SecPAL: Design and Semantics of a Decentralized Authorization Language

Moritz Y. Becker Cédric Fournet Andrew D. Gordon

September 2006
(Revised on 7 February 2007)

Technical Report
MSR–TR–2006–120

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

SecPAL Specification Use Terms

By using or providing feedback on the SecPAL Specification (comprised of the SecPAL formal model, SecPAL Schema Specification, and SecPAL Schema) ("Specification"), you agree to the following terms and conditions:

- Microsoft hereby grants you permission to copy and review the Specification (a) as a reference to assist you in planning and designing your implementation of the Specification and (b) to provide feedback on the Specification to Microsoft. You may not modify, create derivative works from, subset, or extend the Specification.

- Provided that you comply with all the terms of use for the Specification, Microsoft agrees to grant you a royalty-free license under reasonable and non-discriminatory terms and conditions to Microsoft patents that Microsoft deems necessary to implement the Specification. You must comply with and implement **all** normative portions of the Specification in its entirety; you may not elect to implement only portions of the Specification. Unless otherwise specifically mentioned all sections of the Specification should be considered normative.

- You have no obligation to give Microsoft any suggestions, comments or other feedback ("Feedback") relating to the Specification. If you do give Microsoft Feedback on the Specification, You agree: (a) Microsoft may freely use, reproduce, license, distribute, and otherwise commercialize Your Feedback in any Microsoft product or service offering; (b) you also grant third parties, without charge, only those patent rights necessary to implement those portions of the Specification that incorporate your Feedback; and (c) you will not give Microsoft any Feedback (i) that you have reason to believe is subject to any patent, copyright or other intellectual property claim or right of any third party; or (ii) subject to license terms which seek to require any Microsoft product offering incorporating or derived from such Feedback, or other Microsoft intellectual property, to be licensed to or otherwise shared with any third party.

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, John Leen, and Jason Mackay.

SecPAL: Design and Semantics of a Decentralized Authorization Language

Moritz Y. Becker Cédric Fournet Andrew D. Gordon

September 2006
(Revised on 7 February 2007)

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 careful balance between syntactic and semantic simplicity, policy expressiveness, and execution efficiency. The syntax is close to natural language, and 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

Many applications depend on complex and changing authorization criteria. Some domains, such as electronic health records or eGovernment, require that authorization complies with evolving legislation. Distributed systems, 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 advantages, these systems are not much used. We conjecture that 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 respects. The following is an overview of the main technical contributions and features of SecPAL.

Expressiveness Our design is a careful composition of three features for expressing decentralized authorization policies: delegation, constraints, and negation.

- Flexible delegation of authority is the essence of decentralized management.

  We introduce a novel delegation primitive ("can say") that covers a wider spectrum of delegation variants than existing authorization languages, including those specifically designed for flexible delegation such as XrML [19], SPKI/SDSI [25] and Delegation Logic (DL) [37].

- Support for domain-specific constraints is also important, but existing solutions only consider a specific class of constraints [11, 10, 54] or are very restrictive to preserve decidability and tractability [39, 8], and disallow constraints required for expressing idioms commonly used in practice.

  We provide a set of mild, 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.

- Negation is useful for expressing idioms such as separation of duties, but its liberal adoption can make policies hard to understand, and its combination with recursion can cause intractability and semantic ambiguity [53].

  We introduce a syntax for authorization queries, separate from policy assertions. We permit negation within queries (even universally quantified negation), but not within assertions. This separation avoids intractability and ambiguity, and simplifies the task of authoring policies with negation.

Clear, readable syntax The syntax of some policy languages, such as XACML [44] 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. [37, 34, 21, 40, 29, 39, 8]). SecPAL has a concrete syntax consisting of simple statements close to natural language. (It also has an XML schema for exchanging statements between implementations.)

Succinct, unambiguous semantics Languages such as XACML, XrML, or SPKI/SDSI [25] 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. [31, 30, 1]).

For example, it was recently proved that the evaluation algorithms of XrML and the related MPEG REL are not guaranteed to terminate.[1] Moreover, the analysis in [38] 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.

Logic-based languages have a formal semantics and, thus, are unambiguous. In many cases, however, the semantics is specified only indirectly, by translation to another language with a formal semantics, such as Datalog [34, 21, 40], Datalog with Constraints [39, 8] or Prolog [37]. Instead, for the purpose of succinct specification, we define three deduction rules that directly specify the meaning of SecPAL assertions, independently of any other logic.

Effective decision procedures We show that SecPAL query evaluation is decidable and tractable (with polynomial data complexity) by translation into Datalog with Constraints. We describe a deterministic tabling resolution algorithm tailored to efficient evaluation of SecPAL authorization queries with constraints and negation, and present correctness and complexity theorems for the evaluation of policies that meet our syntactic safety conditions.

Extensibility SecPAL builds on the notion of tunable expressiveness introduced in [7] 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.

In combination, we believe that SecPAL achieves a good balance between syntactic and semantic simplicity, policy expressiveness, and execution efficiency for decentralized authorization. Although system implementation is not the subject of this paper, SecPAL has been implemented and deployed as the core authorization mechanism of a large system-development project, initially targeted at grid applications [24]. The system provides a PKI-based, SOAP-encoded infrastructure for exchanging policy assertions. It also includes a policy-editing tool and support for invoking authorization queries from C#. It relies on an instance of our evaluation algorithm specialized for some fixed, domain-specific verbs and constraints. The development of a formal semantics for SecPAL, in parallel with its experimental use for access control within a distributed computing environment, has led to many improvements in its design.

Contents The rest of the paper is organized as follows. Section 2 illustrates SecPAL on a simple example. Section 3 defines the syntax and semantics of SecPAL assertions. Section 4 defines SecPAL authorization queries, built as conjunctions, disjunctions and negations of facts and constraints. Section 5 shows how to express a variety of authorization policy idioms in SecPAL. Sections 6, 7, and 8 give our algorithm for evaluating authorization queries and establish its formal soundness and completeness. SecPAL assertions are first translated into constrained Datalog (Section 6); the resulting program is then evaluated using a deterministic variant of resolution with tabling (Section 7) for a series of Datalog queries obtained from the SecPAL query (Section 8). Section 9 summarizes related works 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.

[1] Personal communication, Vicky Weissman.

2 A Simple Example

To introduce the main features of SecPAL, we consider an example in the context of a simplified grid system. Access control in grids typically involves interaction between several administrative domains with individual policies and requires attribute-based authorization and delegation [56, 16, 55].

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` plus a collection of tokens for the request, expressed as three SecPAL assertions:

$$\texttt{STS}\ \textsf{says}\ \texttt{Alice}\ \textsf{is a researcher} \tag{1}$$

$$\texttt{FileServer}\ \textsf{says}\ \texttt{Alice}\ \textsf{can read}\ \texttt{file://project} \tag{2}$$

$$\texttt{Alice}\ \textsf{says}\ \texttt{Cluster}\ \textsf{can read}\ \texttt{file://project/data}\ \textsf{if}\ \textsf{currentTime}() \leq \texttt{07/09/2006} \tag{3}$$

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. (In practice, an authorization query table on the cluster maps user requests to corresponding queries.) Assume the local policy of the cluster includes the assertions:

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

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

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 well include more complex conditions.) 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` 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` against its local policy plus Alice's tokens. Assume the local policy of the server includes the assertion

$$\texttt{FileServer}\ \textsf{says}\ x\ \textsf{can say}_\infty\ y\ \textsf{can read}\ \textit{file}\ \textsf{if}\ x\ \textsf{can read}\ \textit{dir}, \textit{file} \preceq \textit{dir}, \textsf{markedConfidential}(\textit{file}) \neq \texttt{Yes} \tag{6}$$

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 *conditional fact* (that can be derived from other assertions), 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` provided that `FileServer` successfully checks the two constraints currentTime() $\leq$ `07/09/2006` and markedConfidential(`file://project/data`) $\neq$ `Yes`.

In the delegation rules (4) and (6), the "can say" assertions have different subscripts: in (4), can say$_0$ prevents `STS` from re-delegating the delegated fact; conversely, in (6), 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` may be satisfied by applying (6) twice, with $x = \texttt{Alice}$ then $x = \texttt{Cluster}$. Alternatively, `FileServer` may simply issue the assertion

`FileServer` says `Node23` can act as `Cluster` (7)

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

3 Syntax and semantics

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 An authorization policy is specified as a set of assertions (called *assertion context*) of the form

A says $fact$ if $fact_1, ..., fact_n, c$

where the facts are sentences that state properties on principals, defined below. In the assertion, A is the *issuer*; $fact_1, \ldots, fact_n$ 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 verb phrase 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. The following defines the grammar of facts.

e	::=	x	(variables)
	|	A	(constants)
pred	::=	can read [-]	(user-defined predicates with "holes")
	|	has access from [-] till [-]	
	|	...	
D	::=	0	(no re-delegation)
	|	∞	(with re-delegation)
verbphrase	::=	*pred* e_1 ... e_n	for $n = Arity(pred) \geq 0$
	|	can say$_D$ *fact*	(delegation)
	|	can act as e	(principal aliasing)
fact	::=	e *verbphrase*	

Constants represent data such as IP addresses, URLs, dates, and times. We use A, B, C as meta variables for constants, usually for denoting principals. Variables only range over the domain of constants — not predicates, facts, claims or assertions. Predicates are verb phrases (intended to express capabilities of a subject) 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 $[-]$. In the grammar above, *pred* e_1 ... e_n denotes the verb phrase obtained by inserting the arguments e_1 up to e_n into the predicate's holes. We say that a fact is *nested* when it includes a can say, and is *flat* otherwise. For example, the fact `Bob` can read f is flat, but `Charlie` can say$_0$ `Bob` can read f is nested.

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

f	$\in$	$\{+, -, \mathsf{CurrentTime}, ...\}$	(built-in functions)
e	$::=$	x	
	$\mid$	A	
	$\mid$	$f(e_1, ..., e_n)$	for $n = Arity(f) \geq 0$
pattern	$\in$	RegularExpressions	
c	$::=$	`True`	
	$\mid$	$e_1 = e_2$	
	$\mid$	$e_1 \leq e_2$	(numerical inequality)
	$\mid$	$e_1 \preceq e_2$	(path constraint)
	$\mid$	e matches *pattern*	(regular expression)
	$\mid$	$\mathsf{not}(c)$	(negation)
	$\mid$	c_1, c_2	(conjunction)

Additional constraints can be added without affecting decidability or tractability. 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`, $e_1 \neq e_2$, and c_1 or c_2. We usually omit the `True` constraint, and also omit the if in assertions with no conditional facts, writing A says *fact* for A says *fact* if `True`. We write keywords, function names and predicates in sans serif, constants in `typewriter` font, and variables in *italics*. We use a vector notation to denote a (possibly empty) list of items, e.g. writing $f(\vec{e})$ 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.

$\models$ `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 numerical 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, $[\![e_2]\!]$
$\models e$ 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 $\models c_2$

Safety Conditions The expressiveness of an authorization language depends to a large extent on the supported classes of constraints. However, adding a wide range of different constraint classes to a language is nontrivial: even if the constraint classes are tractable on their own, mixing them can result in an intractable or even undecidable language. Therefore, constraints have so far been either excluded or heavily restricted, to an extent that not even our basic constraint domain would be allowed. For example, RT^C [39] allows only a subclass of unary constraints, and Cassandra [8] allows only constraint-compact

constraint domains. [54] only consider set constraints, and in [10], only temporal periodicity constraints are considered. Furthermore, these systems require complex operations such as satisfiability checking or existential quantifier elimination that can be hard to implement, although ease of implementation is crucial for the success of a standard.

We observe that a wide range of constraints are used in authorization policies, but that their variables are always instantiated before constraint evaluation. Accordingly, rather than restricting constraints, SecPAL's safety conditions only ensure that constraints will be ground at runtime, once all conditional facts have been satisfied. Our approach facilitates high expressiveness while preserving decidability and tractability, and also simplifies the evaluation algorithm (Section 7), thus making it much easier to implement.

Definition 3.1. (Assertion safety) Let A says *fact* if $fact_1, ..., fact_n, c$ be an assertion. Then the assertion is *safe* iff the following conditions hold:

- all conditional facts are flat;
- all variables in c also occur somewhere else in the assertion;
- if *fact* is flat, all variables in *fact* also occur in a conditional fact.

For example, the assertion A says x can read `Foo` if $x \neq$ `Alice` is unsafe because x does not occur in any conditional fact, but it can be made safe by inserting e.g. the conditional fact x is a user. The assertion A says x can say$_0$ y can read z if $x \neq$ `Alice` is safe because x occurs not only in the constraint, but also in the fact.

The safety condition guarantees that the evaluation of the Datalog translation, as described in Section 7, is complete and terminates in all cases.

Semantics To be practically usable, a policy language should not only have a simple, readable syntax, but also a simple, intuitive semantics. We now describe a formal semantics consisting of only three deduction rules that directly reflect the intuition suggested by the syntax. This proof-theoretic approach enhances simplicity and clarity, far more than if we had instead taken the constrained Datalog translation in Section 6 as the language specification.

Let a substitution θ be a function mapping variables to constants and variables, and let ε be the empty substitution. 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 free variables occurring in a phrase of syntax X.

Each deduction rule consists of a set of premises and a single consequence of the form $\mathcal{AC}, D \models A$ 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 $\mathcal{AC}$. If $D = 0$, no instance of the rule (can say) occurs in the derivation.

$$\text{(cond)}\ \frac{\begin{array}{l}(A \text{ says } fact \text{ if } fact_1, ..., fact_k \text{ where } c) \in \mathcal{AC} \\ \mathcal{AC}, D \models A \text{ says } fact_i\theta \text{ for all } i \in \{1..k\} \qquad \models c\theta \qquad vars(fact\theta) = \emptyset\end{array}}{\mathcal{AC}, D \models A \text{ says } fact\theta}$$

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

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

Rule (cond) allows the deduction of matching assertions in $\mathcal{AC}$ with all free variables substituted by constants. All conditional facts must be deducible with the same delegation flag D as in the conclusion. Furthermore, the substitution must also make the constraint ground and valid.

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 in the conclusion is ∞. The matching assertion made by B must be proved with the delegation flag D read from A's can say assertion. In other words, if D is 0, then the matching assertion must be proved without any application of the (can say) rule.

Rule (can act as) asserts that all 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 3.2. If $\mathcal{AC}, D \models A$ says B can act as B' and $\mathcal{AC}, D \models A$ says B' can act as B'' then $\mathcal{AC}, D \models A$ 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 3.3. If $\mathcal{AC}, D \models A$ says *fact* then $vars(fact) = \emptyset$.

Proposition 3.4. If $\mathcal{AC}, 0 \models A$ says *fact* then $\mathcal{AC}, \infty \models A$ says *fact*.

Proposition 3.5. If $\mathcal{AC}_1, D \models A$ says *fact* then $\mathcal{AC}_1 \cup \mathcal{AC}_2, D \models A$ says *fact*.

Proposition 3.6. Let $\mathcal{AC}_A$ be the set of all assertions in $\mathcal{AC}$ whose issuer is A. $\mathcal{AC}, 0 \models A$ says *fact* iff $\mathcal{AC}_A, 0 \models A$ says *fact*.

Proposition 3.6 implies that if A says *fact* is deduced from a zero-depth delegation assertion A says B can say$_0$ *fact* then the delegation chain is guaranteed to depend only on assertions issued by B. XrML and DL [37] also support depth restrictions, but these can be defeated as their semantics do not have the stated property. Section 5 discusses this issue in more detail.

4 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:

$$
\begin{array}{rcl}
q & ::= & e \text{ says } fact \\
& | & q_1,\ q_2 \\
& | & q_1 \text{ or } q_2 \\
& | & \mathsf{not}(q) \\
& | & c \\
& | & \exists x(q)
\end{array}
$$

Negative conditions enable policies such as separation of duties, threshold and prohibition policies (see Section 5). However, coupling negation with a recursive language may cause semantic ambiguities [53], higher computational complexity, or even undecidability [46]. Our solution is based on the observation that for most policies, negated conditions can be effectively separated from recursion by allowing negation only in authorization queries. Collecting negations at the level of authorization queries also makes for clearer policies whose consequences are easier to foresee. Indeed, SecPAL authorization queries

could be further extended by even more powerful composition operators such as aggregation (as in Cassandra [7]) or threshold operators (as in RT^T [40]), without changing the assertion semantics and without affecting the complexity results.

We write θ_{-x} to denote the substitution that has domain $dom(\theta) - \{x\}$ and is equivalent to θ on this domain. The semantics of authorization queries is defined by the relation $\mathcal{AC}, \theta \vdash q$, as follows:

$\mathcal{AC}, \theta \vdash e$ says $fact$	iff $\mathcal{AC}, \infty \models e\theta$ says $fact\theta$, and $dom(\theta) \subseteq vars(e$ says $fact)$
$\mathcal{AC}, \theta_1\theta_2 \vdash q_1,\ q_2$	iff $\mathcal{AC}, \theta_1 \vdash q_1$ and $\mathcal{AC}, \theta_2 \vdash q_2\theta_1$
$\mathcal{AC}, \theta \vdash q_1$ or q_2	iff $\mathcal{AC}, \theta \vdash q_1$ or $\mathcal{AC}, \theta \vdash q_2$
$\mathcal{AC}, \varepsilon \vdash \mathsf{not}(q)$	iff $\mathcal{AC}, \varepsilon \nvdash q$ and $vars(q) = \emptyset$
$\mathcal{AC}, \theta \vdash c$	iff $\models c$
$\mathcal{AC}, \theta\|_{-x} \vdash \exists x(q)$	iff $\mathcal{AC}, \theta \vdash q$

One can easily verify that $\mathcal{AC}, \theta \vdash q$ implies $dom(\theta) \subseteq vars(q)$. (Note that $vars(\exists x(q))$ is defined as $vars(q) - \{x\}$.)

Given a query q and an authorization context $\mathcal{AC}$, an authorization algorithm should return the *answer set* of all substitutions θ such that $\mathcal{AC}, \theta \vdash q$. If the query is ground, the answer set is either empty (meaning "no") or 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. For example, the answer set for the query `Alice` says x can read `Foo` contains all assignments to x of principals who can read `Foo` according to `Alice`. This returns more information than just "yes, the query can be satisfied for some x". Section 8 gives an algorithm for finding this set of substitutions.

Safety Conditions We now give a safety condition on queries to guarantee that (1) the answer set includes a finite number of substitutions, given that the assertions in the assertion context are safe; and (2) subqueries of the form $\mathsf{not}(q)$ or c are always ground when they are evaluated, under the assumption that conjunctive queries are evaluated from left to right (see Section 8).

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 collects the variables that can be assumed to be instantiated before evaluating q, and $I \uplus O$ collects the variables that are guaranteed to be instantiated after evaluating q.

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

$$\frac{I \Vdash q_1 : O_1 \quad I \Vdash q_2 : O_2}{I \Vdash q_1 \text{ or } q_2 : O_1 \cap O_2} \qquad \frac{I \Vdash q : O \quad vars(q) \subseteq I}{I \Vdash \mathsf{not}(q) : \emptyset}$$

$$\frac{vars(c) \subseteq I}{I \Vdash c : \emptyset} \qquad \frac{I \Vdash q : O \quad x \notin I}{I \Vdash \exists x(q) : O - \{x\}}$$

Definition 4.1. (Authorization query safety) An authorization query q is *safe* iff there exists a set of variables O such that $\emptyset \Vdash q : O$.

Examples

Safe queries	**Unsafe queries**
A says C can read `Foo`	A says B can say$_0$ C can read `Foo`
x says y can read f, $x = A$	$x = A$, x says y can read f
x says A can read f, B says y can read f, $x \neq y$	x says A can read f, B says y can read f, $x \neq w$
(x says y can read f or y says x can read f), $x \neq y$	(x says y can read f or y says z can read f), $x \neq y$
x says y can read f, not(y says x can read f)	x says y can read f, not(y says z can read f)
not($\exists x$(A says x can read `Foo`))	$\exists x$(not(A says x can read `Foo`))

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

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*. This 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:

$$\begin{aligned}&\texttt{check-access-permission}(x):\\ &\quad \texttt{FileServer} \text{ says } x \text{ has access from } t_1 \text{ till } t_2,\\ &\quad t_1 \leq \text{currentTime}() \leq t_2\\ &\quad \text{not } \exists t_3, t_4(\texttt{FileServer} \text{ says } x \text{ has no access from } t_3 \text{ till } t_4,\\ &\qquad\qquad t_3 \leq \text{currentTime}() \leq t_4)\end{aligned} \tag{8}$$

When the method `check-access-permission` is called, the above authorization query (with x instantiated) is evaluated against the assertion context, and the answer is returned to the resource guard, which can then enforce the policy.

This example features a universally quantified negated statement, encoded by a negated existential quantifier. It also illustrates how a prohibition policy with a Deny-Override conflict resolution rule can be written in SecPAL. More elaborate conflict resolution rules such as assertions with different priorities can 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 [26, 22].

5 Policy idioms

In this section, we give examples of assertions and queries to show how SecPAL can express a wide range of policy idioms, in comparison with other authorization languages.

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

$$\begin{aligned}&\texttt{FileServer} \text{ says } \textit{user} \text{ can say}_\infty\ x \text{ can } \textit{access resource} \text{ if}\\ &\quad \textit{user} \text{ can } \textit{access resource}\end{aligned} \tag{9}$$

For example, if it follows from the assertion context that `FileServer` says `Alice` can read `file://docs/` and `FileServer` says `Bob` can read `file://docs/`, then `FileServer` says `Bob` can read `file://docs/`. Languages with restricted or no recursion such as XACML [44] or Lithium [29] cannot express such a policy.

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. Assertions (10) and (11) below implement the Simple Security Property and the *-Property from the Bell-LaPadula model [9], respectively.

$$\begin{aligned}&\texttt{FileServer} \text{ says } x \text{ can read } f \text{ if} \\ &\quad x \text{ is a user}, f \text{ is a file}, \text{ level}(x) \geq \text{level}(f)\end{aligned} \tag{10}$$

$$\begin{aligned}&\texttt{FileServer} \text{ says } x \text{ can write } f \text{ if} \\ &\quad x \text{ is a user}, f \text{ is a file}, \text{ level}(x) \leq \text{level}(f)\end{aligned} \tag{11}$$

Role hierarchies The can act as verb phrase can express role membership as well as role hierarchies in which roles inherit all privileges of less senior roles. The following assertions model a part of the hierarchy of medical careers in the UK National Health Service (NHS).

$$\texttt{NHS} \text{ says } \texttt{FoundationTrainee} \text{ can read } \texttt{file://docs/} \tag{12}$$

$$\texttt{NHS} \text{ says } \texttt{SpecialistTrainee} \text{ can act as } \texttt{FoundationTrainee} \tag{13}$$

$$\texttt{NHS} \text{ says } \texttt{SeniorMedPractitioner} \text{ can act as } \texttt{SpecialistTrainee} \tag{14}$$

$$\texttt{NHS} \text{ says } \texttt{Alice} \text{ can act as } \texttt{SeniorMedPractitioner} \tag{15}$$

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 NHS 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 [28, 41]. In SecPAL, parameterized roles, attributes and privileges can be encoded by introducing verb phrases, as in Assertion (16). The following example uses the verb phrases can access health record of $[-]$ and is a treating clinician of $[-]$.

$$\begin{aligned}&\texttt{NHS} \text{ says } x \text{ can access health record of } \textit{patient} \text{ if} \\ &\quad x \text{ is a treating clinician of } \textit{patient}\end{aligned} \tag{16}$$

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 can-initiate-payment(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 Bank says R has initiated P to the local assertion context. The method can-authorize-payment is called when a principal attempts to authorize a payment:

$$\begin{aligned}&\texttt{can-initiate-payment}(\textit{requester}, \textit{payment}) : \\ &\quad \texttt{Bank} \text{ says } \textit{requester} \text{ is a manager}, \\ &\quad \text{not}(\exists x(\texttt{Bank} \text{ says } x \text{ has initiated } \textit{payment}))\end{aligned} \tag{17}$$

$$\begin{aligned}&\texttt{can-authorize-payment}(\textit{requester}, \textit{payment}) : \\ &\quad \texttt{Bank} \text{ says } \textit{requester} \text{ is a manager}, \\ &\quad \texttt{Bank} \text{ says } x \text{ has initiated } \textit{payment}, \\ &\quad x \neq \textit{requester}\end{aligned} \tag{18}$$

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 [40]. 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.

$$\begin{aligned}&\texttt{Alice}\ \textsf{says}\ x\ \textsf{is trusted by}\ \texttt{Alice}\ \textsf{if}\\ &\quad x\ \textsf{is trusted by}\ a,\ x\ \textsf{is trusted by}\ b,\ x\ \textsf{is trusted by}\ c,\\ &\quad a \neq b,\ b \neq c,\ a \neq c\end{aligned} \tag{19}$$

$$\texttt{Alice}\ \textsf{says}\ x\ \textsf{can say}_\infty\ y\ \textsf{is trusted by}\ x\ \textsf{if}\ x\ \textsf{is trusted by}\ \texttt{Alice} \tag{20}$$

Hierarchical resources Suppose the assertion

$$\texttt{FileServer}\ \textsf{says}\ \texttt{Alice}\ \textsf{can read}\ \texttt{file://docs/} \tag{21}$$

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

$$\texttt{FileServer}\ \textsf{says}\ \texttt{Alice}\ \textsf{can read}\ x\ \textsf{if}\ x\ \textsf{matches}\ \texttt{file://docs/*} \tag{22}$$

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

$$\begin{aligned}&\texttt{canRead}(\mathit{requester}, \mathit{path}) :\\ &\quad \texttt{FileServer}\ \textsf{says}\ \mathit{requester}\ \textsf{can read}\ \mathit{path2},\ \mathit{path} \preceq \mathit{path2}\end{aligned} \tag{23}$$

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/`.

$$\begin{aligned}&\texttt{FileServer}\ \textsf{says}\ \mathit{user}\ \textsf{can say}_\infty\ x\ \textsf{can}\ \mathit{access}\ \mathit{path}\ \textsf{if}\\ &\quad \mathit{user}\ \textsf{can}\ \mathit{access}\ \mathit{path2},\\ &\quad \mathit{path} \preceq \mathit{path2}\end{aligned} \tag{24}$$

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 [39], 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, a shop gives a discount to students every Friday. Both this temporal periodicity requirement and the expiration date of the student attribute can be expressed by 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.

```
Shop says x is entitled to discount  if                                          (25)
    x is a student till date,
    currentTime() ≤ date, currentDay() = Friday
Shop says univ can say_∞ x is a student till date  if  univ is a university      (26)
Shop says BoardOfEducation can say_∞ univ is a university                        (27)
```

SPKI/SDSI [25], DL [37], Binder [21], RT [40] and Cassandra [8] can all express attribute-based delegation and linked local name spaces. SecPAL makes the delegation step explicit and thus allows for more fine-grained delegation control, as demonstrated in the following examples of various sorts of delegation. There are other general techniques to constrain delegation; for example, Bandmann, Firozabadi and Dam [4] propose the use of regular expressions to constrain the shape of delegation trees.

Constrained delegation Delegators may wish to restrict the parameters of the delegated fact. Such policies typically require domain-specific constraints that are not supported by previous languages for the sake of tractability. 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.

```
FileServer says STS can say_∞ x has access from t1 till t2  if  t2 − t1 ≤ 8 hours   (28)
```

The delegation depth in Assertion (28) is unlimited, so STS can in turn delegate the same right to some STS2, possibly with additional constraints. For example, with Assertion (29) issued by STS, FileServer accepts tickets issued by STS2 with a validity period of at most eight hours, where the start date is not before 01/01/2007 (but STS2 may not re-delegate).

```
STS says STS2 can say_0 x has access from t1 till t2  if  t1 ≥ 01/01/2007        (29)
```

Depth-bounded delegation The verb phrase can say$_0$ *fact* allows no further delegation of *fact*, while can say$_\infty$ *fact* allows arbitrary further delegation. This dichotomy may seem restrictive at first sight. However, SecPAL can express any fixed integer delegation depth by nesting can say$_0$. In the following example, Alice delegates the authority over is a friend facts to Bob and allows Bob to re-delegate at most one level further.

```
Alice says Bob can say_0 x is a friend                                           (30)
Alice says Bob can say_0 x can say_0 y is a friend                               (31)
```

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.

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 [19] 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 by Charlie:

```
Charlie says x is a friend if x is a friend2                                     (32)
Charlie says Doris can say_0 x is a friend2                                      (33)
```

In XrML and DL, Charlie can then re-delegate to Doris via is a friend2, thereby circumventing the depth specification. The SecPAL semantics prevents this by threading the depth restriction through the entire branch of the proof; this is a corollary of Proposition 3.6. 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.

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 verb phrase 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 (34)
  x is a delegator

`Alice` says `Bob` is a delegator (35)

`Alice` says x can say$_0$ y is a delegator if (36)
  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 SPKI/SDSI, DL or XrML.

6 Translation into constrained Datalog

We now give a translation from SecPAL assertion contexts into equivalent constrained Datalog programs. In Section 7, we then exploit Datalog's computational complexity properties (polynomial data complexity) and use translated Datalog program for query evaluation.

Our terminology for constrained Datalog is as follows. (See [15] or [3] for a detailed introduction to Datalog and [47, 46] 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 $P_0 \leftarrow P_1, \ldots, P_n, c$, consists of a *head* literal, a list of *body* literals, and a constraint. 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 standard immediate consequence operator $T_{\mathcal{P}}$.

Definition 6.1. (Consequence operator) The *immediate consequence operator* $T_{\mathcal{P}}$ is a function between sets of ground literals and is defined as:

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

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

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 $\mathsf{says\_can\_say\_infinity\_can\_say\_zero\_can\_act\_as}(A,k,x,y,B,z)$.

Algorithm 6.2. The translation of an assertion context $\mathcal{AC}$ 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 clauses as follows.

   (a) We add the Datalog clause

   $$A\ \mathsf{says}_k\ \hat{fact}_0 \leftarrow A\ \mathsf{says}_k\ fact_1,...,A\ \mathsf{says}_k\ fact_m,c$$

   where k is a fresh variable.

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

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

   where x is a fresh variable.

3. Finally, for each Datalog clause created above with head A says_k *e verbphrase* we add a clause A says_k *x verbphrase* $\leftarrow$ A says_k x can act as e, A says_k *e verbphrase* where x is a fresh variable.

Example For example, the assertion

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

is translated into

A says_k B $\mathsf{can\ say}_\infty$ y $\mathsf{can\ say}_0$ C can read z $\leftarrow A$ says_k y can read `Foo`

A says_∞ y $\mathsf{can\ say}_0$ C can read z $\leftarrow$
  x says_∞ y $\mathsf{can\ say}_0$ C can read z ,
  A says_∞ x $\mathsf{can\ say}_\infty$ y $\mathsf{can\ say}_0$ C can read z

A says_∞ C can read z $\leftarrow$
  x says_0 C can read z ,
  A says_∞ x $\mathsf{can\ say}_0$ C can read z

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

A says$_k$ x can say$_\infty$ y can say$_0$ C can read z $\leftarrow$
    A says$_k$ x can act as B,
    A says$_k$ B can say$_\infty$ y can say$_0$ C can read z

A says$_\infty$ x can say$_0$ C can read z $\leftarrow$
    A says$_k$ x can act as y,
    A says$_\infty$ y can say$_0$ C can read z

A says$_\infty$ x can read z $\leftarrow$
    A says$_k$ x can act as C,
    A says$_\infty$ C can read z

Intuitively, the says subscript keeps track of the delegation depth, just like the D in the three semantic rules in Section 3. This correspondence is reflected in the following theorem that relates the Datalog translation to the SecPAL semantics.

Theorem 6.3. (Soundness and completeness) Let $\mathcal{P}$ be the Datalog translation of the assertion context $\mathcal{AC}$. We have A says$_D$ *fact* $\in T_{\mathcal{P}}^{\omega}(\emptyset)$ iff $\mathcal{AC}, D \models A$ says *fact*.

7 Evaluation of authorization queries

This section describes an algorithm for evaluating authorization queries (Section 4) against a SecPAL assertion context.

The first step is to evaluate atomic Datalog queries of the form e says$_\infty$ *fact* (i.e., computing all query instances that are in $T_{\mathcal{P}}^{\omega}(\emptyset)$) against the Datalog program $\mathcal{P}$ obtained by translation. The usual bottom-up approach [3], where the fixed-point model is precomputed for all queries, is not suitable, as the assertion context may be completely different between different requests. Furthermore, top-down resolution algorithms are usually more efficient in computing fully or partially instantiated goals. However, standard SLD resolution (as used in e.g. Prolog) may run into loops even for simple assertion contexts. Tabling, or memoing, is an efficient approach for guaranteeing termination by incorporating some bottom-up techniques into a top-down resolution strategy [50, 23, 18]. Tabling has also been applied to Datalog with Constraints, but requires complex constraint solving procedures [52].

Our tabling algorithm is a simplified and deterministic version that is tailored to the clauses produced by the translation of a safe assertion context (as described in Section 6). It does not require constraint solving and is thus simpler to implement. A *node* is either a *root node* of the form $\langle P \rangle$, where the *index* P is a literal, or a sextuple $\langle P; \vec{Q}; c; S; \vec{nd}; Cl \rangle$, where $\vec{Q}$ is a list of literals (the *subgoals*), c a constraint, S a literal (the *partial answer*), $\vec{nd}$ a list of sextuple nodes (the *child nodes*), and Cl a clause. The algorithm makes use of two tables. The *answer table* Ans maps literals to sets of answer nodes (i.e., nodes where $\vec{Q}$ is empty and $c = \texttt{True}$). The set $Ans(P)$ is used to store all the found answer nodes pertaining to a query $\langle P \rangle$. The *wait table* $Wait$ maps literals to sets of nodes with nonempty lists of subgoals. $Wait(P)$ is a list of all those nodes whose *current subgoal* (i.e., the left-most 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. The function *simplify* is 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

```
RESOLVE-CLAUSE(⟨P⟩)
  Ans(P) := ∅;
  foreach (Q ← Q⃗, c) ∈ 𝒫 do
      if nd = resolve(⟨P; Q :: Q⃗; c; Q; [ ]; Cl⟩, 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 Q0 :: _ in
      if there exists Q′0 ∈ dom(Ans)
            such that Q0 ⇒ 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(Q0) := {nd};
        RESOLVE-CLAUSE(⟨Q0⟩)
```

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

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 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 [35] for an overview) but there are more intricate algorithms that run in linear time, see e.g. [45, 42]. Let P be an *instance* of Q iff $P = Q\theta$ for some substitution θ, in which case we write $P \Rightarrow Q$.

A node $nd \equiv \langle P; Q :: \vec{Q}; c; S; \vec{nd}; Cl\rangle$ and a literal Q' are *resolvable* iff some Q'' is a fresh variable renaming of Q', $\theta \equiv mgu(Q, Q'')$ exists and $d \equiv simplify(c\theta) \neq \texttt{False}$. Their resolvent is $nd'' \equiv \langle P; \vec{Q}\theta; d; S\theta; \vec{nd}; Cl\rangle$, and θ is their *resolution unifier*. We write $resolve(nd, Q') = nd''$ if nd and Q' are resolvable. By extension, a node $nd \equiv \langle P; Q :: \vec{Q}; c; S; \vec{nd}; Cl\rangle$ 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 $nd'' \equiv \langle P; \vec{Q}\theta; d; S\theta; \vec{nd}@[nd']; Cl\rangle$. 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. Then $Answers_{\mathcal{P}}(P, Ans)$ 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 execution of 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 7.2 states the termination, soundness and completeness properties of $Answers_{\mathcal{P}}$. These properties will be exploited in Section 8 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.

Lemma 7.1. If $\mathcal{AC}$ 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

$$\text{for all } P \in dom(Ans): \ \langle P'; [\,]; \texttt{True}; S; \_; \_\rangle \in Ans(P) \text{ implies } P = P', S \Rightarrow P, \text{ and } S \in T_{\mathcal{P}}^{\omega}(\emptyset).$$

Ans is *complete* (with respect to $\mathcal{P}$) iff for all $P \in dom(Ans)$: $S \in T_{\mathcal{P}}^{\omega}(\emptyset)$ 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 7.2. (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 $Answers_{\mathcal{P}}(P, Ans)$ is defined, finite and equal to $\{\theta \ : \ P\theta \in T_{\mathcal{P}}^{\omega}(\emptyset), dom(\theta) \subseteq vars(P)\}$.

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 $\langle \_; \_; \_; S; \vec{nd}; Cl\rangle$ is interpreted to have edges pointing to each of its child nodes $\vec{nd}$ and an edge pointing to the clause Cl. Appendix B shows how this Datalog proof graph can be converted back into a corresponding SecPAL proof graph.

8 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 4.

Let $\mathcal{AC}$ be an assertion context and $\mathcal{P}$ its Datalog translation. We define the function $AuthAns_{\mathcal{AC}}$ on authorization queries as follows.

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

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

Theorem 8.1. (Finiteness, soundness, and completeness of query evaluation) For all safe assertion contexts $\mathcal{AC}$ and safe authorization queries q,

1. $AuthAns_{\mathcal{AC}}(q)$ is defined and finite, and

2. $\mathcal{AC}, \theta \vdash q$ iff $\theta \in AuthAns_{\mathcal{AC}}(q)$.

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 with respect to its domain (it will just have a larger domain), an 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. Data complexity [3, 20] is a measure of the computation time for evaluating a fixed query with fixed intensional database (IDB) but variable extensional database (EDB). This measure is most often used for policy languages, as the size of the EDB (the number of "plain facts") typically exceeds the size of the IDB (the number of "rules") by several orders of magnitude.

Theorem 8.2. Let M be the number of flat atomic assertions (i.e., those without conditional facts) in $\mathcal{AC}$ and let N be the maximum length of constants occurring in these assertions. The time complexity of computing $AuthAns_{\mathcal{AC}}$ is polynomial in M and N.

9 Discussion

Related work The ABLP logic [2, 36] introduced the "says" modality and the use of logic rules for expressing decentralized authorization policies. The ABLP logic includes a "speaks for" relation, governing when the right to assert a statement is delegated from one principal to another. SecPAL's delegation operators "can say_∞" and "can act as" are related to but not the same as forms of "speaks for". Initial investigations suggest there is a model of a fragment of SecPAL, without "can say_0" or constraints, within a mild extension of ABLP, but we leave a precise study as future work.

PolicyMaker and Keynote [14, 13] introduced the notion of decentralized trust management. Quite a few other authorization languages have been developed since. SPKI/SDSI [25] 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 [49], or as k-out-of-n threshold subjects. Grants can have validity restrictions and indicate whether they may be delegated.

XrML [19] (and its offspring, MPEG 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 [44] 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 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 [43] 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 [27] 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 [51], Permis [17], 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 [31] 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 a formal semantics.

Policy languages such as Binder [21], SD3 [34], Delegation Logic (DL) [37] and the RT family of languages [40] use Datalog as basis for both syntax and semantics. To support attribute-based delegation, these languages allow predicates to be qualified by an issuing principal. Cassandra [8, 7] and RT^C [39] are based on Datalog with Constraints [32, 46] for higher expressiveness. The Cassandra framework also defines a transition system for evolving policies and supports automated credential retrieval and automated trust negotiation.

Much research has been done on logic-based access control languages for single administrative domains that do not require decentralized delegation of authority. Many of these are also based on Datalog or Datalog with Constraints, e.g. [10, 12, 33, 5, 54]. Lithium [29] is a language for reasoning about digital rights and is based on a different fragment of first order logic. It is the only language allowing real logical negation in the conclusion as well as in the premises of policy rules. This is useful for analysing merged policies, but Lithium restricts recursion and cannot easily express delegation.

Conclusions We have designed an authorization language that supports fine-grained delegation control for decentralized systems, highly expressive constraints and negative conditions that are needed in practice but cannot be expressed in other languages. Combining all these features in a single language without sacrificing decidability and tractability is nontrivial. If authorization queries are extended by an aggregation operator (which can be easily done without modifying the assertion semantics and without sacrificing polynomial data complexity), SecPAL can (safely) express the entire benchmark policy in [6], one of the largest and most complex examples of a formal authorization policy to date. Despite its expressiveness, we argue that SecPAL is relatively simple and intuitive, due to the resemblance of its syntax to natural language, its small semantic specification and its purely syntactic safety conditions.

A prototype of SecPAL, including an auditing infrastructure and editing tools, has been implemented as part of a project investigating access control solutions for large-scale Grid computing environments [24]. 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. Future work includes tools for policy authoring, deployment, and formal analysis.

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 [24] presents the design goals and introduces the language informally. Gregory Fee and Jason Mackay implemented the SecPAL prototype. Martín Abadi, Blair Dillaway, Peter Sewell, and Vicky Weissman suggested improvements to a draft of this paper. We also thank Tuomas Aura, Michael Roe, and Sebastian Nanz for valuable discussions.

A Assertion expiration and revocation

This appendix describes how expiration and revocation can be expressed in SecPAL. Expiration dates can be specified as ordinary verb phrase parameters:

`UCambridge` says `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 [48]. In this case, the assertion could just contain the date without enforcing it:

`UCambridge` says `Alice` is a student till `31/12/2007`

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

`Admin` says x is entitled to discount if
x is a student till *date*,
currentTime() $\leq$ *date*,
date $-$ currentTime() $\leq$ 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. Historically, revocation is most commonly associated with X.509-based PKIs, which only support revocation of issued certificates. SecPAL includes a simple mechanism for finer-grained revocation, at the level of an individual assertion, as explained below.

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 $\mathcal{AC}$ and a set of revocation assertions $\mathcal{AC}_{rev}$ where $\mathcal{AC} \cap \mathcal{AC}_{rev} = \emptyset$, we remove all assertions revoked by $\mathcal{AC}_{rev}$ in $\mathcal{AC}$ before evaluating authorization queries. The filtered assertion context is defined by

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

The condition that $\mathcal{AC}$ and $\mathcal{AC}_{rev}$ 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.

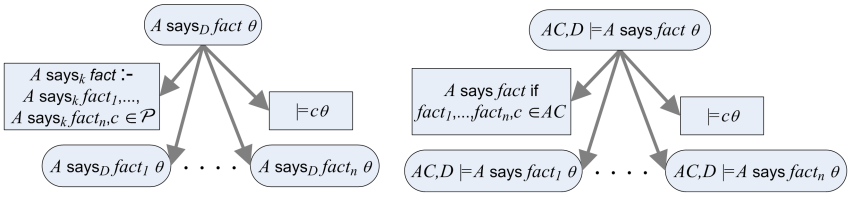

Figure 2: Left: Datalog proof node with parent from Translation Step 1 or 2a. Right: corresponding SecPAL proof node using Rule (cond).

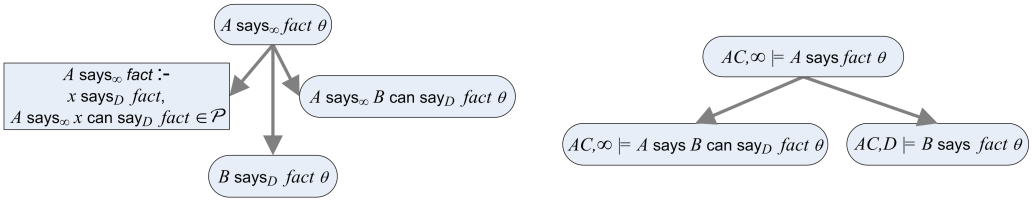

Figure 3: Left: Datalog proof node with parent from Translation Step 2b. Right: corresponding SecPAL proof node using Rule (can say).

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 3.

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 $T_{\mathcal{P}}^{\omega}(\emptyset)$ 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 $\mathcal{AC}$. 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 $\mathcal{AC}, \infty \models A$ says *fact* according to the three deduction rules of Section 3 if and only if there is a SecPAL proof graph with $\mathcal{AC}, \infty \models A$ says *fact* as a root node.

If during execution of Algorithm 6.2, 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.

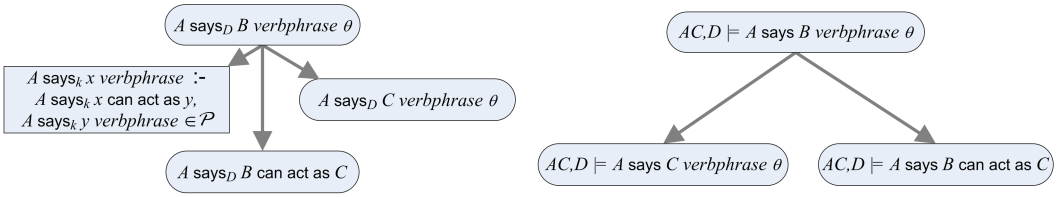

Figure 4: Left: Datalog proof node with parent from Translation Step 3. Right: corresponding SecPAL proof node using Rule (can act as).

C Auxiliary definitions and proofs

This appendix contains proofs of all theorems stated in the main part of the paper as well as supporting lemmas and definitions.

C.1 Authorization queries

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

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

Lemma C.2. If $\mathcal{AC}, \theta \vdash e$ 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 $\mathcal{AC}, \theta \vdash q$ 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 $\mathcal{AC}, \theta_1 \vdash q_1$ 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

Definition C.7. (Consequence operator) The *immediate consequence operator* $T_{\mathcal{P}}$ is a function between sets of ground literals and is defined as:

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

Lemma C.8. (Soundness) Let $\mathcal{P}$ be the Datalog translation of the assertion context $\mathcal{AC}$. If A says$_D$ *fact* $\in T_{\mathcal{P}}^{\omega}(\emptyset)$ then $\mathcal{AC}, D \models A$ 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 clause produced by Step 1 or 2a, then by the inductive hypothesis, $\mathcal{AC}, D \models A$ says $fact_i\theta$ for $i = 1...n$. Furthermore, $c\theta$ is ground and valid, so by Rule (cond), $\mathcal{AC}, D \models A$ says *fact*.

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

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

Lemma C.9. If $\mathcal{AC}, D \models A$ says B *verbphrase* then there exists an assertion in $\mathcal{AC}$ of the form

A says e_1 can say$_{D_1}$ ... e_n can say$_{D_n}$ e *verbphrase*$'$ 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 $\mathcal{AC}, D \models A$ says B *verbphrase* was (cond), there exists an assertion in $\mathcal{AC}$ of the form

A says e *verbphrase*$'$ if ...

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

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

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

Lemma C.10. (Completeness) Let $\mathcal{P}$ be the Datalog translation of the assertion context $\mathcal{AC}$. If $\mathcal{AC}, D \models A$ says *fact* then A says$_D$ *fact* $\in T_{\mathcal{P}}^{\omega}(\emptyset)$.

Proof. Assume $\mathcal{AC}, D \models A$ says *fact*. We 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 $fact_1, \ldots, fact_k, c) \in \mathcal{AC}$ is translated in Step 1 or 2a. Also, $\mathcal{AC}, D \models A$ 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. $\mathcal{AC}, K \models B$ says $fact$, and
3. $\mathcal{AC}, \infty \models A$ says B can say$_K$ $fact$.

By Lemma C.9, there is an assertion in $\mathcal{AC}$ of the form

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

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

$$\begin{array}{l} A \text{ says}_\infty fact \leftarrow \\ \quad x \text{ says}_K fact', \\ \quad A \text{ says}_\infty x \text{ can say}_K fact' \end{array}$$

where x is a fresh variable not occurring anywhere else in the clause, 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 $\mathcal{AC}, D \models A$ says C *verbphrase*, and $\mathcal{AC}, D \models A$ 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 clause in $\mathcal{P}$ of the form

$$A \text{ says}_k \ e \ verbphrase' \leftarrow ...$$

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

$$\begin{array}{l} A \text{ says}_k \ y \ verbphrase' \leftarrow \\ \quad A \text{ says}_k \ y \text{ can act as } e \\ \quad A \text{ says}_k \ e \ verbphrase' \end{array}$$

where y is a fresh variable not occurring anywhere else in the clause, 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 6.3. (Soundness and completeness) Let $\mathcal{P}$ be the Datalog translation of the assertion context $\mathcal{AC}$. We have A says$_D$ $fact \in T_{\mathcal{P}}^{\omega}(\emptyset)$ iff $\mathcal{AC}, D \models A$ says $fact$.

Proof. Follows from Lemmas C.8 and C.10. □

C.3 Datalog evaluation with tabling

The tabling evaluation algorithm in Section 7 can also be described as a non-deterministic labelled transition system. We present this system here 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 straightforward implementation of the former.

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

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

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

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

Definition C.11. 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.

Restatement of Theorem 7.1. If $\mathcal{AC}$ 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 6.2 are of the form e_1 says_{e_2} *fact*. We assign OUT to the parameter position of e_1, and IN to the position of e_2. The parameter positions in *fact* are all OUT if it is flat, 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. □

A *state* is a triple $(Nodes, Ans, Wait)$ 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 $Ans[P \mapsto A]$ is a function that maps literals Q to $Ans(Q)$ if $Q \in dom(Ans)$ and $Q \neq P$ and additionally maps P to A.

A state $(Nodes, Ans, Wait)$ is an *initial state* iff $Nodes = \{\langle P\rangle\}$ 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 $\mathcal{S} \xrightarrow{\ell} \mathcal{S}'$.

Lemma C.12. (answer groundness) If $(Nodes, Ans, Wait)$ is reachable from some initial state and $\langle P; [\,]; c; S; \vec{nd}; Cl\rangle \in Nodes$ then S and c are ground and c is valid.

Proof. We prove the following, stronger, invariant by induction on the transition rules. If $\langle P\rangle \in Nodes$ then all IN-parameters in P are ground. If $\langle P; \vec{Q}; c; S; \vec{nd}; Cl\rangle \in Nodes$ 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 clauses in $\mathcal{P}$ are IN/OUT-safe. If the node has a current subgoal, its IN-parameters are either already ground in the original clause, or they also occur as IN-parameters in the head of the clause, 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 $\langle P; Q_0 :: \vec{Q}; \_; S'; \_; Cl \rangle$ 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 clause Cl 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 Cl) 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. □

Lemma C.13. (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 $\langle P; \vec{Q}; c; S; \vec{nd}; Cl \rangle \in Nodes$ with $Cl \equiv R \leftarrow \vec{R}, d$, then:

1. $S \Rightarrow P$;
2. $Cl \in \mathcal{P}$;
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.14. (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 $nd \equiv \langle P; [\,]; \texttt{True}; S; \vec{nd}; Cl \rangle$. By Lemma C.13, $S \Rightarrow P$; furthermore, $Cl \equiv R \leftarrow \vec{R}, d \in \mathcal{P}$, 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, $\vec{R}\theta \subseteq T_{\mathcal{P}}^{\omega}(\emptyset)$. Therefore, by definition of $T_{\mathcal{P}}$, $S \in T_{\mathcal{P}}(T_{\mathcal{P}}^{\omega}(\emptyset)) = T_{\mathcal{P}}^{\omega}(\emptyset)$, as required.

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

Lemma C.15. (table monotonicity) If $\mathcal{S} \equiv (\mathit{Nodes}, \mathit{Ans}, \mathit{Wait})$ is reachable from an initial state, and $\mathcal{S}' \equiv (\mathit{Nodes}', \mathit{Ans}', \mathit{Wait}')$ is reachable from $\mathcal{S}$, then $\mathit{dom}(\mathit{Ans}) \subseteq \mathit{dom}(\mathit{Ans}')$, $\mathit{dom}(\mathit{Wait}) \subseteq \mathit{dom}(\mathit{Wait}')$, and $\mathit{dom}(\mathit{Wait}) \subseteq \mathit{dom}(\mathit{Ans})$. For all $P \in \mathit{dom}(\mathit{Ans})$: $\mathit{Ans}(P) \subseteq \mathit{Ans}'(P)$. For all $P \in \mathit{dom}(\mathit{Wait})$: $\mathit{Wait}(P) \subseteq \mathit{Wait}'(P)$.

Proof. By induction on the transition rules. The statements follow from the observation that *PropagateAnswer* and *RecycleAnswers* only increase $\mathit{Ans}(P)$ and $\mathit{Wait}(P)$, 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.16. (completeness) If $\mathcal{S}_f \equiv (\mathit{Nodes}, \mathit{Ans}, \mathit{Wait})$ 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 $S \in T^n_{\mathcal{P}}(\emptyset)$, $P \in \mathit{dom}(\mathit{Ans})$ 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.15). 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 clause $Cl \equiv R \leftarrow R_1, ..., R_n, c \in \mathcal{P}$ 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 $R'_i \in \mathit{dom}(\mathit{Ans})$ such that $R_i\theta \Rightarrow R'_i$ there is an answer node nd'_i in $\mathit{Ans}(R'_i)$ 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

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

for $i = 0..n$. Note that nd_n is an answer node with answer S, by Lemma C.12. We will now show that $nd_n \in \mathit{Ans}(P)$.

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 $\langle P; [R, R_1, ..., R_n]; c; R; [\,]; Cl \rangle$ 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 $\mathit{Ans}(R'_{i+1})$, 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 $\mathit{Ans}(P)$ (by Lemma C.15), as required. □

Lemma C.17. (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.18. 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.17, C.14 and C.16. □

Restatement of Theorem 7.2. (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 $Answers_{\mathcal{P}}(P, Ans)$ is defined, finite and equal to $\{\theta : P\theta \in T_{\mathcal{P}}^{\omega}(\emptyset), dom(\theta) \subseteq vars(P)\}$.

Proof. This follows directly from Theorem C.18, 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 8.1. (Finiteness, soundness, and completeness of query evaluation) For all safe assertion contexts $\mathcal{AC}$ and safe authorization queries q,

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

Proof. By induction on the structure of q.

Case $q \equiv e$ says $fact$

1. By authorization query safety, *fact* is flat, hence all parameters in e says_∞ *fact* can be assigned OUT as in the proof of Lemma 7.1, hence q is an IN/OUT-safe Datalog query. Therefore, $AuthAns_{\mathcal{AC}}(q)$ is defined and finite by Theorem 7.2.
2. Assume $\mathcal{AC}, \theta \vdash e$ says $fact$. This holds iff $\mathcal{AC}, \infty \models e\theta$ says $fact\theta$, by definition of $\vdash$. The translated program $\mathcal{P}$ is IN/OUT-safe, by Lemma 7.1, and we have already established above that the query e says_∞ *fact* is also IN/OUT-safe. So by Theorems 6.3 and 7.2, this holds iff $(e\theta$ says_∞ $fact\theta) \in Answers_{\mathcal{P}}(e$ says_∞ $fact)$. Since $dom(\theta) \subseteq vars(e$ says_∞ $fact)$ (by Lemma C.1) and $vars(e\theta$ says_∞ $fact\theta) = \emptyset$, this holds iff the most general unifier of the two is θ, and iff $\theta \in AuthAns_{\mathcal{AC}}(e$ 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, $\mathit{AuthAns}_{\mathcal{AC}}(q_1)$ is defined and finite. By the induction hypothesis for soundness, $\theta_1 \in \mathit{AuthAns}_{\mathcal{AC}}(q_1)$ implies $\mathcal{AC}, \theta_1 \vdash q_1$. It follows from Lemma C.6 that $q_2\theta_1$ is safe, so by the induction hypothesis for finiteness, $\mathit{AuthAns}_{\mathcal{AC}}(q_2\theta_1)$ is defined and finite, and hence $\mathit{AuthAns}_{\mathcal{AC}}(q)$ is defined and finite.

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

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

1. By definition of authorization query safety, q_0 must also be safe, hence $\mathit{AuthAns}_{\mathcal{AC}}(q)$ is defined and finite.

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

Case $q \equiv c$

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

2. This is similar to the previous case.

Case $q \equiv \exists x(q_0)$

1. If q is safe, then q_0 is also safe, by definition of authorization query safety. By the inductive hypothesis, $\mathit{AuthAns}_{\mathcal{AC}}(q_0)$ is defined and finite, hence $\mathit{AuthAns}_{\mathcal{AC}}(q)$ is also defined and finite.

2. This is straightforward by the inductive hypothesis and the definition of $\vdash$.

□

Restatement of Theorem 8.2. Let M be the number of flat atomic assertions (i.e., those without conditional facts) in $\mathcal{AC}$ and let N be the maximum length of constants occurring in these assertions. The time complexity of computing $\mathit{AuthAns}_{\mathcal{AC}}$ 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.17 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 $\mathit{Answers}_{\mathcal{P}}$ is polynomial-time computable with respect to M and N. The time complexity of $\mathit{AuthAns}_{\mathcal{AC}}$ for a fixed query is clearly polynomial in the computation time for $\mathit{Answers}_{\mathcal{P}}$, 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] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, 1993.

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

[4] O. Bandmann, B. S. Firozabadi, and M. Dam. Constrained delegation. In *IEEE Symposium on Security and Privacy*, pages 131–140, 2002.

[5] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Trans. Inf. Syst. Secur.*, 6(4):501–546, 2003.

[6] 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`.

[7] 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.

[8] 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.

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

[10] E. Bertino, C. Bettini, E. Ferrari, and P. Samarati. An access control model supporting periodicity constraints and temporal reasoning. *ACM Trans. Database Syst.*, 23(3), 1998.

[11] E. Bertino, C. Bettini, and P. Samarati. A temporal authorization model. In *CCS '94: Proceedings of the 2nd ACM Conference on Computer and communications security*, pages 126–135, New York, NY, USA, 1994. ACM Press.

[12] E. Bertino, B. Catania, E. Ferrari, and P. Perlasca. A logical framework for reasoning about access control models. In *SACMAT '01: Proceedings of the sixth ACM symposium on Access control models and technologies*, pages 41–52, New York, NY, USA, 2001. ACM Press.

[13] 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.

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

[15] 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.

[16] D. Chadwick. Authorisation in Grid Computing. *Information Security Technical Report*, 10(1):33–40, 2005.

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

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

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

[20] 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.

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

[22] 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.

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

[24] B. Dillaway. A unified approach to trust, delegation, and authorization in large-scale grids. Whitepaper, Microsoft Corporation. See `http://research.microsoft.com/projects/SecPAL/`, Sept. 2006.

[25] 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`.

[26] 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, 2004.

[27] 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.

[28] 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.

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

[30] 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.

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

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

[33] S. Jajodia, P. Samarati, M. L. Sapino, and V. S. Subrahmanian. Flexible support for multiple access control policies. *ACM Trans. Database Syst.*, 26(2):214–260, 2001.

[34] 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.

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

[36] 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.

[37] 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.

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

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

[40] 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.

[41] 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.

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

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

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

[45] 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.

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

[47] P. Z. Revesz. Constraint databases: A survey. In *Semantics in Databases*, volume 1358 of *Lecture Notes in Computer Science*, pages 209–246. Springer, 1995.

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

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

[50] 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.

[51] 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.

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

[53] 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.

[54] L. Wang, D. Wijesekera, and S. Jajodia. A logic-based framework for attribute based access control. In *FMSE '04: Proceedings of the 2004 ACM workshop on Formal methods in security engineering*, pages 45–55, 2004.

[55] V. Welch, I. Foster, T. Scavo, F. Siebenlist, and C. Catlett. Scaling TeraGrid access: A roadmap for attribute-based authorization for a large cyberinfrastructure (draft August 24). 2006. `http://gridshib.globus.org/docs/tg-paper/TG-Attribute-Authz-Roadmap-draft-aug24.pdf`.

[56] V. Welch, F. Siebenlist, I. Foster, J. Bresnahan, K. Czajkowski, J. Gawor, C. Kesselman, S. Meder, L. Pearlman, and S. Tuecke. Security for grid services. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC'03)*, page 48, 2003.