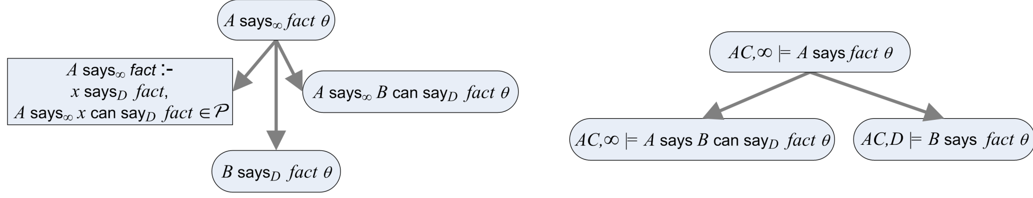

Figure 3: Left: Datalog proof node with parent from Translation Step 2b. Right: corresponding SecPAL proof node using Rule (can say).

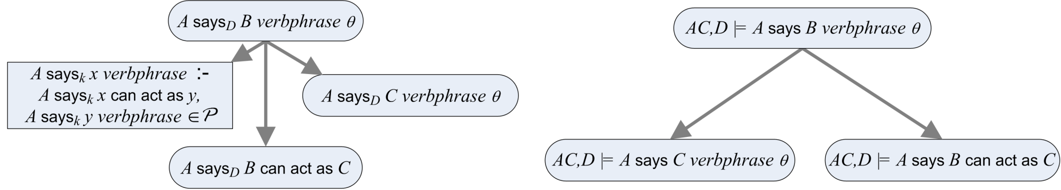

Figure 4: Left: Datalog proof node with parent from Translation Step 3. Right: corresponding SecPAL proof node using Rule (can act as).

Lemma C.2. If $AC, \theta \vdash e$ says $fact$ then $dom(\theta) = vars(e \text{ says } fact)$.

Proof. Follows immediately from the definitions of $\vdash$ and $\models$. □

Lemma C.3. If $I \Vdash q : O$ then for all substitution θ, $I - dom(\theta) \Vdash q\theta : O - dom(\theta)$.

Proof. By induction on q. □

Corollary C.4. If $I \Vdash q : O$ and $I \subseteq dom(\theta)$ then $q\theta$ is safe.

Lemma C.5. If $\emptyset \Vdash q : O$ and $AC, \theta \vdash q$ then $O \subseteq dom(\theta)$.

Proof. By induction on q.

Suppose $q \equiv e$ says $fact$. Then $O = vars(e \text{ says } fact) = dom(\theta)$, by Lemma C.2.

Suppose $q \equiv q_1,\ q_2$ and $\theta \equiv \theta_1\theta_2$. By the induction hypothesis, $O_1 \subseteq dom(\theta_1)$. Therefore, by Lemma C.3, $\emptyset \Vdash q_2\theta_1 : O_2 - dom(\theta_1)$. Then by the induction hypothesis, $O_2 - dom(\theta_1) \subseteq dom(\theta_2)$. Therefore, $O_1 \cup O_2 \subseteq dom(\theta_1\theta_2)$.

The other cases are straightforward. □

Lemma C.6. If $q_1,\ q_2$ is safe and $AC, \theta_1 \vdash q_1$ then $q_2\theta_1$ is safe.

Proof. From the definition of safety and $\Vdash$ it follows that $\emptyset \Vdash q_1,\ q_2 : O_1 \cup O_2$ where $\emptyset \Vdash q_1 : O_1$. By Lemma C.5, $O_1 \subseteq dom(\theta_1)$. Then by Corollary C.4, $q_2\theta_1$ is safe. □

C.2 Translation into constrained Datalog

Lemma C.7. (Soundness) Let $\mathcal{P}$ be the Datalog translation of the assertion context AC. If A says_D $fact$ $\in T_{\mathcal{P}}^{\omega}(\emptyset)$ then $AC, D \models A$ says $fact$.

Proof. We assume A says_D $fact$ $\in T_{\mathcal{P}}^{\omega}(\emptyset)$ and prove the statement by induction on stages of $T_{\mathcal{P}}^{n}$.

Case Step 1 and 2a If A says_D $fact$ is added based on a rule produced by Step 1 or 2a, then by the inductive hypothesis, $AC, D \models A$ says $fact_i\theta$ for $i = 1...n$. Furthermore, $c\theta$ is ground and valid, so by Rule (cond), $AC, D \models A$ says $fact$.

Case Step 2b If A says_∞ $fact$ is added based on a rule produced by Step 2b, then by the inductive hypothesis, $AC, K \models B$ says $fact$ and $AC, \infty \models A$ says B $\mathsf{can\ say}_K$ $fact$, for some B and K. By Rule (can say), $AC, \infty \models A$ says $fact$.

Case Step 3 If A says_D B $verbphrase$ is added based on a rule produced by Step 3, then by the inductive hypothesis, $AC, D \models A$ says B $\mathsf{can\ act\ as}$ C and $AC, D \models A$ says C $verbphrase$, for some C. By Rule (can act as), $AC, D \models B$ says C $verbphrase$. □

Lemma C.8. If $AC, D \models A$ says B $verbphrase$ then there exists an assertion in AC of the form

$$A \ \mathsf{says}\ e_1\ \mathsf{can\ say}_{D_1} ... e_n\ \mathsf{can\ say}_{D_n}\ e\ verbphrase'\ \mathsf{if}\ ...$$

for some e and e_i, for $i = 1...n$ where $n \geq 0$, and $verbphrase$ is an instance of $verbphrase'$.

Proof. By induction on the SecPAL rules. If the last rule used in the deduction of $AC, D \models A$ says B $verbphrase$ was (cond), there exists an assertion in AC of the form

$$A\ \mathsf{says}\ e\ verbphrase'\ \mathsf{if}\ ...$$

where B $verbphrase = (e\ verbphrase')\theta$.

If the last rule used was (can say), we have $AC, \infty \models A$ says B $\mathsf{can\ say}_D$ B $verbphrase$. Therefore, by the induction hypothesis, there exists an assertion in AC of the required form.

If the last rule used was (can act as), we have $AC, D \models A$ says C $verbphrase$. Therefore, by the induction hypothesis, there exists an assertion in AC of the required form. □

Lemma C.9. (Completeness) Let $\mathcal{P}$ be the Datalog translation of the assertion context AC. If $AC, D \models A$ says $fact$ then A says_D $fact$ $\in T_{\mathcal{P}}^{\omega}(\emptyset)$.

Proof. We assume $AC, D \models A$ says $fact$ and prove the statement by induction on the SecPAL rules.

Case (cond) If the last rule used in the deduction was (cond), (A says $fact$ if $fact_1, \ldots, fact_k, c) \in AC$ is translated in Step 1 or 2a. Also, $AC, D \models A$ says $fact_i\theta$, and by the induction hypothesis, A says$_D$ $fact_i\theta \;\in\in T_{\mathcal{P}}^{\omega}(\emptyset)$. Furthermore, $S \models c\theta$ and $vars(fact\theta) = \emptyset$, so by definition of $T_{\mathcal{P}}$, A says$_D$ $fact \;\in T_{\mathcal{P}}^{\omega}(\emptyset)$.

Case (can say) If Rule (can say) was used last, we assume

1. $D = \infty$,
2. $AC, K \models B$ says $fact$, and
3. $AC, \infty \models A$ says B can say$_K$ $fact$.

By Lemma C.8, there is an assertion in AC of the form

A says e_1 can say$_{D_1}$...e_n can say$_{D_n}$ e can say$_K$ $fact'$,

for some e, e_i, D_i with $i = 1...n$, $n \geq 0$ and where $fact$ is an instance of $fact'$. In Step 2b, this is translated into

A says$_\infty$ $fact \leftarrow$
x says$_K$ $fact'$,
A says$_\infty$ x can say$_K$ $fact'$

where x is a fresh variable not occurring anywhere else in the rule, so it can in particular bind to B. By the induction hypothesis, B says$_K$ $fact \in T_{\mathcal{P}}^{\omega}(\emptyset)$ and A says$_\infty$ B can say$_K$ $fact \in T_{\mathcal{P}}^{\omega}(\emptyset)$. By definition of $T_{\mathcal{P}}$, A says$_\infty$ $fact \;\in T_{\mathcal{P}}^{\omega}(\emptyset)$.

Case (can act as) If Rule (can act as) was used last, we assume $AC, D \models A$ says C $verbphrase$, and $AC, D \models A$ says B can act as C, where $fact = B$ $verbphrase$. By the induction hypothesis, A says$_D$ C $verbphrase \in T_{\mathcal{P}}^{\omega}(\emptyset)$. This is only possible if there is a rule in $\mathcal{P}$ of the form

A says$_k$ e $verbphrase' \leftarrow ...$

where C $verbphrase = (e$ $verbphrase')\theta$ for some θ. By Step 3, there must also be a rule in $\mathcal{P}$ of the form

A says$_k$ y $verbphrase' \leftarrow$
A says$_k$ y can act as e
A says$_k$ e $verbphrase'$

where y is a fresh variable not occurring anywhere else in the rule, so it can in particular bind to B. By the induction hypothesis, we also have A says$_D$ B can act as $C \in T_{\mathcal{P}}^{\omega}(\emptyset)$. Therefore, by definition of $T_{\mathcal{P}}$, A says$_D$ B $verbphrase \in T_{\mathcal{P}}^{\omega}(\emptyset)$. □

Restatement of Theorem 5.3. (Soundness and completeness) Let $\mathcal{P}$ be the Datalog translation of the assertion context AC. A says$_D$ $fact \;\in T_{\mathcal{P}}^{\omega}(\emptyset)$ iff $AC, D \models A$ says $fact$.

Proof. Follows from Lemmas C.7 and C.9. □

C.3 Datalog evaluation with tabling

The tabling evaluation algorithm in Section 6 can also be described as a non-deterministic labelled transition system. We present this system in the following because it is easier to prove properties for it than for the pseudocode in Figure 1. The results for the transition system apply also to the pseudocode, as the latter is a straight-forward implementation of the former.

A *state* is a triple $(\mathit{Nodes}, \mathit{Ans}, \mathit{Wait})$ where Nodes is a set of nodes, Ans is an answer table, and Wait is a wait table. A *path* is a series of 0 or more labelled transitions between states, as defined in the labelled transition system below. A state $\mathcal{S}'$ is *reachable* from a state $\mathcal{S}$ iff there is a path from $\mathcal{S}$ to $\mathcal{S}'$. In the following, let $A \uplus B$ denote the union of A and B with the side condition that the sets be disjoint. If Ans is a function mapping literals to sets of nodes, then $\mathit{Ans}[P \mapsto A]$ is a function that maps literals Q to $\mathit{Ans}(Q)$ if $Q \in \mathit{dom}(\mathit{Ans})$ and $Q \neq P$ and additionally maps P to A.

A state $(\mathit{Nodes}, \mathit{Ans}, \mathit{Wait})$ is an *initial state* iff $\mathit{Nodes} = \{\langle P \rangle\}$ for some IN/OUT-safe query P (with respect to the IN/OUT-safe program $\mathcal{P}$), Ans is sound and complete, and Wait is empty. A state $\mathcal{S}$ is a *final state* iff there is no state $\mathcal{S}'$ and no label ℓ such that $\mathcal{S} \xrightarrow{\ell} \mathcal{S}'$.

$$(\{\langle P \rangle\} \uplus \mathit{Nodes}, \mathit{Ans}, \mathit{Wait}) \xrightarrow{\mathit{ResolveClause}} (\mathit{Nodes} \cup \mathit{Nodes}', \mathit{Ans}[P \mapsto \emptyset], \mathit{Wait})$$
$$\text{if} \quad \mathit{Nodes}' = \{\mathit{nd} \ : \ \mathit{Rl} \equiv Q \leftarrow \vec{Q}, c \in \mathcal{P}, \ \mathit{nd} = \mathit{resolve}(\langle P; Q :: \vec{Q}; c; Q; [\,]; \mathit{Rl} \rangle, P) \text{ exists } \}$$

$$(\{\mathit{nd}\} \uplus \mathit{Nodes}, \mathit{Ans}, \mathit{Wait}) \xrightarrow{\mathit{PropagateAnswer}} (\mathit{Nodes} \cup \mathit{Nodes}', \mathit{Ans}[P \mapsto \mathit{Ans}(P) \cup \{\mathit{nd}\}], \mathit{Wait})$$
$$\text{if} \quad \mathit{nd} \equiv \langle P; [\,]; \mathtt{True}; _; _; _ \rangle, \quad \mathit{nd} \notin \mathit{Ans}(P), \quad \mathit{Nodes}' = \{\mathit{nd}'' \ : \ \mathit{nd}' \in \mathit{Wait}(P), \ \mathit{nd}'' = \mathit{resolve}(\mathit{nd}', \mathit{nd}) \text{ exists}\}$$

$$(\{\mathit{nd}\} \uplus \mathit{Nodes}, \mathit{Ans}, \mathit{Wait}) \xrightarrow{\mathit{RecycleAnswers}} (\mathit{Nodes} \cup \mathit{Nodes}', \mathit{Ans}, \mathit{Wait}[Q' \mapsto \mathit{Wait}(Q') \cup \{\mathit{nd}\}])$$
$$\text{if} \quad \mathit{nd} \equiv \langle _; Q :: _; _; _; _; _ \rangle, \quad \exists\, Q' \in \mathit{dom}(\mathit{Ans}) \ : \ Q \Rightarrow Q', \quad \mathit{Nodes}' = \{\mathit{nd}'' \ : \ \mathit{nd}' \in \mathit{Ans}(Q'), \ \mathit{nd}'' = \mathit{resolve}(\mathit{nd}, \mathit{nd}') \text{ exists}\}$$

$$(\{\mathit{nd}\} \uplus \mathit{Nodes}, \mathit{Ans}, \mathit{Wait}) \xrightarrow{\mathit{SpawnRoot}} (\mathit{Nodes} \cup \{\langle Q \rangle\}, \mathit{Ans}[Q \mapsto \emptyset], \mathit{Wait}[Q \mapsto \{\mathit{nd}\}])$$
$$\text{if} \quad \mathit{nd} \equiv \langle _; Q :: _; _; _; _; _ \rangle, \quad \forall\, Q' \in \mathit{dom}(\mathit{Ans}) \ : \ Q \not\Rightarrow Q'$$

Lemma C.10. (answer groundness) If $(\mathit{Nodes}, \mathit{Ans}, \mathit{Wait})$ is reachable from some initial state and $\langle P; [\,]; c; S; \vec{\mathit{nd}}; \mathit{Rl} \rangle \in \mathit{Nodes}$ then S and c are ground and c is valid.

Proof. We prove the following, stronger, invariant by induction on the transition rules. If $\langle P \rangle \in \mathit{Nodes}$ then all IN-parameters in P are ground. If $\langle P; \vec{Q}; c; S; \vec{\mathit{nd}}; \mathit{Rl} \rangle \in \mathit{Nodes}$ then

all IN-parameters in S are ground, and all OUT-parameters in S are either ground or occur as OUT-variable in $\vec{Q}$. If the node has a current subgoal Q (the head of $\vec{Q}$), all IN-parameters of Q are ground.

The statement holds for any initial state because it only contains a root node with an IN/OUT-safe query. Root nodes are only produced by *SpawnRoot* transitions. By induction, all IN-parameters of the current subgoal Q are ground, hence the new root node $\langle Q \rangle$ satisfies the required property as well.

Suppose the node is produced by *ResolveClause*. The IN-parameters in its partial answer S are ground because it is resolved with P which satisfies the same property, by the inductive hypothesis. If an OUT-parameter in S is a variable, it must occur as an OUT-parameter in $\vec{Q}$, as all rules in $\mathcal{P}$ are IN/OUT-safe. If the node has a current subgoal, its IN-parameters are either already ground in the original rule, or they also occur as IN-parameters in the head of the rule, Q. But Q is resolved against P which grounds its IN-parameters by the inductive hypothesis, therefore all IN-parameters in Q are also grounded by the resolution unifier, which is also applied to the current subgoal.

In all other cases, the node is the resolvent of an existing node $\langle P; Q_0 :: \vec{Q}; _; S'; _; Rl \rangle$ with an existing answer node $\langle P'; [\ ]; _; S''; _; _ \rangle$, both of which enjoy the stated property by the inductive hypothesis. All IN-parameters of the partial answer S of the resolvent are ground because S is the product of applying the resolution unifier to S' which already has the same property. For the sake of contradiction, assume an OUT-parameter of S is neither ground nor occurs as an OUT-parameter in $\vec{Q}$. Then it must be an OUT-variable in S' which occurs as an OUT-variable in Q_0. But the resolution unifier unifies Q_0 with (a renaming of) S'', and S'' is completely ground, by the inductive hypothesis. But then the resolution unifier must also ground that variable, which contradicts the assumption. Finally, if $\vec{Q}$ is non-empty and has a head Q, all its IN-parameters must be ground: if the corresponding parameter in the rule Rl is a variable, it must be an IN-variable, therefore it must occur as an IN-variable in the head or as an OUT-variable in a preceding body literal. In the former case, the corresponding parameter in S (which originates from the head of Rl) is ground, and thus the parameter in Q is also ground. In the latter case, it the corresponding OUT-parameter in the preceding Q_0 is either ground or grounded by the resolution unifier, as established before. Either way, the parameter in Q will therefore also be ground. $\square$

Lemma C.11. (node invariant) We write $\bigcup Ans$ as short hand for $\bigcup_{P \in dom(Ans)} Ans(P)$. If $(Nodes, Ans, Wait)$ is reachable from some initial state and $\langle P; \vec{Q}; c; S; \vec{nd}; Rl \rangle \in Nodes$ with $Rl \equiv R \leftarrow \vec{R}, d$, then:

1. $S \Rightarrow P$;
2. $Rl \in \mathcal{P}$;
3. $\vec{nd} \subseteq \bigcup Ans$;

4. there is some θ such that $R\theta = S$, and $\vec{R}\theta = \vec{Q}'@\vec{Q}$ (where $\vec{Q}'$ are the answers in $\vec{nd}$), and $d\theta$ is equivalent to c.

Proof. By induction on the transition rules. The statements follow directly from the definition of the transition rules and from the definition of resolution. □

Lemma C.12. (soundness) If $(Nodes, Ans, Wait)$ is reachable from an initial state $\mathcal{S}_0$ then Ans is sound.

Proof. By induction on transition rules. The statement holds by definition for $\mathcal{S}_0$.

Now assume the state is not an initial state, and let Ans' be the answer table of the preceding state. For *PropagateAnswer*, we only have to consider the new answer $nd \equiv \langle P; [\,]; \texttt{True}; S; \vec{nd}; Rl\rangle$. By Lemma C.11, $S \Rightarrow P$; furthermore, $Rl \equiv R \leftarrow \vec{R}, d \in \mathcal{P}$, and there exists θ such that $R\theta = S$ and $d\theta = \texttt{True}$. Also, $\vec{R}\theta$ is equal to the set of answers in $\vec{nd}$ which in turn is a subset of $\bigcup Ans'$. So by the inductive hypothesis, $\vec{R}\theta \subseteq T^{\omega}_{\mathcal{P}}(\emptyset)$. Therefore, by definition of $T_{\mathcal{P}}$, $S \in T_{\mathcal{P}}(T^{\omega}_{\mathcal{P}}(\emptyset)) = T^{\omega}_{\mathcal{P}}(\emptyset)$, as required.

For the other transition rules the statement trivially holds by the inductive hypothesis. □

Lemma C.13. (table monotonicity) If $\mathcal{S} \equiv (Nodes, Ans, Wait)$ is reachable from an initial state, and $\mathcal{S}' \equiv (Nodes', Ans', Wait')$ is reachable from $\mathcal{S}$, then $dom(Ans) \subseteq dom(Ans')$, $dom(Wait) \subseteq dom(Wait')$, and $dom(Wait) \subseteq dom(Ans)$. For all $P \in dom(Ans)$: $Ans(P) \subseteq Ans'(P)$. For all $P \in dom(Wait)$: $Wait(P) \subseteq Wait'(P)$.

Proof. By induction on the transition rules. The statements follow from the observation that *PropagateAnswer* and *RecycleAnswers* only increase $Ans(P)$ and $Wait(P)$, respectively; *SpawnRoot* always increases the domains of Ans and $Wait$; and *ResolveClause* leaves $Wait$ unchanged and either increases the domain of Ans (that can only happen if in the very first transition from the initial state) or leaves Ans unchanged. □

Lemma C.14. (completeness) If $\mathcal{S}_f \equiv (Nodes, Ans, Wait)$ is a final state reachable from an initial state $\mathcal{S}_0$ then Ans is complete.

Proof. By induction on n in $T^n_{\mathcal{P}}(\emptyset)$. The statement vacuously holds for $n = 0$.For $n > 0$, consider any $S \in T^n_{\mathcal{P}}(\emptyset)$, $P \in dom(Ans)$ and $S \Rightarrow P$. If P is already in the domain of $\mathcal{S}_0$'s answer table the statement holds by definition of initial state and by monotonicity of the transition rules with respect to the answer table (Lemma C.13). So now assume that P was added to the domain as a result of a *SpawnRoot* transition.

By definition of $T_{\mathcal{P}}$, there exists a rule $Rl \equiv R \leftarrow R_1, ..., R_n, c \in \mathcal{P}$ and a substitution θ such that $R\theta = S$, $R_i\theta \in T^{n-1}_{\mathcal{P}}(\emptyset)$, and $c\theta$ is valid. By the inductive hypothesis, for all $R'_i \in dom(Ans)$ such that $R_i\theta \Rightarrow R'_i$ there is an answer node nd'_i in $Ans(R'_i)$ with answer $R_i\theta$.

Let P' be a fresh renaming of P, $\theta_0 = mgu(R, P')$, and for $i = 1..n$, let

$$\theta_i = \theta_{i-1} mgu(R_i\theta_{i-1}, R_i\theta).$$

Furthermore, let

$$nd_i \equiv \langle P; [R_{i+1}\theta_i, ..., R_n\theta_i]; c\theta_i; P'\theta_i; [nd'_1, ..., nd'_i]; Rl\rangle$$

for $i = 0..n$. Note that nd_n is an answer node with answer S, by Lemma C.10. We will now show that $nd_n \in Ans(P)$.

After P is added to the domain of the answer table in the *SpawnRoot* transition, there will eventually be a *ResolveClause* transition producing a new set of nodes that contains nd_0, because $\langle P; [R, R_1, ..., R_n]; c; R; [\,]; Rl\rangle$ and P are resolvable with resolution unifier θ_0.

Suppose for some $i = 0..n-1$ that nd_i gets produced as a node along a path leading from $\mathcal{S}_0$ to $\mathcal{S}_f$. Then there must be a later *RecycleAnswers* or a *SpawnRoot* transition where nd_i is added to the wait table for some R'_{i+1} where $R_{i+1}\theta_i \Rightarrow R'_{i+1}$. Since θ_i is more general than θ, we also have $R_{i+1}\theta \Rightarrow R'_{i+1}$, so $R_{i+1}\theta$ is the answer of some answer node in $Ans(R'_{i+1})$, by the inductive hypothesis. Therefore, this answer node is resolved with nd_i either in a *PropagateAnswer* or a *RecycleAnswers* transition, and the set of nodes produced by this transition contains nd_{i+1}.

Therefore, along all paths leading from $\mathcal{S}_0$ to $\mathcal{S}_f$ the nodes nd_0,...,nd_n are produced. Therefore, nd_n is eventually added to the answer table for P in a *PropagateAnswer* transition, and hence it is in $Ans(P)$ (by Lemma C.13), as required. □

Lemma C.15. (termination and complexity) All transition paths starting from an initial state are of finite length, and the path lengths are polynomial in the number of facts (i.e., clauses with empty body) in $\mathcal{P}$.

Proof. Let S be the set of predicate names that occur in the body of a clause in $\mathcal{P}$, and let C be the number of constants in $\mathcal{P}$ that occur as a parameter of a predicate in S. Further, let N be the number of clauses in $\mathcal{P}$, M the maximum number of distinct variables in the head of any clause in $\mathcal{P}$, and V the maximum number of distinct variables occurring in the body of any clause in $\mathcal{P}$. When a root node $\langle P\rangle$ is produced in a *SpawnRoot* transition, P is permanently added to the domain of the answer table. Due to the side conditions of *SpawnRoot*, such a node is only produced if there is no P' in the domain of the answer table for which $P \Rightarrow P'$. Moreover, the only predicate names and constants that can occur in a path are the ones that also occur in $\mathcal{P}$, of which there are only finitely many. Therefore, the number of *SpawnRoot* transitions is bounded by C^M, and thus the number of *ResolveClause* transitions is bounded by $C^M N$ which is also an upper bound on the number of nodes produced by *ResolveClause*. *PropagateAnswer* and *RecycleAnswers* both replace a node with a number of nodes whose subgoal lists are strictly shorter until an answer node is produced. From any given node, these two transition rules together produce no more than C^V new nodes. Thus the number of nodes produced by them is bounded by $C^{M+V}N$.

It follows that the length of any path is bounded by $4C^{M+V}N$. Hence all path lengths are polynomial in the number of facts in $\mathcal{P}$ as C is proportional to this number. □

Theorem C.16. All paths from an initial state $(\{\langle P\rangle\}, _, _)$ terminate at a final state. The answer table of any such final state is sound and complete, and its domain contains P.

Proof. This follows immediately from Lemmas C.15, C.12 and C.14. □

Restatement of Theorem 6.1. If AC is a safe assertion context and $\mathcal{P}$ its Datalog translation then there exists an IN/OUT assignment to predicate parameters in $\mathcal{P}$ such that $\mathcal{P}$ is IN/OUT-safe.

Proof. Literals introduced by Algorithm 5.2 are of the form e_1 $\textsf{says}_{e_2}$ e_3 *verbphrase*. We assign OUT to the parameter positions of e_1 and e_3, and IN to the position of e_2. The parameters in *verbphrase* are all OUT if e_3 *verbphrase* is a flat fact, and IN otherwise. Then by inspection and by assertion safety, all OUT variables in the head of a clause produced by the algorithm also occur in the body, and all IN variables in the body of a clause also occur in its head. □

Restatement of Theorem 6.2. (soundness, completeness, termination) Let Ans be a sound and complete answer table, $\mathcal{P}$ an IN/OUT-safe program and P an IN/OUT-safe query. Then $Answers_{\mathcal{P}}(P, Ans) = \Theta$ is defined, finite and equal to $\{\theta : P\theta \in T^{\omega}_{\mathcal{P}}(\emptyset), dom(\theta) \subseteq vars(P)\}$.

Proof. This follows directly from Theorem C.16, noting that the pseudocode in Figure 1 is a deterministic implementation of the labelled transition system. □

C.4 Evaluation of authorization queries

Restatement of Theorem 7.1. (Finiteness, soundness, and completeness of authorization query evaluation) For all safe assertion contexts AC and safe authorization queries q,

1. $AuthAns_{AC}(q)$ is defined and finite, and
2. $AC, \theta \vdash q$ iff $\theta \in AuthAns_{AC}(q)$.

Proof. By induction on the structure of q.

Case $q \equiv e$ $\textsf{says}$ *fact*

1. By authorization query safety, *fact* is flat, hence all parameters in e $\textsf{says}_\infty$ *fact* can be assigned OUT as in the proof of Theorem 6.1, hence q is an IN/OUT-safe Datalog query. Therefore, $AuthAns_{AC}(q)$ is defined and finite by Theorem 6.2.

2. Assume $AC, \theta \vdash e$ $\textsf{says}$ *fact*. This holds iff $AC, \infty \models e\theta$ $\textsf{says}$ $fact\theta$, by definition of $\vdash$. The translated program $\mathcal{P}$ is IN/OUT-safe, by Theorem 6.1, and we have already established above that the query e $\textsf{says}_\infty$ *fact* is also IN/OUT-safe. So by Theorems 5.3 and 6.2, this holds iff $(e\theta\ \textsf{says}_\infty\ fact\theta) \in Answers_{\mathcal{P}}(e\ \textsf{says}_\infty\ fact)$. Since $dom(\theta) \subseteq vars(e\ \textsf{says}_\infty\ fact)$ (by Lemma C.1) and $vars(e\theta\ \textsf{says}_\infty\ fact\theta) = \emptyset$, this holds iff the most general unifier of the two is θ, and iff $\theta \in AuthAns_{AC}(e\ \textsf{says}\ fact)$.

Case $q \equiv q_1,\ q_2$

1. If q is safe, then q_1 must also be safe, so by the induction hypothesis for finiteness, $AuthAns_{AC}(q_1)$ is defined and finite. By the induction hypothesis for soundness, $\theta_1 \in AuthAns_{AC}(q_1)$ implies $AC, \theta_1 \vdash q_1$. It follows from Lemma C.6 that $q_2\theta_1$ is safe, so by the induction hypothesis for finiteness, $AuthAns_{AC}(q_2\theta_1)$ is defined and finite, and hence $AuthAns_{AC}(q)$ is defined and finite.

2. Assume $AC, \theta \vdash q_1,\ q_2$. This holds iff $\theta = \theta_1\theta_2$ such that $AC, \theta_1 \vdash q_1$ and $AC, \theta_2 \vdash q_2\theta_1$. By the induction hypothesis, this holds iff $\theta_1 \in AuthAns_{AC}(q_1)$ and $\theta_2 \in AuthAns_{AC}(q_2\theta_1)$ and hence $\theta \in AuthAns_{AC}(q)$.

Case $q \equiv q_1$ **or** q_2 The statements follow directly from the induction hypotheses.
Case $q \equiv \textbf{not}(q_0)$

1. By definition of authorization query safety, $vars(q_0) \subseteq \emptyset$, hence $AuthAns_{AC}(q)$ is defined and finite.

2. Assume $AC, \theta \vdash \mathsf{not}(q_0)$. By definition of $\vdash$, q_0 must be ground. Lemma C.1 implies that $\theta = \epsilon$, therefore $AC, \epsilon \nvdash q_0$. In other words, there exists no σ such that $AC, \sigma \nvdash q_0$. By the induction hypothesis, this holds iff $AuthAns_{AC}(q_0) = \emptyset$ and hence $AuthAns_{AC}(\mathsf{not}(q_0)) = \{\epsilon\}$.

Case $q \equiv c$

1. By definition of authorization query safety, $vars(c) \subseteq \emptyset$, hence $AuthAns_{AC}(q)$ is defined and finite.

2. This is similar to the previous case.

□

Restatement of Theorem 7.2. Let M be the number of flat atomic assertions (i.e., those without conditional facts) in AC and N be the maximum length of constants occurring in these assertions. The time complexity of computing $AuthAns_{AC}$ is polynomial in M and N.

Proof. The number of clauses with empty body in the translated Datalog program is proportional to M. The constants stay unchanged, so the maximum length of any constant occurring in the set of those clauses is N. From Lemma C.15 and the fact that each step in the labelled transition system can be computed in time polynomial with respect to N (as the validity of a ground constraint can be checked in polynomial time), we get that $Answers_{\mathcal{P}}$ is polynomial-time computable with respect to M and N. The time complexity of $AuthAns_{AC}$ for a fixed query is clearly polynomial in the computation time for $Answers_{\mathcal{P}}$, hence it is also polynomial in M and N. □

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–22, 1998.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] M. Y. Becker. Cassandra: Flexible trust management and its application to electronic health records (Ph.D. thesis). Technical Report UCAM-CL-TR-648, University of Cambridge, Computer Laboratory, 2005. See `http://www.cl.cam.ac.uk/TechReports/UCAM-CL-TR-648.html`.

[4] M. Y. Becker and P. Sewell. Cassandra: distributed access control policies with tunable expressiveness. In *IEEE 5th International Workshop on Policies for Distributed Systems and Networks*, pages 159–168, 2004.

[5] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *IEEE Computer Security Foundations Workshop*, pages 139–154, 2004.

[6] D. E. Bell and L. J. LaPadula. Secure computer systems: Unified exposition and Multics interpretation. Technical report, The MITRE Corporation, July 1975.

[7] M. Blaze, J. Feigenbaum, and A. D. Keromytis. The role of trust management in distributed systems security. In *Secure Internet Programming*, pages 185–210, 1999.

[8] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.

[9] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about Datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[10] D. W. Chadwick and A. Otenko. The PERMIS X.509 role based privilege management infrastructure. *Future Generation Computer Systems*, 19(2):277–289, 2003.

[11] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[12] ContentGuard. *eXtensible rights Markup Language (XrML) 2.0 specification part II: core schema*, 2001. At `www.xrml.org`.

[13] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. In *CCC '97: Proceedings of the 12th Annual IEEE Conference on Computational Complexity*, page 82, Washington, DC, USA, 1997. IEEE Computer Society.

[14] J. DeTreville. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy*, pages 105–113, 2002.

[15] S. D. C. di Vimercati, P. Samarati, and S. Jajodia. Policies, models, and languages for access control. In *Databases in Networked Information Systems*, volume 3433, pages 225–237, 2005.

[16] S. W. Dietrich. Extension tables: Memo relations in logic programming. In *Symposium on Logic Programming*, pages 264–272, 1987.

[17] B. Dillaway. A unified approach to trust, delegation, and authorization in large-scale grids. Whitepaper, Microsoft Corporation, Sept. 2006.

[18] C. M. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory, RFC 2693, September 1999. See `http://www.ietf.org/rfc/rfc2693.txt`.

[19] M. Evered and S. Bögeholz. A case study in access control requirements for a health information system. In *CRPIT '04: Proceedings of the second workshop on Australasian information security, Data Mining and Web Intelligence, and Software Internationalisation*, pages 53–61, Darlinghurst, Australia, 2004. Australian Computer Society, Inc.

[20] I. T. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security*, pages 83–92, 1998.

[21] L. Giuri and P. Iglio. Role templates for content-based access control. In *Proceedings of the 2nd ACM Workshop on Role-Based Access Control (RBAC-97)*, pages 153–159, 1997.

[22] J. Y. Halpern and V. Weissman. Using first-order logic to reason about policies. In *IEEE Computer Security Foundations Workshop*, pages 187–201, 2003.

[23] J. Y. Halpern and V. Weissman. A formal foundation for XrML. In *CSFW '04: Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, page 251, Washington, DC, USA, 2004. IEEE Computer Society.

[24] P. Humenn. *The formal semantics of XACML (draft).* Syracuse University, 2003. At `lists.oasis-open.org/archives/xacml/200310/pdf00000.pdf`.

[25] J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.

[26] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, pages 106–115, 2001.

[27] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys (CSUR)*, 21(1):93–124, 1989.

[28] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, 1992.

[29] N. Li, B. Grosof, and J. Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, pages 27–42, 2000.

[30] N. Li and J. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Computer Security Foundations Workshop*, 2003.

[31] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proc. PADL*, pages 58–73, 2003 2003.

[32] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[33] E. C. Lupu and M. Sloman. Reconciling role-based management and role-based access control. In *ACM Workshop on Role-based Access Control*, pages 135–141, 1997.

[34] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, 1982.

[35] OASIS. *Security Assertion Markup Language (SAML)*. At `www.oasis-open.org/committees/security`.

[36] OASIS. *eXtensible Access Control Markup Language (XACML) Version 2.0 core specification*, 2005. At `www.oasis-open.org/committees/xacml/`.

[37] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM Press.

[38] P. Revesz. *Introduction to constraint databases*. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[39] P. Z. Revesz. Constraint databases: A survey. In *Semantics in Databases*, pages 209–246, 1995.

[40] R. L. Rivest. Can we eliminate certificate revocations lists? In *Financial Cryptography*, pages 178–183, 1998.

[41] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure, August 1996. See `http://theory.lcs.mit.edu/~rivest/sdsi10.ps`.

[42] H. Tamaki and T. Sato. OLD resolution with tabulation. In *Proceedings on Third international conference on logic programming*, pages 84–98, New York, NY, USA, 1986. Springer-Verlag New York, Inc.

[43] M. Thompson, A. Essiari, and S. Mudumbai. Certificate-based authorization policy in a PKI environment. *ACM Transactions on Information and System Security*, 6(4):566–588, 2003.

[44] D. Toman. Memoing evaluation for constraint extensions of Datalog. *Constraints*, 2(3/4):337–359, 1997.

[45] J. D. Ullman. Assigning an appropriate meaning to database logic with negation. In H. Yamada, Y. Kambayashi, and S. Ohta, editors, *Computers as Our Better Partners*, pages 216–225. World Scientific Press, 1994.