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 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 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.

---

1Personal 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:

STSSays Alice is a researcher (1)
FileServerSays Alice can read /project (2)
AliceSays Cluster can read /project/data if currentTime() ≤ 07/09/2006 (3)

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

Before processing the request, the cluster authenticates Alice as the requester, validates her tokens, and runs the query Cluster says Alice can execute dbgrep against the set of assertions formed by its local policy plus these tokens. (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:

Cluster says STS can say₀ x is a researcher (4)
Cluster says x can execute dbgrep if x is a researcher (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

FileServerSays x can sayₙ y can read file if x can read dir, file ∈ dir, markedConfidential(file) ≠ Yes (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 = Alice and y = 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 currentTime() ≤ 07/09/2006 and markedConfidential(/project/data) ≠ Yes.

In the delegation rules (4) and (6), the “can say” assertions have different subscripts: in (4), can say₀ prevents STS from re-delegating the delegated fact; conversely, in (6), can sayₙ 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 = Alice then x = Cluster. Alternatively, FileServer may simply issue the assertion

FileServerSays Node23 can act as Cluster (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 AC of assertions (called assertion context) of the form

A says fact if fact₁, ..., factₙ, c

where the facts are sentences that state properties on principals, defined below. In the assertion, A is the issuer; fact₁, ..., factₙ 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.

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

Constraints Expressions \( r \) occurring in constraints range over variables, constants and applications \( f(r_1, \ldots, r_n) \) of built-in functions such as \( \text{CurrentTime()} \). Constraints range over any constraint domain that includes the following basic constraints: the trivial constraint \( \text{True} \), 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 \( r \text{ matches regExp} \) for ad hoc string filtering. Constraint domains are closed under variable renaming, conjunction \( (c_1, c_2) \), and negation \( (\text{not}(c)) \).

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 \( \text{ground} \) 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 \( \text{CurrentTime()} \). 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. \( \text{False}, r_1 \neq r_2, \text{ and } c_1 \lor c_2 \). We usually omit the \( \text{True} \) constraint, and also omit the \( \text{if} \) in assertions with no conditional facts, writing \( A \text{ says } f \) for \( A \text{ says } f \text{ if True} \). We write keywords, function names and predicates in \textit{sans serif}, constants in \texttt{typewriter} font, and variables in \textit{italics}. We use a vector notation to denote a (possibly empty) list of items, e.g. writing \( f(i) \) for \( f(r_1, \ldots, r_n) \).

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 \( f \text{ if } \text{fact}_1, \ldots, \text{fact}_n, c \) is \textit{safe} iff the following conditions hold:

- all conditional facts are flat;
- all variables in \( c \) also occur somewhere else in the assertion;
- if \( \text{fact} \) is flat, all variables in \( \text{fact} \) 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 \( \text{FileServer says } x \text{ can read } \text{Foo} \) (everybody can read \( \text{Foo} \)). 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 say₀ x is a user`. Alternatively, the server may accept self-issued statements: `FileServer says x can say₀ 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 $\theta$ be a function mapping variables to constants and variables, and let $\varepsilon$ 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 $\text{vars}(X)$ 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 $\mathcal{AC}, D \models A \text{ says fact}$ where $\text{vars(fact)} = \emptyset$ and the delegation flag $D$ is 0 or $\infty$. 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.

\[
\begin{align*}
(A \text{ says fact} \text{ if fact}_1, \ldots, \text{fact}_k \text{ where } c) \in \mathcal{AC} \\
\mathcal{AC}, D \models A \text{ says fact}_i \theta \text{ for all } i \in \{1..k\} \\
\quad \models c \theta \\
\quad \text{vars(fact)} = \emptyset \\
\hline
\mathcal{AC}, D \models A \text{ says fact}\theta
\end{align*}
\]

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 $\infty$. 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 $D = \infty$, 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 $\mathcal{AC}, 0 \models A \text{ says fact}$ iff $\mathcal{AC}_A, 0 \models A \text{ says fact}$.

Proposition 3.2 implies that if $A$ says $\text{ fact}$ is deduced from a zero-depth delegation assertion $A$ says $B$ can say₀ $\text{ fact}$ 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 $A \text{ says fact}$ and constraints, combined by logical connectives including negation:

\[
q ::= e \text{ says fact} \mid q_1 \land q_2 \mid \neg q_1 \lor q_2 \mid \text{ not}(q) \mid c \lor \exists x(q)
\]

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 $\theta_{\rightarrow x}$ to denote the substitution that has domain $\text{dom}(\theta) = \{x\}$ and is equivalent to $\theta$ on this domain. The semantics of authorization queries is defined by the relation $\mathcal{AC}, \theta \vdash q$, as follows:
Definition 4.1 (Authorization query safety). An authorization query \( q \) is safe iff there exists a set of variables \( O \) such that \( \emptyset \models q : O \).

For example, the query \( x \text{ says } y \text{ can read } f \), \( \text{not}(y \text{ says } x \text{ can read } f) \) is safe, because all variables occurring under the negation get instantiated by the left hand side of the conjunction. In contrast, \( x \text{ says } y \text{ can read } f, \text{not}(y \text{ says } z \text{ can read } f) \) 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:

\[
\text{check-access-permission}(x) :
\]
\[
\text{FileServer says } x \text{ has access from } t_1 \text{ till } t_2, \\
t_1 \leq \text{currentTime}() \leq t_2, \\
\text{not } \exists_{t_3, t_4}( \\
\text{FileServer says } x \text{ has no access from } t_3 \text{ till } t_4, \\
t_3 \leq \text{currentTime}() \leq t_4)
\]

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.

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

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

**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).

\[
\text{NHS says } \text{FoundationTrainee can read /docs/} \quad (10)
\]

\[
\text{NHS says } \text{SpecialistTrainee can act as } \text{FoundationTrainee} \quad (11)
\]

\[
\text{NHS says } \text{SeniorMedPractitioner can act as } \text{SpecialistTrainee} \quad (12)
\]

\[
\text{NHS says } \text{Alice can act as } \text{SeniorMedPractitioner} \quad (13)
\]

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).

\[
\text{NHS says } x \text{ can access health record of patient if } x \text{ is a treating clinician of patient} \quad (14)
\]

**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:

\[
\text{can-initiate-payment}\(\text{requester, payment}\) : \quad (15)
\]

\[
\text{can-authorize-payment}\(\text{requester, payment}\) : \quad (16)
\]

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.

\[
\text{Alice says } x \text{ is trusted by Alice if } x \text{ is trusted by } a, x \text{ is trusted by } b, x \text{ is trusted by } c, \text{distinct }([a, b, c]) = \text{Yes} \quad (17)
\]

\[
\text{Alice says } x \text{ can say } y \text{ is trusted by } x \text{ if } x \text{ is trusted by Alice} \quad (18)
\]

**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
Delegators may wish to restrict constrained delegation to constrain the shape of delegation trees. There are other general techniques 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.

\[
\text{Shop says } x \text{ is entitled to discount if } \begin{align*}
  & x \text{ is a student till date,} \\
  & \text{currentTime()} \leq \text{date, } \text{currentDay()} = \text{Friday}
\end{align*}
\]  

(19)

\[
\text{Shop says } \text{univ} \text{ can say}_m x \text{ is a student till date if }
\]

(20)

\[
\text{Shop says } \text{BoardOfEducation} \text{ can say}_m
\]

(21)

\[
\text{univ is a university}
\]

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.

\[
\text{FileServer says STS can say}_m x \text{ has access from } t_1 \text{ till } t_2 \text{ if } t_2 - t_1 \leq 8 \text{ hours}
\]

(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).

\[
\text{STS says STS2 can say}_0 x \text{ has access from } t_1 \text{ till } t_2 \text{ if } t_2 - t_1 \geq 01/01/2007
\]

(23)

**Depth-bounded delegation** The verb phrase can say\(_m\) fact allows no further delegation of fact, while can say\(_0\) 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.

Suppose Bob re-delegates to Charlie with the assertion \(\text{Bob says}\) \(\text{Doris can say}_0 x \text{ is a friend}\). Since Alice does not accept any longer delegation chains, Alice (in contrast to Bob) does not allow Charlie to re-delegate with \(\text{Charlie says}\) \(\text{Doris can say}_0 x \text{ is a friend}\).

In XrML and DL, the delegation depth can be specified, and can be either an integer or \(\infty\). However, in both languages, the depth restrictions can be defeated by Charlie:

\[
\text{Charlie says } x \text{ is a friend if } x \text{ is a friend}_2
\]

(26)

\[
\text{Charlie says } \text{Doris can say}_0 x \text{ is a friend}_2
\]

(27)

**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 a friend facts, and to transitively re-delegate this attribute, but only amongst principals with a matching email address.

\[
\text{Alice says } x \text{ can say}_0 y \text{ is a friend if } x \text{ is a delegator}
\]

(28)

\[
\text{Alice says } \text{Bob is a delegator}
\]

(29)

\[
\text{Alice says } x \text{ can say}_0 y \text{ is a delegator if } x \text{ is a delegator, } y \text{ possesses Email email, email matches } ^*@\text{fabrikam.com}
\]

(30)

If these are the only assertions by Alice that mention the predicate is a friend or is a delegator, then any derivation of \(\text{Alice says } x \text{ is a friend}\) 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, \( P \), is a finite set of clauses. The semantics of a program \( P \) is the least fixed point of the standard immediate consequence operator \( T_P \), denoted by \( T_P^0(\emptyset) \). Intuitively, this is the set of all ground literals derivable from \( P \).

We treat expressions of the form \( e \) \( \text{say}_k \) \( fact \) as Datalog literals, where \( k \) is either a variable or 0 or \( \infty \). This can be seen as a sugared notation for a literal where the predicate name is the string concatenation of all infix operators (\( \text{say}_k \), \( \text{can} \), \( \text{say} \), \( \text{act} \), and \( \text{predicates} \)) occurring in the expression, including subscripts for \( \text{say} \). The arguments of the literal are the collected expressions between these infix operators. For example, the expression \( A \text{say}_k x \text{can say}_\infty y \text{can say}_0 B \text{can act as } z \) is shorthand for \( \text{say}_\infty \text{can say}_\infty \text{can say}_\infty \text{can say}_\infty \text{can say}_\infty \text{can say}_\infty \text{can act as } (A, k, x, y, B, z) \).

Algorithm 6.1. The translation of an assertion context \( AC \) proceeds as follows:

1. If \( fact_0 \) is flat, then an assertion \( A \text{say}_k fact_0 \) if \( fact_1, \ldots, fact_n, c \) is translated into the clause \( A \text{say}_k fact_0 \leftarrow A \text{say}_k fact_1, \ldots, A \text{say}_k fact_n, c \) where \( k \) is a fresh variable.

2. Otherwise, \( fact_0 \) is of the form \( e_0 \text{can say}_k \ldots \text{can say}_n \text{fact} \), for some \( n \geq 1 \), where \( \text{fact} \) is flat. Let \( fact_n = fact \) and \( fact_i = e_i \text{can say}_k fact_{i+1} \), for \( i \in \{0, \ldots, n-1\} \). Note that \( fact_0 = fact \). Then the assertion \( A \text{say}_k fact_0 \text{if } fact_1, \ldots, fact_n, c \) is translated into a set of \( n + 1 \) Datalog clauses as follows:
   (a) We add the Datalog clause \( A \text{say}_k fact_0 \leftarrow A \text{say}_k fact_1, \ldots, A \text{say}_k fact_n, c \) where \( k \) is a fresh variable.
   (b) For each \( i \in \{1, \ldots, n\} \), we add a Datalog clause \( A \text{say}_\infty fact_i \leftarrow x \text{say}_k fact_{i-1}, \text{fact}_i \), \( A \text{say}_\infty x \text{can say}_k fact_{i-1}, \text{fact}_i \), where \( x \) is a fresh variable.

3. Finally, for each Datalog clause created above with head \( A \text{say}_k x \text{verbphrase} \) we add a clause
   \[ A \text{say}_k x \text{verbphrase} \leftarrow A \text{say}_k x \text{can act as } e, A \text{say}_k e \text{verbphrase} \]
   where \( x \) is a fresh variable.

Intuitively, the \( \text{say} \) 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 \( P \) be the Datalog translation of the assertion context \( AC \). We have \( A \text{say}_D fact \in T_P^0(\emptyset) \) iff \( AC, D \models A \text{say}_C fact \).

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 \( e \text{say}_\infty fact \) (i.e., computing all query instances that are in \( T_P^0(\emptyset) \)) against the Datalog program \( P \) 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 \( \langle P; \tilde{Q}; c; S; \tilde{a}; Cl \rangle \), where \( \tilde{Q} \) is a list of literals (the subgoals), \( c \) a constraint, \( S \) a literal (the partial answer), \( \tilde{a} \) 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 \( \tilde{Q} \) is empty and \( c = \text{True} \)). The set \( Ans(P) \) is used to store all the found answer nodes pertaining to a query \( P \). 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((P))
Ans(P) := ∅;
foreach (Q ← Q, c) ∈ P do
  if nd = resolve((P; Q :: Q, c; [ ]; Cl), P) exists then
    PROCESS-NODE(nd)

PROCESS-ANSWER(nd)
match nd with (P; [ ]; c; ⃗P) 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′)

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

answers from ⟨P⟩. Whenever a new answer for ⟨P⟩ 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 = Qθ for some substitution θ, in which case we write P ⊢ Q.

A node nd = ⟨P; Q :: ⃗Q, c; ⃗S; nd; Cl⟩ and a literal Q are resolvable iff some Q′ is a fresh variable renaming of Q′, θ = mgu(Q, Q′) exists and d = simplify(cθ) ≠ False. Their resolvent is nd′ = ⟨P; Qθ; d; Sθ; nd; Cl⟩, and θ is their resolution unifier. We write resolve(nd, Q′) = nd′ if nd and Q′ are resolvable. By extension, a node nd = ⟨P; Q :: ⃗Q, c; ⃗S; nd; Cl⟩ and an answer node nd′ = ⟨P; Qθ; d; Sθ; ndθ; [ ]; Cl⟩. We write resolve(nd, nd′) = nd″ if nd and nd′ are resolvable. The function simplify(θ) is defined as

{θ : ⟨⃗S; ⃗Q⟩ ∈ Ans(P′), S = Pθ, dom(θ) ⊆ vars(P)}

if there exists a literal P′ ∈ dom(Ans) such that P ⇒ 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((P)) terminates with initial answer table Ans and an initially empty wait table, then AnswersP,Ans is defined as

{θ : ⟨⃗S; ⃗Q⟩ ∈ Ans′(P), S = Pθ, dom(θ) ⊆ vars(P)}

where Ans′ is the modified answer table after the call. In all other cases AnswersP,Ans is undefined.

The function AnswersP evaluates atomic and non-atomic queries. Based on this function, the following algorithm evaluates general authorization queries that are constructed from atomic ones. Let AC be an assertion context and P its Datalog translation. The function AuthAnsAC on authorization queries is defined in Figure 2.

The following theorem shows that AuthAnsAC is an algorithm for evaluating safe queries.

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

1. AuthAnsAC(q) is defined and finite, and
2. AC, θ ⊨ q iff θ ∈ AuthAnsAC(q).

The evaluation of the base case e says fact calls the function AnswersP 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 AnswersP.

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 AC and let N
be the maximum length of constants occurring in these assertions. The time complexity of computing \( \text{AuthAns}_{\text{AC}} \) 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” 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 autho-
rization 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 Martin Abadi, Blair Dillaway, Peter Sewell, Vicky Weissman, Tuomas Aura, Michael Roe, Sebastian Nanz and the anonymous referees for valuable comments.

References