Design and Semantics of a Decentralized Authorization Language

Moritz Y. Becker Cédric Fournet Andrew D. Gordon
Microsoft Research, Cambridge, CB3 0FB, United Kingdom
{moritzb, fournet, adg}@microsoft.com

Abstract

We present a declarative authorization language that 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.

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 preestablished 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 employ a 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 [20], SPKI/SDSI [26] and Delegation Logic (DL) [36]. The semantics of “can say” is close to the “controls” operator in the original logical treatment of authorization, the ABLP logic [2].

- Support for domain-specific constraints is also important, but existing solutions only consider a specific class of constraints (e.g. temporal constraints [13], periodicity [12], set constraints [48]) or are very restrictive to preserve decidability and tractability (e.g. unary constraints [38], constraint-compact domains [10]), 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 parameterized 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 [47].

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 [41] 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. [36, 34, 22, 39, 29, 38, 10]). 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 [26] 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 [37] 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, 22, 39], Datalog with Constraints [38, 10] or Prolog [36]. 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 Cassandra [9] 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 [25]. 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 and 7 give our algorithm for evaluating authorization queries and establish its formal soundness and completeness. SecPAL assertions are first translated into Datalog with Constraints (Section 6); the resulting program is then evaluated using a deterministic variant of resolution with tabling for a series of Datalog queries obtained from the SecPAL query (Section 7). Section 8 summarizes related works and concludes. A technical report [7] contains full proofs and discusses SecPAL's mechanism for fine-grained revocation of assertions.

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 [50, 18, 49].

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.

[1] Personal communication, Vicky Weissman.

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 /project/data plus a collection of tokens for the request, expressed as three SecPAL assertions:

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

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

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

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 /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 /project/data against its local policy plus Alice's tokens. Assume the local policy of the server includes the assertion

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

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 /project/data provided that FileServer successfully checks the two constraints $\textsf{currentTime}() \leq \texttt{07/09/2006}$ and $\textsf{markedConfidential}(\texttt{/project/data}) \neq \texttt{Yes}$.

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

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

This means roughly that every fact concerning Cluster also applies to Node23. Every fact in SecPAL takes the form *e verbphrase*, where *e* is the *subject* of the fact, and *verbphrase* is the remainder. Assertion (7) means that for any *verbphrase*, 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 as it does not affect the semantics of the language.)

Assertions An authorization policy is specified as a set $\mathcal{AC}$ of assertions (called *assertion context*) of the form

$$A\ \textsf{says}\ \mathit{fact}\ \textsf{if}\ \mathit{fact}_1, ..., \mathit{fact}_n, c$$

where the facts are sentences that state properties on principals, defined below. In the assertion, A is the *issuer*; fact_1, ..., 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.

$$\begin{array}{rcll} e & ::= & x & \text{(variables)} \\ & | & A & \text{(constants)} \\ pred & ::= & \textsf{can read [-]} & \text{(predicates)} \\ & | & \textsf{has access from [-] till [-]} & \\ & | & \ldots & \\ D & ::= & 0 & \text{(no re-delegation)} \\ & | & \infty & \text{(with re-delegation)} \\ verbphrase & ::= & pred\ e_1\ ...\ e_n & \text{for } n = Arity(pred) \\ & | & \textsf{can say}_D\ fact & \text{(delegation)} \\ & | & \textsf{can act as}\ e & \text{(principal aliasing)} \\ fact & ::= & e\ verbphrase & \end{array}$$

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 user-defined, application-specific 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 Expressions r occurring in constraints range over variables, constants and applications $f(r_1,...,r_n)$ of built-in functions such as CurrentTime$()$. Constraints range over any constraint domain that includes the following basic constraints: the trivial constraint (`True`), equality ($r_1 = r_2$), numerical inequalities ($r_1 \leq r_2$; for expressing e.g. temporal constraints), path constraints ($r_1 \preceq r_2$; for hierarchical resources), and regular expressions (r matches *regExp*) for ad hoc string filtering. Constraint domains are closed under variable renaming, conjunction (c_1, c_2), and negation (not(c)).

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 write $\models c$ iff the constraint c is ground and valid. Validity is defined with respect to an implicit current state for evaluating functions such as CurrentTime$()$. We omit the standard definition of validity for basic constraints.

We use a sugared notation for constraints that can be derived from the basic ones, e.g. `False`, $r_1 \neq r_2$, 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 $f(\vec{r})$ for $f(r_1,...,r_n)$.

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 [38] allows only a subclass of unary constraints, and Cassandra [10] allows only constraint-compact constraint domains. [48] only consider set constraints, and in [12], 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 (Definition 3.1) only ensure that constraints will be ground at runtime, once all conditional facts have been satisfied. This 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**)**.** The assertion A says *fact* if $fact_1, ..., fact_n, c$ 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.

Note the similarity to the safety condition in Datalog where all variables in the head literal must also occur in a conditional literal [17]. SecPAL's safety conditions are less restrictive, as variables in "can say" assertions need not occur in any conditional fact.

At first sight, the safety conditions seem to rule out blanket permissions such as `FileServer` says x can read `Foo` (everybody can read `Foo`). However, this is not a problem in practice, because it is possible to make the assertion safe by adding a conditional fact qualifying x, for example "if x is a user". The list of users could either be stored locally, or the server could delegate to a trusted third party, e.g. by `FileServer` says `TrustedDirectory` can say$_0$ x is a user. Alternatively, the server may accept self-issued statements: `FileServer` says x can say$_0$ x is a user.

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 translation to Datalog with Constraints 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 the assertion context $\mathcal{AC}$. If $D = 0$, no instance of the rule (can say) occurs in the derivation.

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

$$\text{(can say)}\ \frac{\begin{array}{c}\mathcal{AC},\infty \models A\ \mathsf{says}\ B\ \mathsf{can\ say}_D\ fact \\ \mathcal{AC},D \models B\ \mathsf{says}\ fact\end{array}}{\mathcal{AC},\infty \models A\ \mathsf{says}\ fact}$$

$$\text{(can act as)}\ \frac{\begin{array}{c}\mathcal{AC},D \models A\ \mathsf{says}\ B\ \mathsf{can\ act\ as}\ C \\ \mathcal{AC},D \models A\ \mathsf{says}\ C\ verbphrase\end{array}}{\mathcal{AC},D \models A\ \mathsf{says}\ B\ verbphrase}$$

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. Therefore, if D is 0, then the matching assertion must be proved without any application of the (can say) rule. If on the other hand D is ∞, then B can redelegate. In Section 5, we show that the boolean delegation flag $D \in \{0, \infty\}$ is sufficient for expressing a wide range of complex delegation policies, including depth-restricted delegation.

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.

Proposition 3.2. Let $\mathcal{AC}_A$ be the set of all assertions in $\mathcal{AC}$ whose issuer is A. We have $\mathcal{AC}, 0 \models A$ says *fact* iff $\mathcal{AC}_A, 0 \models A$ says *fact*.

Proposition 3.2 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 [36] also support depth restrictions, but these can be defeated as their constructs for depth-restricted delegation do not guarantee this 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:

$$q \ ::= \ e\ \mathsf{says}\ fact \mid q_1, q_2 \mid q_1\ \mathsf{or}\ q_2 \mid \mathsf{not}(q) \mid c \mid \exists x(q)$$

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 [47], higher computational complexity, or even undecidability [42]. Our solution is based on the observation that 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 [9]) or threshold operators (as in RT^T [39]), 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:

$$\begin{array}{ll}
\mathcal{AC},\theta \vdash e \textsf{ says } fact & \text{iff } \mathcal{AC},\infty \models e\theta \textsf{ says } fact\theta, \text{ and} \\
 & \qquad dom(\theta) \subseteq vars(e \textsf{ says } fact) \\
\mathcal{AC},\theta_1\theta_2 \vdash q_1,\, q_2 & \text{iff } \mathcal{AC},\theta_1 \vdash q_1 \text{ and } \mathcal{AC},\theta_2 \vdash q_2\theta_1 \\
\mathcal{AC},\theta \vdash q_1 \textsf{ or } q_2 & \text{iff } \mathcal{AC},\theta \vdash q_1 \text{ or } \mathcal{AC},\theta \vdash q_2 \\
\mathcal{AC},\varepsilon \vdash \textsf{not}(q) & \text{iff } \mathcal{AC},\varepsilon \nvdash q \text{ and } vars(q) = \emptyset \\
\mathcal{AC},\varepsilon \vdash c & \text{iff } \models c \\
\mathcal{AC},\theta_{-x} \vdash \exists x(q) & \text{iff } \mathcal{AC},\theta \vdash q
\end{array}$$

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 7 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 $\textsf{not}(q)$ or c are always ground when they are evaluated, under the assumption that conjunctive queries are evaluated from left to right (see Section 7).

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

$$\frac{fact \text{ is flat}}{I \Vdash e \textsf{ says } fact : vars(e \textsf{ says } fact) - I} \qquad \frac{vars(c) \subseteq I}{I \Vdash c : \emptyset}$$

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

$$\frac{I \Vdash q_1 : O_1 \qquad I \cup O_1 \Vdash q_2 : O_2}{I \Vdash q_1,\, q_2 : O_1 \cup O_2} \qquad \frac{I \Vdash q : O \qquad x \notin I}{I \Vdash \exists x(q) : O - \{x\}}$$

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

For example, the query x says y can read f, not(y says x can read f) is safe, because all variables occurring under the negation get instantiated by the left hand side of the conjunction. In contrast, x says y can read f, not(y says z can read f) is not safe, because z will not be instantiated by the time the negated subquery is evaluated.

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 operation names to queries. Upon a request, the resource guard calls an instantiated operation (instead of issuing a query directly) that gets mapped by the table to the corresponding authorization query, which is then used to query the assertion context.

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

$$\begin{array}{l}
\texttt{check-access-permission}(x): \\
\quad \texttt{FileServer} \textsf{ says } x \textsf{ has access from } t_1 \textsf{ till } t_2, \\
\quad t_1 \le \textsf{currentTime}() \le t_2, \\
\quad \textsf{not } \exists t_3, t_4(\\
\qquad \texttt{FileServer} \textsf{ says } x \textsf{ has no access from } t_3 \textsf{ till } t_4, \\
\qquad t_3 \le \textsf{currentTime}() \le t_4)
\end{array}$$

When the operation `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 [27, 23].

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/Mandatory Access Control (DAC/MAC) Assertion (6) from Section 1 is an example of a DAC policy: users with read permissions can delegate that right. Languages with restricted or no recursion such as XACML [41] or Lithium [29] cannot express such a policy. The path constraint in the assertion facilitates delegation at the granularity of single files within a hierarchical file system.

The support of hierarchical resources is a common requirement in practice, but existing authorization languages cannot express a policy such as Assertion (6). For example, in RT^C [38], the path constraint cannot take two variable arguments, as only unary constraints are allowed in order to preserve tractability. It is SecPAL's safety conditions that allow expressive constraint domains without losing efficiency.

As an example of a simple MAC policy, Assertions (8) and (9) below implement the Simple Security Property and the *-Property from the Bell-LaPadula model [11], respectively.

`FileServer` says x can read f if (8)
  x is a user, f is a file, $\text{level}(x) \geq \text{level}(f)$

`FileServer` says x can write f if (9)
  x is a user, f is a file, $\text{level}(x) \leq \text{level}(f)$

Roles 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).

`NHS` says `FoundationTrainee` can read `/docs/` (10)

`NHS` says `SpecialistTrainee` can act as (11)
  `FoundationTrainee`

`NHS` says `SeniorMedPractitioner` can act as (12)
  `SpecialistTrainee`

`NHS` says `Alice` can act as (13)
  `SeniorMedPractitioner`

The first assertion assigns a privilege to a role; the second and third establish seniority relations between roles; and the last assertion assigns Alice the role of a Senior Medical Practitioner. From these assertions it follows that `NHS` says `Alice` can read `/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 roles can add significant expressiveness to a role-based system and reduce the number of roles [28, 40]. In SecPAL, parameterized roles, attributes and privileges can be encoded by introducing verb phrases with arguments that correspond to the parameters, as in Assertion (14).

`NHS` says x can access health record of *patient* if (14)
  x is a treating clinician of *patient*

Separation of duties In this simple example of separation of duties, a payment transaction proceeds in two phases, initiation and authorization, which are required to be executed by two distinct bank managers. The following shows a fragment of the authorization query table. The operation `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 operation `can-authorize-payment` is called when a principal attempts to authorize a payment:

`can-initiate-payment`(*requester*,*payment*) : (15)
  `Bank` says *requester* is a manager,
  not($\exists x$(`Bank` says x has initiated *payment*))

`can-authorize-payment`(*requester*,*payment*) : (16)
  `Bank` says *requester* is a manager,
  `Bank` says x has initiated *payment*,
  $x \neq requester$

The requirement that a successful execution of `can-initiate-payment` adds a new fact to the assertion context is not specified within the authorization query table. However, SecPAL can be extended to support dynamic insertions and deletions of facts [8], which is also useful for RBAC policies with role activation and deactivation within sessions.

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 [39]. 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. We assume a function "distinct" that takes as argument a list of constants and returns `Yes` if the list contains no duplicates, and `No` otherwise. Since the assertion is safe, the list will be fully instantiated at the time the function is called.

`Alice` says x is trusted by `Alice` if (17)
  x is trusted by a, x is trusted by b, x is trusted by c,
  distinct$([a,b,c]) =$ `Yes`

`Alice` says x can say$_\infty$ y is trusted by x if (18)
  x is trusted by `Alice`

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.

$$\begin{aligned}&\texttt{Shop}\ \textsf{says}\ x\ \textsf{is entitled to discount if}\\&\quad x\ \textsf{is a student till}\ \mathit{date},\\&\quad \textsf{currentTime}() \le \mathit{date},\ \textsf{currentDay}() = \texttt{Friday}\end{aligned}\tag{19}$$

$$\begin{aligned}&\texttt{Shop}\ \textsf{says}\ \mathit{univ}\ \textsf{can say}_\infty\ x\ \textsf{is a student till}\ \mathit{date}\ \textsf{if}\\&\quad \mathit{univ}\ \textsf{is a university}\end{aligned}\tag{20}$$

$$\begin{aligned}&\texttt{Shop}\ \textsf{says}\ \texttt{BoardOfEducation}\ \textsf{can say}_\infty\\&\quad \mathit{univ}\ \textsf{is a university}\end{aligned}\tag{21}$$

SPKI/SDSI [26], DL [36], Binder [22], RT [39] and Cassandra [10] 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.

$$\begin{aligned}&\texttt{FileServer}\ \textsf{says}\ \texttt{STS}\ \textsf{can say}_\infty\\&\quad x\ \textsf{has access from}\ \mathit{t1}\ \textsf{till}\ \mathit{t2}\ \textsf{if}\\&\quad \mathit{t2} - \mathit{t1} \le 8\ \textsf{hours}\end{aligned}\tag{22}$$

The delegation depth in Assertion (22) is unlimited, so STS can in turn delegate the same right to some STS2, possibly with additional constraints. For example, with Assertion (23) 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).

$$\begin{aligned}&\texttt{STS}\ \textsf{says}\ \texttt{STS2}\ \textsf{can say}_0\ x\ \textsf{has access from}\ \mathit{t1}\ \textsf{till}\ \mathit{t2}\ \textsf{if}\\&\quad \mathit{t1} \ge \texttt{01/01/2007}\end{aligned}\tag{23}$$

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.

$$\texttt{Alice}\ \textsf{says}\ \texttt{Bob}\ \textsf{can say}_0\ x\ \textsf{is a friend}\tag{24}$$

$$\texttt{Alice}\ \textsf{says}\ \texttt{Bob}\ \textsf{can say}_0\ x\ \textsf{can say}_0\ y\ \textsf{is a friend}\tag{25}$$

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 [20] 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:

$$\texttt{Charlie}\ \textsf{says}\ x\ \textsf{is a friend if}\ x\ \textsf{is a friend2}\tag{26}$$

$$\texttt{Charlie}\ \textsf{says}\ \texttt{Doris}\ \textsf{can say}_0\ x\ \textsf{is a friend2}\tag{27}$$

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 3.2. 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.

$$\begin{aligned}&\texttt{Alice}\ \textsf{says}\ x\ \textsf{can say}_0\ y\ \textsf{is a friend if}\\&\quad x\ \textsf{is a delegator}\end{aligned}\tag{28}$$

$$\texttt{Alice}\ \textsf{says}\ \texttt{Bob}\ \textsf{is a delegator}\tag{29}$$

$$\begin{aligned}&\texttt{Alice}\ \textsf{says}\ x\ \textsf{can say}_0\ y\ \textsf{is a delegator if}\\&\quad x\ \textsf{is a delegator},\\&\quad y\ \textsf{possesses}\ \texttt{Email}\ \mathit{email},\\&\quad \mathit{email}\ \textsf{matches}\ \texttt{*@fabrikam.com}\end{aligned}\tag{30}$$

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 Datalog with Constraints

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

Our terminology for Datalog with Constraints is as follows. (See [17] or [3] for a detailed introduction to Datalog and [43, 42] for Datalog with Constraints.) 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 *clause*, written $P_0 \leftarrow P_1, \ldots, P_n, c$, consists of a *head* literal, a list of *body* literals, and a constraint. 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 $T_\mathcal{P}$, denoted by $T^\omega_\mathcal{P}(\emptyset)$. Intuitively, this is the set of 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 can say_∞ y can say_0 B can act as z is shorthand for says_can_say_infinity_can_say_zero_can_act_as(A,k,x,y,B,z).

Algorithm 6.1. 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 can say_{K_0} ... e_{n-1} can $\mathsf{say}_{K_{n-1}}$ *fact*, for some $n \geq 1$, where *fact* is flat. Let $\hat{fact}_n \equiv fact$ and $\hat{fact}_i \equiv e_i$ 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 says_k $\hat{fact}_0 \leftarrow A$ says_k $fact_1, ..., A$ says_k $fact_m, c$

   where k is a fresh variable.

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

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

   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.

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 6.2 (Soundness and completeness). Let $\mathcal{P}$ be the Datalog translation of the assertion context $\mathcal{AC}$. We have A says_D $fact \in T^\omega_\mathcal{P}(\emptyset)$ 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_∞ *fact* (i.e., computing all query instances that are in $T^\omega_\mathcal{P}(\emptyset)$) 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 [45, 24, 19]. Tabling has also been applied to Datalog with Constraints, but requires complex constraint solving procedures [46].

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

```
RESOLVE-CLAUSE($\langle P \rangle$)
  $Ans(P) := \emptyset$;
  foreach $(Q \leftarrow \vec{Q}, c) \in \mathcal{P}$ do
    if $nd = resolve(\langle P; Q :: \vec{Q}; c; Q; [\,]; Cl \rangle, P)$
      exists then
      PROCESS-NODE($nd$)

PROCESS-ANSWER($nd$)
  match $nd$ with $\langle P; [\,]; c; \_; \_; \_ \rangle$ in
    if $nd \notin Ans(P)$ then
      $Ans(P) := Ans(P) \cup \{nd\}$;
    foreach $nd' \in Wait(P)$ do
      if $nd'' = resolve(nd', nd)$ exists then
        PROCESS-NODE($nd''$)

PROCESS-NODE($nd$)
  match $nd$ with $\langle P; \vec{Q}; c; \_; \_; \_ \rangle$ in
    if $\vec{Q} = [\,]$ then
      PROCESS-ANSWER($nd$)
    else match $\vec{Q}$ with $Q_0 :: \_$ in
      if there exists $Q_0' \in dom(Ans)$
          such that $Q_0 \Rightarrow Q_0'$ then
        $Wait(Q_0') := Wait(Q_0') \cup \{nd\}$;
        foreach $nd' \in Ans(Q_0')$ do
          if $nd'' = resolve(nd, nd')$ exists then
            PROCESS-NODE($nd''$)
      else
        $Wait(Q_0) := \{nd\}$;
        RESOLVE-CLAUSE($\langle Q_0 \rangle$)
```

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

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. The most general unifier of literals P and Q is denoted by $mgu(P,Q)$. 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 $d \equiv simplify(c\theta) \neq$ `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.

The function $Answers_{\mathcal{P}}$ evaluates atomic authorization queries. Based on this function, the following algorithm evaluates general authorization queries that are constructed from atomic ones. Let $\mathcal{AC}$ be an assertion context and $\mathcal{P}$ its Datalog translation. The function $AuthAns_{\mathcal{AC}}$ on authorization queries is defined in Figure 2.

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

Theorem 7.1 (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, 21] 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 7.2. Let M be the number of flat atomic assertions (i.e., those without conditional facts) in $\mathcal{AC}$ and let N

$$
\begin{array}{rcl}
\mathit{AuthAns}_{\mathcal{AC}}(e\ \textsf{says}\,\mathit{fact}) & = & \mathit{Answers}_{\mathcal{P}}(e\ \textsf{says}_\infty\,\mathit{fact},\ \emptyset) \\
\mathit{AuthAns}_{\mathcal{AC}}(q_1,\, q_2) & = & \{\theta_1\theta_2 \mid \theta_1 \in \mathit{AuthAns}_{\mathcal{AC}}(q_1) \text{ and } \theta_2 \in \mathit{AuthAns}_{\mathcal{AC}}(q_2\theta_1)\} \\
\mathit{AuthAns}_{\mathcal{AC}}(q_1\ \textsf{or}\ q_2) & = & \mathit{AuthAns}_{\mathcal{AC}}(q_1) \cup \mathit{AuthAns}_{\mathcal{AC}}(q_2) \\[1ex]
\mathit{AuthAns}_{\mathcal{AC}}(\textsf{not}(q)) & = & \begin{cases} \{\varepsilon\} & \text{if } \mathit{vars}(q) = \emptyset \text{ and } \mathit{AuthAns}_{\mathcal{AC}}(q) = \emptyset \\ \emptyset & \text{if } \mathit{vars}(q) = \emptyset \text{ and } \mathit{AuthAns}_{\mathcal{AC}}(q) \neq \emptyset \\ \text{undefined} & \text{otherwise} \end{cases} \\[1ex]
\mathit{AuthAns}_{\mathcal{AC}}(c) & = & \begin{cases} \{\varepsilon\} & \text{if } \models c \\ \emptyset & \text{if } \mathit{vars}(c) = \emptyset \text{ and } \not\models c \\ \text{undefined} & \text{otherwise} \end{cases} \\[1ex]
\mathit{AuthAns}_{\mathcal{AC}}(\exists x(q)) & = & \{\theta_{-x} \mid \theta \in \mathit{AuthAns}_{\mathcal{AC}}(q)\}
\end{array}
$$

Figure 2. SecPAL evaluation algorithm

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.

8. Discussion

Related work The ABLP logic [2, 35] introduced the "says" modality and the use of logic rules for expressing decentralized authorization policies. The semantics of SecPAL's delegation operators "can say$_\infty$" and "can act as" are related to the "controls" and "speaks for" operators, respectively, of ABLP. In current work, Abadi and Garg are investigating translations from SecPAL to the ABLP logic.

PolicyMaker and Keynote [16, 15] introduced the notion of decentralized trust management. Quite a few other authorization languages have been developed since. SPKI/SDSI [26] 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 [44], or as *k*-out-of-*n* threshold subjects. Grants can have validity restrictions and indicate whether they may be delegated.

XrML [20] (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 [41] 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.

Policy languages such as Binder [22], SD3 [34], Delegation Logic (DL) [36] and the RT family of languages [39] 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 [10, 9] and RT^C [38] are based on Datalog with Constraints [32, 42] 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. [12, 14, 33, 5, 48]. 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 multi-domain grid computing environments [25]. The implementation is XML-based and makes use of web services protocols for interoperability. Active Directory, Kerberos and X.509 infrastructures are used for key and identity management. 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 [25] presents the design goals and introduces the language informally. Gregory Fee and Jason Mackay implemented the SecPAL prototype. We also thank Martín Abadi, Blair Dillaway, Peter Sewell, Vicky Weissman, Tuomas Aura, Michael Roe, Sebastian Nanz and the anonymous referees for valuable comments.

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, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. Technical report, Microsoft Research, 2006. `research.microsoft.com/research/pubs/view.aspx?tr_id=1166`.

[8] M. Y. Becker and S. Nanz. A logic for state-modifying authorization policies. Technical Report MSR-TR-2007-32, Microsoft Research, 2007. `research.microsoft.com/research/pubs/view.aspx?tr_id=1274`.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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.

[15] 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.

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

[17] 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.

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

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

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

[21] 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.

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

[23] 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.

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

[25] 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.

[26] 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`.

[27] 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.

[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] 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.

[36] 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.

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

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

[39] 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.

[40] 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.

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

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

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

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

[45] 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.

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

[47] 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.

[48] 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.

[49] 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`.

[50] 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.