An Abductive Protocol for Authorization Credential Gathering in Distributed Systems

Moritz Y. Becker Jason F. Mackay[1]
Blair Dillaway[1]

[1]Microsoft Corporation, One Microsoft Way, Redmond, WA 98052

{moritzb,jmackay,blaird}@microsoft.com

February 2009

Technical Report
MSR–TR–2009–19

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

An Abductive Protocol for Authorization Credential Gathering in Distributed Systems

Moritz Y. Becker Jason F. Mackay[1]
Blair Dillaway[1]
[1]Microsoft Corporation, One Microsoft Way, Redmond, WA 98052
{moritzb,jmackay,blaird}@microsoft.com

February 2009

Abstract

The problem of authorization in large-scale decentralized systems has been addressed by a number of logic-based policy languages utilizing delegation of authority and distributed security credentials. A central task in this context is that of gathering a set of credentials for a given access request. Previous approaches have focused on methods in which credentials are pulled on-demand from credential providers during authorization. These methods may result in multiple, and potentially futile, costly queries to the same remote credential provider, and require that providers be known and available to the resource guard at access time. A novel decentralized protocol is presented in this paper to address these shortcomings. The approach uses logical abduction to statically and locally compute a specification of credentials needed to satisfy a given query against a policy. Based on such a specification, credentials are gathered using a single-pass protocol that queries each provider only once and does not involve any communication with the resource guard. The credentials gathered thus are then pushed to the resource guard at authorization time. This approach decouples authorization from credential gathering, and, in comparison to server-side pull methods, reduces the number of messages sent between participants, and allows for communication topologies in which some credential providers are unknown or unavailable to the resource guard at authorization time.

1 Introduction

Large-scale decentralized systems present unique challenges for authorization and access control. Several logic-based authorization policy languages specialized for such environments have emerged which leverage the concept of delegated authority in order to remove the need for centralized control (e.g. [10, 15, 4, 2]). Credentials in these systems are not stored in a central location but rather in a distributed manner; in fact they may be stored anywhere as long as they are made available to the resource guard at the time of authorization. Requiring that users gather the credentials themselves and *push* them to the resource guard has been considered problematic

[8], because the expressiveness of policy languages can make it difficult for a human to understand precisely what kinds of credentials are required. Furthermore, the user generally does not, should not need to, know the policy. An automated method for distilling such requirements from a policy and an access request and then gathering the right credentials is therefore desirable.

Previous work in this area has focused on server-side on-demand *pull* methods [8, 1, 17, 10, 4, 13] to take the burden off the user. In these approaches, the resource guard attempts to construct a proof for the access request based on local policy and a set of provided credentials. Whenever a required credential is not locally available during the proof process, an attempt is made to retrieve it from some remote credential provider. Consider the following schematic example (written in SecPAL [2]):

$\texttt{Srv}\ \textsf{says}:\ x\ \textsf{can read}\ f\ \textsf{if}\ x\ \textsf{is a}\ \texttt{Mgr}\ \textsf{in}\ \mathit{dept},\ \mathit{dept}\ \textsf{owns}\ f$

$\texttt{Srv}\ \textsf{says}:\ \texttt{Bob}\ \textsf{can say}_0\ x\ \textsf{is a}\ \mathit{role}\ \textsf{in}\ \mathit{dept}$

When Alice requests to read a file `foo`, the proof process first tries to prove that she is a manager of some arbitrary department *dept*. Authority over role memberships has been delegated to Bob (by the second assertion), so in the absence of any relevant local information, the resource guard attempts to pull all credentials of the form $\texttt{Bob}\ \textsf{says}\ \texttt{Alice}\ \textsf{is a}\ \texttt{Mgr}\ \textsf{in}\ \mathit{dept}$ from a suitable remote credential provider (e.g. `Bob`). When the credentials have arrived, the proof can proceed with the second condition, $\mathit{dept}\ \textsf{owns}\ \texttt{foo}$. There are two problems with the pull-based approach:

1. **Communication cost.** The proof of the second condition may fail based on local information, in which case the costly communication with the credential provider was futile. Similarly, the second condition may turn out to force a constraint on *dept*, which could have made the remote query narrower and more efficient, if the constraint had been known in advance. Finally, the proof of the second condition may itself require missing credentials from the same provider, resulting in multiple, separate message exchanges.
2. **Connectivity requirements.** The proof process fails (or becomes inaccurate) if any of the credential providers required during the proof happens to be unavailable to the resource guard at authorization time, for example due to unexpected downtime, or because the provider is behind a firewall, or because human interaction is required at the provider's site. Also, there are cases where the location of a missing credential is not discoverable by the resource guard, but is determined by another part of the workflow.

This paper presents a novel *push*-based method for gathering credentials in distributed systems that addresses these two problems and yet does not require the user (or any principal other than the resource guard) to know the policy. To deal with the first problem, we present an algorithm for statically precomputing a complete specification of missing credentials, without requiring remote communication. This goal-directed algorithm performs logical abduction over constrained Datalog, using memoization to improve efficiency. To deal with the second problem, we present a distributed single-pass protocol for collecting credentials, based on the abductive specification, that does not involve the resource guard (or any other central pull-mechanism) at all, and therefore does not require simultaneous availability of providers at access time or at any other time. In fact, the protocol makes only very weak assumptions on the connectivity of participating parties: arbitrary communication paths through the providers are supported, enabling a wider variety of scenarios and minimizing communications overhead compared to the server-side pull approach. The credentials gathered thus are guaranteed to sufficiently support the query, as long as the policy is not modified during the run of the protocol, and are pushed to the resource guard at access time. The algorithms and protocols are described in the context of SecPAL [2], a highly expressive policy language that can be translated into constrained recursive Datalog, but are applicable to any similar language of equal or lesser expressiveness, as long as it supports decentralized delegation.

The organization of the remainder of this paper is as follows. Section 2 defines SecPAL, the authorization language on which this work is based. Section 3 presents an abductive reasoning algorithm for computing a complete specification of credentials that would result in success for a given access request. Section 4 describes a protocol which, given such a specification, incrementally gathers satisfying credentials from a set of credential providers. Section 5 illustrates the protocol in the context of an example scenario based on electronic health records. Section 6 provides some insights gained through the implementation of a prototype system. Related work is discussed in Section 7. Section 8 concludes the paper with a discussion of limitations and future work.

2 Security Policy Assertions

This section defines the core of the Security Policy Assertion Language (SecPAL), an authorization language we are developing for large-scale federated systems. We leave out language features that are irrelevant to this paper.

Syntax *Expressions e* are either typed variables (written in lowercase *italics* in this paper) or constants (including principal identifiers, resources, role names, date times, etc., written in `typewriter font`). We use U as a meta-variable for constant principal identifiers. A syntactic phrase is *ground* if it is variable-free.

A *fact* is a simple sentence consisting of a principal expression (the *subject*) followed by a verb phrase. A *verb phrase* is a typed predicate with parameters, usually written in infix notation in order to resemble natural language. There are *flat* and *nested* verb phrases (and, by extension, facts). Flat verb phrases can be defined by application writers and do not have any intrinsic semantics; for example, in the scenario in Section 5 we use four application-specific flat facts

e_1 can access e_2's data
e_1 is a e_2
e_1 is treating e_2 (from e_3 until e_4)
e_2 has e_1's consent (from e_3 until e_4)

Nested verb phrases are of the form can say$_K$ *fact* where K is either 0 or ∞. Note that *fact* itself can again be nested, so can say$_K$ can be used to construct arbitrarily long verb phrases. Nested verb phrases have a built-in semantics defined below.

A *constraint* is a formula that can be efficiently evaluated to true or false when ground. The constraint language consists of base constraints and is closed under conjunction, disjunction and negation. Base constraint types can be added depending on application requirements by providing a procedure for evaluating ground constraints; our implementation has a set of predefined base constraints including disequality, arithmetic inequalities and regular expression matching.

Assertions α are of the form

e says : *fact* if $\mathit{fact}_1,...,\mathit{fact}_n$ where c

The expression e is a principal called the *issuer*. The fact before the if-clause is the *concluding fact*. The facts of the if-clause are the *conditional facts* and do not mention can say$_K$. The formula c inside the where-clause is the *constraint* of the assertion.

If $n = 0$, the entire if-clause is omitted; similarly, the where-clause is omitted if $c = \texttt{True}$. An assertion with no if-clause and no where-clause is called an *atomic* assertion.

We impose a *safety condition* on assertions that guarantee completeness and termination of SecPAL evaluation: the issuer e must be ground; all variables in c must occur in the rest of the assertion; and when the concluding fact is flat, all its variables must occur in the conditional facts. Note that any safe atomic assertion not mentioning can say is always ground. Only safe assertions will be considered henceforth unless explicitly noted otherwise.

Assertions can be stored in various ways: as parseable text written in the concrete syntax described above; as objects in main memory; as items stored in a relational database; or as XML-encoded *credentials*, signed by the assertion's issuer. When assertions are sent over the network, they will usually be in the form of credentials.

An *authorization policy* is a finite set of assertions. Finally, a *query* q is a possibly unsafe atomic assertion.

Semantics We give SecPAL a proof-theoretic semantics that defines proof rules for deriving ground atomic assertions α from a given policy $\mathcal{P}$. The judgments are of the form $\mathcal{P} \vdash \alpha$. The following proof rule schemas are a simpler and more intuitive version of the ones in an earlier paper [2], but formally equivalent.

The first rule formalizes the intuition that the concluding fact can be deduced if all conditional facts can be deduced in such a way that the constraint is satisfied:

$$\frac{\mathcal{P} \vdash U \text{ says : } \gamma(\mathit{fact}_i) \quad \text{for all } i \in \{1..k\}}{\mathcal{P} \vdash U \text{ says : } \gamma(\mathit{fact})}$$

provided that there exists an assertion

$\langle U$ says : *fact* if $\mathit{fact}_1,...,\mathit{fact}_k$ where $c\rangle$

in the policy $\mathcal{P}$, γ is a ground variable substitution (a total mapping from variables to ground expressions), and $\gamma(c)$ evaluates to true. Note that this rule reduces to an axiom when $k = 0$.

The special verbs can say$_\infty$ and can say$_0$ express delegation of authority over some fact. In the following two rules, U_1 is the *delegator* and U_2 the *delegatee*:

$$\frac{\mathcal{P} \vdash U_1 \text{ says : } U_2 \text{ can say}_\infty \mathit{fact} \qquad \mathcal{P} \vdash U_2 \text{ says : } \mathit{fact}}{\mathcal{P} \vdash U_1 \text{ says : } \mathit{fact}}$$

For example, if $\mathcal{P}$ consists of

> `Alice` **says :** `Bob` **can say**$_\infty$ x **can read** f
> `Bob` **says :** `Charlie` **can say**$_\infty$ x **can read** f
> `Charlie` **says :** `Doris` **can read** `file:///foo/`

we can deduce
$\mathcal{P} \vdash$ `Alice` says : `Doris` can read `file:///foo/`,
using the above rule. In this example, Alice delegates authority over read access to Bob, and Bob *re-delegates* this authority to Charlie.

If delegation is expressed using can say$_0$, the delegatee is prevented from re-delegating to any other principal. The rule system enforces this by requiring that the delegatee's statement of the fact relies solely on assertions said by U_2 herself:

$$\frac{\mathcal{P} \vdash U_1 \text{ says : } U_2 \text{ can say}_0\, \mathit{fact} \qquad \mathcal{P}_{U_2} \vdash U_2 \text{ says : } \mathit{fact}}{\mathcal{P} \vdash U_1 \text{ says : } \mathit{fact}}$$

where $\mathcal{P}_{U_2}$ is the set of assertions in $\mathcal{P}$ that are said by U_2.

Definition 2.1. The *answer* to a query e says : *fact* is the set of all substitutions θ (with domain restricted to variables occurring in the query) such that $\mathcal{P} \vdash \theta(e \text{ says : } \mathit{fact})$ is derivable. □

Principals offering services via the network control access to these by defining a local SecPAL policy. Users and other services can submit access requests, which may be accompanied by a set of supporting SecPAL credentials (or other security credentials that can be mapped to SecPAL credentials). When an access request is received, it is mapped to a corresponding SecPAL query, which is then evaluated against the local policy combined with the supporting credentials. Access is granted only if the answer to the query is non-empty.

3 Abductive SecPAL Evaluation

Access is denied if the query statement is not provable, indicated by an empty answer to the corresponding query. The attempted proof may fail for several reasons. It may be the case that the local policy simply does not permit the specific request, or the requester has not provided the correct set of supporting credentials. The latter case is common in federated environments where credentials issued by third parties are used for constructing delegation chains. This section presents an algorithm that not only decides if a set of submitted SecPAL credentials is sufficient for authorizing a request, but, in the case of access denial, also computes a complete specification of which missing credentials *would* be sufficient. The computation is entirely local, i.e., it does not require communication with the providers of the missing credentials. As such, it can be run *before* access is required, as a preparation step for credential gathering, which is described in Section 4.

On an abstract level, we reduce the problem of computing the specification of missing credentials to the problem of *abduction*. Abduction is a term coined by the philosopher Charles

Peirce in the late 19th century, who used it to describe how observations can be explained, given a set of rules known about the world. More specifically, abduction is the process of finding a set of facts that, together with the rules, explain the observation. For example, given the rule "whenever it rains, the grass is wet," the observation that the grass is wet could be explained by a hypothetical fact that it has rained. Abduction is thus dual to deduction, where from a given set of rules and facts, the expected observations (or conclusions) can be logically derived. Logic-based abduction has been extensively used in fault diagnosis, automated planning, and other AI applications [12]. Applying these concepts to authorization, rules correspond to the local policy, facts to submitted credentials, and observations to queries. In this framework, deduction then corresponds to deciding if access should be granted, and abduction to deciding which missing credentials would result in access being granted.

Datalog Translation The first step of the evaluation procedure translates the assertions (from the local policy and supporting credentials) and the query corresponding to the access request into *constrained Datalog*. For a detailed introduction to constrained Datalog, see e.g. [9, 18].

In the following, we write A, B, P, Q, R, ... for atoms (positive literals built from predicate symbols) and c for constraints. We use vector notation to denote a (possibly empty) list of atoms, e.g. $\vec{P}$, and write $Q_0 :: \vec{Q}$ for a predicate list with head element Q_0 and tail list $\vec{Q}$. An atom P is an *instance* of Q if there exists some substitution θ such that $P = \theta(Q)$.

In constrained Datalog, *clauses* are of the form $P \leftarrow \vec{P}, c$ where the atom P is the *head*, the (possibly empty) list of atoms $\vec{P}$ the *body*, and c the *constraint* of the clause. A *program* $\mathcal{C}$ is a finite set of clauses. A constrained Datalog *query* consists of an atom.

To emphasize the similarity between SecPAL and constrained Datalog, we define the semantics of the latter with a similar proof system, consisting of a single proof rule that resembles the first SecPAL proof rule in Section 2:

$$\frac{\mathcal{C} \vdash \gamma(P_i) \quad \text{for all } i \in \{1..k\}}{\mathcal{C} \vdash \gamma(P)}$$

provided that there exists a clause $\langle P \leftarrow \vec{P}, c\rangle$ in the program $\mathcal{C}$, γ is a ground variable substitution, and $\gamma(c)$ evaluates to true. As for SecPAL, we define the *answer* to a constrained Datalog query Q to be the set of all substitutions θ (with domain restricted to variables occurring in Q) such that $\mathcal{C} \vdash \theta(Q)$.

Given an atomic assertion e says : *fact* and an expression k which may either be a variable or 0 or ∞, let $[\![e \text{ says : } \mathit{fact}]\!]_k$ denote the Datalog atom where the predicate name is the string concatenation of all non-parameter strings in *fact*, and where its parameters are k, followed by e, followed by the collected expressions between these infix operators. For example, the expression $[\![x \text{ says : } \texttt{Bob} \text{ can say}_0\ y \text{ can read } z]\!]_\infty$ denotes the atom $\text{can\_say0\_can\_read}(\infty, x, \texttt{Bob}, y, z)$.

(1) An assertion U says fact_0 if $\mathit{fact}_1, ..., \mathit{fact}_k, c$ is translated into the clause

$$[\![U \text{ says } \mathit{fact}_0]\!]_x \leftarrow [\![U \text{ says } \mathit{fact}_1]\!]_x, ..., [\![U \text{ says } \mathit{fact}_k]\!]_x, c$$

where x is a variable (ranging over 0 and ∞) not occurring in the assertion. This translation step effectively implements the first proof rule from Section 2.

(2) If $fact_0$ is nested, it is of the form e_0 can say$_{K_0}$ ... e_{n-1} can say$_{K_{n-1}}$ $fact$, for some $n \geq 1$, where $fact$ is not nested. 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 $fact_0 = \hat{fact}_0$. For each $i \in \{1..n\}$, we add a clause

$$\begin{aligned}[\![U \text{ says } \hat{fact}_i]\!]_\infty \leftarrow & [\![x \text{ says } \hat{fact}_i]\!]_{K_{i-1}}, \\ & [\![U \text{ says } x \text{ can say}_{K_{i-1}} \hat{fact}_i]\!]_\infty\end{aligned}$$

where x is variable not occurring in the assertion. Note that if $K_{i-1} = 0$, the first body atom $[\![x$ says $\hat{fact}_i]\!]_0$ can only be proved using clauses from translation step (1), because all clauses from step (2) have ∞ as their subscript in the head. This effectively implements the second and third proof rules.

Given a SecPAL policy $\mathcal{P}$, let $[\![\mathcal{P}]\!]$ denote the constrained Datalog program obtained by translating each assertion in $\mathcal{P}$ as described above. Similarly, given a SecPAL query e says $fact$, let $[\![e \text{ says } fact]\!] = [\![e \text{ says } fact]\!]_\infty$ be the *translation* of the query.

Proposition 3.1. The answer to a SecPAL query q with respect to a policy $\mathcal{P}$ is equal to the answer of the constrained Datalog query $[\![q]\!]$ with respect to $[\![\mathcal{P}]\!]$. □

Constrained Tabled Abduction Having translated SecPAL into constrained Datalog, we can reduce the problem of finding a complete specification of missing credentials to an abduction problem in constrained Datalog. More precisely, we now have to find sets of atoms, each of which would make the query provable if added to the program.

Our starting point is the deductive evaluation procedure described in [2], a backward-chaining resolution algorithm that attempts to unify the current goal atom against the head of some program clause. If unification succeeds, the body atoms of that clause are added as new goals. The algorithm makes use of *tabling*, or *memoization*, to guarantee deduction termination and to avoid proving goals that have already been proved [5, 19, 7]: a new proof branch is created for a goal A only if there is no existing branch for some goal A' of which A is an instance. If there already exists such a goal A', its existing answers, cached in its *answer table*, are reused as answers for A; furthermore, the goal A is suspended and put onto a *waiting table* for A', and resumed only when new answers for A' are found.

To find the atoms that are missing to complete a proof, one might think that the failed resolution proof graph would contain sufficient information. This is not the case: the points of failure in a failed proof graph show which missing atoms would have made the proof progress by one step, but not whether any such progress would eventually lead to a successful proof and which further missing atoms would be required. However, this observation suggests the basic idea for an abduction algorithm for translated SecPAL policies: during a proof, whenever an attempt to prove a goal fails, the corresponding atom is nevertheless assumed to be true (it is then said to be *abduced*) and the proof resumes from there. The algorithm keeps track of these assumptions, so that each subgoal can be associated with a set of abduced assertions its proof depended on.

The algorithm constructs a forest of proof trees. Each tree consists of a *root node*, intermediate *goal nodes*, and *answer nodes* as leaf nodes, defined as follows.

Definition 3.2 (Proof nodes)**.** A *root node* is of the form $\langle P \rangle$. A *goal node* is a quintuple of the form $\langle P; \vec{Q}; R; \vec{A}; c \rangle$. The atom P is the *index* of the goal node, $\vec{Q}$ are the *subgoals*, R is an

```
RESOLVE-CLAUSE($\langle P \rangle$)
01  $Ans(P) := \emptyset$;
02  foreach $(Q \leftarrow \vec{Q}, c) \in [\![\mathcal{P}]\!]$ do
03     if $nd = \mathsf{resolve}(\langle P; Q :: \vec{Q}; Q; [\,]; c\rangle, \langle P; [\,]; P; [\,]; \mathtt{True}\rangle)$
04        exists then
05          PROCESS-NODE($nd$);
06  if $P$ is abducible then
07     PROCESS-ANSWER($\langle P; [\,]; P; [P]; \mathtt{True}\rangle$)

PROCESS-ANSWER($nd$)
01  match $nd$ with $\langle P; [\,]; \_; \_; c\rangle$ in
02     if there is no $nd_0 \in Ans(P)$ such that $nd \preceq nd_0$ then
03        $Ans(P) := Ans(P) \cup \{nd\}$;
04        foreach $nd' \in Wait(P)$ do
05           if $nd'' = \mathsf{resolve}(nd', nd)$ exists then
06              PROCESS-NODE($nd''$)
```

```
PROCESS-NODE($nd$)
01  match $nd$ with $\langle P; \vec{Q}; \_; \_; c\rangle$ in
02   if $\vec{Q} = [\,]$ then
03     PROCESS-ANSWER($nd$)
04   else match $\vec{Q}$ with $Q_0 :: \_$ in
05     if there exists $Q_0' \in \mathsf{dom}(Ans)$
06          such that $Q_0$ is an instance of $Q_0'$ then
07       $Wait(Q_0') := Wait(Q_0') \cup \{nd\}$;
08       foreach $nd' \in Ans(Q_0')$ do
09         if $nd'' = \mathsf{resolve}(nd, nd')$ exists then
10            PROCESS-NODE($nd''$)
11     else
12       $Wait(Q_0) := \{nd\}$;
13       RESOLVE-CLAUSE($\langle Q_0 \rangle$)
```

Figure 1: Deductive evaluation algorithm with abductive extension

instance of P called the *partial answer*, $\vec{A}$ are the *abductive assumptions*, and c is the *constraint* of the goal node. A goal node with an empty list of subgoals is an *answer node*. □

Starting from some root node $\langle P \rangle$, resolution with program clauses produces goal nodes with index P. As the subgoals $\vec{Q}$ are processed one by one, new P-indexed goal nodes are created with the remaining subgoals and with increasingly instantiated variants of P as partial answer. A proof branch ends when no subgoals are left, i.e., in the case of an answer node.

An answer node $\langle P; [\,]; R; \vec{A}; c\rangle$ has the following property for all ground substitutions γ such that $\gamma(c)$ is true: if the set of abductive assumptions $\gamma(\vec{A})$ had been supplied together with $[\![\mathcal{P}]\!]$, a successful proof of $\gamma(R)$ (which is a ground instance of P) could have been constructed. The list $\vec{A}$ thus corresponds to a set of missing atomic assertions, constrained by c. In the degenerate case where $\vec{A}$ is empty, $\gamma(R)$ can be proved without any abductive assumptions, corresponding to access granted without any additional credentials.

Fig. 1 shows the pseudocode of the algorithm. Underscores denote distinct anonymous variables (and can be read as 'don't care'). The auxiliary function resolve and the subsumption relation $\preceq$ are defined below in Definitions 3.3 and 3.4. As in the standard tabled deductive algorithm, our abductive algorithm utilizes two initially empty tables (i.e., partial functions), Ans and $Wait$: $Ans(P)$ holds the set of answer nodes that have so far been found for the goal indexed by P, and $Wait(P)$ contains the set of goal nodes that are suspended and waiting for future answers to P.

The algorithm consists of three procedures that each take a proof node as input. The RESOLVE-CLAUSE procedure takes as input a root node $\langle P \rangle$ and creates a new proof tree for it by initializing an entry in the answer table (Line 1). It then proceeds by resolving P against the clauses in $[\![\mathcal{P}]\!]$ (Line $2-4$). The resolved clauses are processed further by PROCESS-NODE (Line 5). Additionally, RESOLVE-CLAUSE implements the abductive base case: it "invents" a trivial answer for P by simply assuming P to hold; this is tracked by adding P to the list of abductive assumptions of the new answer node, which is then further processed by PROCESS-ANSWER (Line 6,7).

PROCESS-NODE takes as input a goal node *nd* and first checks if it is an answer node, in which case it is further processed by PROCESS-ANSWER Line $1-3$). Otherwise, the leftmost subgoal Q_0 is chosen to be solved next (Line 4). If the answer table already contains an entry for some Q'_0 that is more general than Q_0 (Line 5,6), then the currently existing and future answers of Q'_0 are candidates for resolving against *nd* (Line $7-10$). Otherwise, a new root node is spawned for Q_0, whose proof tree should eventually provide answer nodes to be resolved against *nd* (Line 12,13).

PROCESS-ANSWER takes as input an answer node $\langle P; [\,]; \_; \_; c\rangle$ and adds it to the answers of P, if it is not subsumed by any already existing answer (Line $1-3$). Furthermore, it attempts to resolve the new answer against all suspended goal nodes waiting for it (Line $4-6$).

The algorithm differs from the standard deductive evaluation algorithm in three respects. Firstly, resolving a goal node against an answer node requires the assumptions and constraints from both nodes to be merged. Abductive answers may be non-ground and may contain constraints, so these have to be combined as well. Formally, the abductive resolve function is defined as follows.

Definition 3.3 (Resolution)**.** Let $\mathsf{mgu}(A,B)$ denote a most general unifier of atoms A and B, if one exists, and be undefined otherwise. Let $nd_0 = \langle P_0; Q_0 :: \vec{Q}; R_0; \vec{A_0}; c_0\rangle$ be a goal node, and let $\langle P_1; [\,]; R_1; \vec{A_1}; c_1\rangle$ be a fresh renaming of an answer node nd_1. Then $\mathsf{resolve}(nd_0, nd_1)$ exists iff $\theta = \mathsf{mgu}(Q_0, R_1)$ is defined and $c = \theta(c_0 \wedge c_1)$ is satisfiable, and its value is $\langle P_0; \theta(\vec{Q}); \theta(R_0); \theta(\vec{A_0}) \cup \theta(\vec{A_1}); c\rangle$. $\square$

Secondly, the procedure RESOLVE-CLAUSE is extended by an if-clause (Line 6,7), which creates a new abductive answer for the subgoal P if P is *abducible*, i.e., if it is amongst the atoms that are allowed as an assumption in an abductive answer. For example, in delegation policies, one is usually only interested in abducing atomic assertions said by someone other than the local authority; in this case, only atoms corresponding to such assertions would be deemed abducible. This abducibility filter effectively prunes the space of possible abductive proofs the algorithm will consider.

The third difference is in PROCESS-ANSWER where a newly found answer is added to the answer table only if it is not *subsumed* by an existing answer. Intuitively, an abductive answer is subsumed by a second answer if providing the missing atoms specified by the former also always provides those specified by the latter. Having already found the second answer, it is therefore not desirable to add the first answer, which is harder to satisfy and thus less useful. For example, an answer node with abductive assumptions

$\{[\![\texttt{Charlie}\ \mathsf{says}:\ \texttt{Doris}\ \mathsf{is\ a}\ \texttt{user}]\!],$
$[\![\texttt{Charlie}\ \mathsf{says}:\ \texttt{Doris}\ \mathsf{is\ a}\ \texttt{admin}]\!]\}$

and constraint `True` is subsumed by an answer node with abductive assumption $\{[\![\texttt{Charlie}\ \mathsf{says}:\ \texttt{Doris}\ \mathsf{is\ a}\ r]\!]\}$ and the constraint $\langle r\ \mathsf{matches}\ \texttt{adm*}\rangle$. Clearly, any set of atoms satisfying the first set also covers the second set.

Formally, subsumption between answer nodes is defined as follows.

Definition 3.4 (Node subsumption)**.** Let $nd_0 = \langle P_0; [\,]; R_0; \vec{A_0}; c_0\rangle$ and nd_1 be answer nodes, and let $\langle P_1; [\,]; R_1; \vec{A_1}; c_1\rangle$ be a fresh renaming of nd_1. Then nd_0 is *subsumed* by nd_1 (we write $nd_0 \preceq nd_1$) iff $|\vec{A_0}| \geq |\vec{A_1}|$ and there exists a substitution θ such that $R_0 = \theta(R_1)$ and $\vec{A_0} \supseteq \theta(\vec{A_1})$ and $\sigma(\theta(c_1))$ is true for all substitutions σ for which $\sigma(c_0)$ is true. $\square$

Running the Algorithm The algorithm takes as input a query q and a (possibly empty) set of supporting credentials $\mathcal{A}_{sup}$, that is combined with the service's local policy $\mathcal{P}_{loc}$ to form the input policy $\mathcal{P} = \mathcal{P}_{loc} \cup \mathcal{A}_{sup}$. The entry point is a call to RESOLVE-CLAUSE($\langle Q\rangle$), where $Q = [\![q]\!]$. On termination, $\mathit{Ans}(Q)$ contains a complete set of answers of the form $\langle Q; [\,]; R; \vec{A}; c\rangle$, where R is a (not necessarily ground) instance of Q. Such an answer can be interpreted as follows: if some ground instantiation (satisfying the constraint c) of the atoms in $\vec{A}$ had been in the original set of clauses $[\![\mathcal{P}]\!]$ (or had been derivable from that set), then R, under the same ground instantiation, could have been proven.

Any atom A occurring in the abductive proof forest can be translated back into the corresponding atomic SecPAL assertion $\|A\|$ by inverting the function $[\![\_]\!]_k$ (and dropping k). For example, if $A = \mathsf{can\_say0\_can\_read}(\infty, x, \mathtt{Bob}, y, z)$ then $\|A\| = x\ \mathsf{says}:\ \mathtt{Bob}\ \mathsf{can\ say}_0\ y\ \mathsf{can\ read}\ z$.

The algorithm returns a set of *templates* of the form

$$\langle \alpha; \mathcal{A}_{req}; \mathcal{A}_{acq}; c\rangle,$$

one for each answer node $\langle Q; [\,]; R; \{A_1, ..., A_n\}; c\rangle \in \mathit{Ans}(Q)$, where $\alpha = \|R\|$, $\mathcal{A}_{req} = \{\|A_1\|, ..., \|A_n\|\}$ specifies the *requirements*, and $\mathcal{A}_{acq} = \mathcal{A}_{sup}$ is the set of already *acquired credentials*, equal to the set of supporting credentials submitted as part of the input.

Definition 3.5. A set of credentials $\mathcal{A}$ *satisfies* $\langle \alpha; \mathcal{A}_{req}; \mathcal{A}_{acq}; c\rangle$ where $\mathcal{A}_{req} = \{\alpha_1, ..., \alpha_n\}$ iff there exists a ground substitution γ such that $\mathcal{A} \vdash \gamma(\alpha_i)$ (for all $i = 1...n$) and $\gamma(c)$ is true. $\square$

The main correctness property of the algorithm is that any set of credentials that satisfies one of the returned templates is sufficient for supporting the original access request. Therefore, the template set is a complete specification of the missing credentials:

Proposition 3.6. Let $\langle \alpha; \mathcal{A}_{req}; \mathcal{A}_{acq}; c\rangle$ be one of the return values from the algorithm (with local policy $\mathcal{P}_{loc}$ and query q). Then α is an instance of q. Furthermore, for all sets of credentials $\mathcal{A}$ that satisfy the template, $\mathcal{P}_{loc} \cup \mathcal{A}_{acq} \cup \mathcal{A} \vdash \gamma(\alpha)$ for some ground substitution γ. $\square$

The next section shows how the returned template set is used to encode the state of a distributed protocol which attempts to collect sets of satisfying credentials along a path of credential providers.

4 Credential Gathering Protocol

Based on the local abductive algorithm described above, we now present a protocol for the distributed collection of credentials prior to an access request. The protocol does not require any communication or other involvement of the resource guard from the time after the abductive answer has been returned, up until the time when the user requests access. When access is requested, the collected credentials are pushed to the resource guard, which can then, again completely locally, verify that the requested access is permitted. This protocol therefore decouples the distributed task of collecting credentials (which obviously requires communication, albeit not with the resource guard) from the local authorization task.

Overview The initial setting is as follows: a user U_{ini} intends to start a workflow which will eventually require access by some U_{acc} (which may be identical to U_{ini}) to some resource on service U_{srv} (the resource guard). The access by U_{acc} will take place at some future time T_{acc} called *access time*. U_{ini} does not know what supporting credentials are required by the policy at U_{srv} for this access. Therefore, at some point in time $T_{abd} < T_{acc}$ called *abduction time*, U_{ini} contacts U_{srv} in order to receive a complete specification for credentials required for the proposed access request. If the specification is empty, early failure is reported back to U_{ini}. If the specification is non-empty, U_{ini} initiates an automated process which visits a number of *credential providers* in turn, each of which may provide either stored credentials or new credentials issued on behalf of some provider-specific set of principals. In practice, credential providers may include U_{ini}'s local store, public directory services, firewalled security token servers, etc.

Any distributed credential gathering protocol must involve such a set of providers to be consulted for missing credentials. While previous credential retrieval protocols require the existence of one or more providers which can directly communicate with all other providers, the protocol described in this section assumes only that each credential provider is able to decide which provider should be consulted next and to communicate with that next provider. The protocol is agnostic as to the method used by each provider to make this decision. (The tradeoffs of this generalization are discussed in Section 8.) In practice, the next credential provider in the path may depend on the network topology, the application workflow, and information on where credentials are stored (e.g. always with the issuer, always with the subject, type-based [17], or policy-based [10, 4]).

At time step $T_1 > T_{abd}$, U_{ini} sends the credential specification (from U_{srv}) to some credential provider C_1. Based on this specification, C_1 collects matching local credentials that it is willing to disclose and generates matching credentials that it is willing to issue. C_1 also decides on the credential provider next on the path, C_2. The specification, together with the credentials, is then sent to C_2 at time T_2. This is repeated at each step of the protocol, until the last credential provider C_N is reached at step $T_N < T_{acc}$ (for some $N \geq 1$), after which either all required credentials have been successfully collected, or else the protocol reports failure.

Detailed Description At time step T_{abd}, U_{ini} requests from U_{srv} a complete specification of credentials required for the future resource access by U_{acc}. This specification is the result of running the abduction algorithm from Section 3. Recall that the algorithm takes as input a policy $\mathcal{P}$ and a query q_{abd}. In this case, $\mathcal{P}$ consists of U_{srv}'s local policy together with a (possibly empty) set $\mathcal{A}_0$ of supporting credentials submitted by U_{ini}. The query q_{abd} is the one associated with the access request at T_{acc} and may be provided, for example, manually by U_{ini}, by some piece of task-specific software, or by the service U_{srv} when some exposed API is called by U_{ini}. At time step T_{abd}, some of the values occurring in the actual access query q_{acc} (run at time T_{acc}) may not yet be known, therefore the query q_{abd} given to U_{srv} at T_{abd} may contain variables in

```
PROCESS-TEMPLATE-SET($\mathcal{T}$)
$\mathcal{T}' := \emptyset$;
foreach $\langle \alpha; \mathcal{A}_{req}; \mathcal{A}_{acq}; c \rangle \in \mathcal{T}$ do
    foreach $\langle \mathcal{A}; \theta; c' \rangle \in \mathsf{creds}_{C_i}(\mathcal{A}_{req}, c)$ do
        $c'' := \theta(c) \wedge c'$;
        $\mathcal{A}'_{req} := \theta(\mathcal{A}_{req}) \setminus \mathcal{A}$;
        $\mathcal{F} := \mathsf{instFacts}(\mathsf{addInst}(\mathcal{A}) \cup \theta(\mathcal{A}_{req}))$;
        $\mathcal{A}'_{req} := \mathcal{A}'_{req} \setminus \mathsf{instAssrts}(\theta(\mathcal{A}_{req}))$;
        $\mathcal{A}'_{req} := \mathcal{A}'_{req} \cup \{\langle C_{i+1}\ \mathsf{says} : \mathit{fact} \rangle : \mathit{fact} \in \mathcal{F}\}$;
        $\mathcal{A}_{inst} := \{C_i\ \mathsf{says} :\ C_{i+1}\ \mathsf{can\ say}_\infty \mathit{fact} : \mathit{fact} \in \mathcal{F}\}$;
        $\mathcal{A}'_{acq} := \mathcal{A}_{acq} \cup \mathsf{issue}(\mathsf{addInst}(\mathcal{A})) \cup \mathsf{issue}(\mathcal{A}_{inst})$;
        $\mathcal{T}' := \mathcal{T}' \cup \{\langle \theta(\alpha); \mathcal{A}'_{req}; \mathcal{A}'_{acq}; c'' \rangle\}$;
send $\mathcal{T}'$ to $C_{i+1}$;
```

```
PROCESS-FINAL-TEMPLATE-SET($\mathcal{T}$)
foreach $\langle \alpha; \mathcal{A}_{req}; \mathcal{A}_{acq}; c \rangle \in \mathcal{T}$ do
    foreach $\langle \mathcal{A}; \theta; c' \rangle \in \mathsf{creds}_{C_i}(\mathcal{A}_{req}, c)$ do
        $c'' := \theta(c) \wedge c'$;
        $\mathcal{A}'_{req} := \theta(\mathcal{A}_{req}) \setminus \mathcal{A}$;
        if $\mathcal{A}'_{req} \setminus \mathsf{instAssrts}(\theta(\mathcal{A}_{req})) = \emptyset$ and $\exists\, \gamma$ such that
            ($\gamma(c'')$ is true and $\gamma(\alpha)$ is an instance of $q_{acc}$)
        then
            $\mathcal{F} := \mathsf{instFacts}(\mathsf{addInst}(\mathcal{A}) \cup \theta(\mathcal{A}_{req}))$;
            $\mathcal{A}_{inst} := \{C_i\ \mathsf{says} :\ \gamma(\mathit{fact}) : \mathit{fact} \in \mathcal{F}\}$;
            $\mathcal{A}_{res} := \mathcal{A}_{acq} \cup \mathsf{issue}(\mathsf{addInst}(\mathcal{A})) \cup \mathsf{issue}(\mathcal{A}_{inst})$;
            send $\mathcal{A}_{res}$ to $U_{acc}$;
            return;
report failure;
```

Figure 2: Processing template set information

places where q_{acc} has concrete values; more precisely, q_{abd} must be such that q_{acc} is an instance of q_{abd}.

Recall that the result of the abduction algorithm is a set of templates of the form $\langle \alpha; \mathcal{A}_{req}; \mathcal{A}_0; c \rangle$. If the set is empty, U_{ini} is notified that the future request will fail, no matter which additional credentials are provided. Otherwise the set is processed by C_1 (which may be identical to U_{ini}) and subsequently used to encode the state of the protocol.

At each time step T_i (for $i = 1, ..., N-1$), the credential provider C_i receives a template set $\mathcal{T}$ and executes the procedure PROCESS-TEMPLATE-SET($\mathcal{T}$) (Fig. 2), which attempts to partially satisfy as many templates as possible, and send it to the next credential provider. At the final time step T_N, C_N receives a template set $\mathcal{T}$ and executes PROCESS-FINAL-TEMPLATE-SET($\mathcal{T}$), which will finalize the supporting credential set, to be used for the access query q_{acc}.

The procedures in Fig. 2 make use of a number of auxiliary functions and procedures defined below.

Definition 4.1. Let $\mathcal{A}$ be a set of assertions, $\mathcal{A}_{atm}$ a set of possibly unsafe atomic assertions, θ a substitution, and c a constraint.

- $\mathsf{creds}_{C_i}(\mathcal{A}_{atm}, c)$ returns a set of triples $\langle \mathcal{A}'_{atm}; \theta; c' \rangle$ such that $\mathcal{A}'_{atm} \subseteq \theta(\mathcal{A}_{atm})$ and $\theta(c) \wedge c'$ is satisfiable. Furthermore, no fact of the form $\mathsf{inst}(\_, \_)$ occurs in $\mathcal{A}'_{atm}$.

- $\mathsf{addInst}(\mathcal{A}_{atm})$ is the set of assertions obtained by augmenting each $\alpha \in \mathcal{A}_{atm}$ with a conditional fact $\mathsf{inst}(\mathtt{hash}_x, x)$ for each distinct variable x occurring in α. The expression $\mathtt{hash}_x$ stands for a constant that is unique for every variable x and for this particular run of the protocol.

- $\mathsf{issue}(\mathcal{A})$ is a procedure that issues all assertions in $\mathcal{A}$, i.e., it creates signed credentials corresponding to those assertions (or retrieves existing credentials from the local store), and returns them.

- $\mathsf{instFacts}(\mathcal{A})$ is the set of (concluding or conditional) facts of the form $\mathsf{inst}(\_,\_)$ occurring in $\mathcal{A}$.
- $\mathsf{instAssrts}(\mathcal{A})$ is the set of assertions in $\mathcal{A}$ whose concluding facts are of the form $\mathsf{inst}(\_,\_)$.

□

The function creds_{C_i} is specific to each credential provider C_i. Given a constrained set of atomic assertions $(\mathcal{A}_{atm}, c)$ as input, it returns a set of triples $\langle \mathcal{A}'_{atm}; \theta; c' \rangle$. Each triple represents a set of credentials that the provider is willing and able to provide and that match a subset of the input specification (including the constraint c). These credentials may be from a local store, or freshly issued and may contain variables that are constrained by c'. They may be more instantiated than the input specification, hence the function also returns a substitution θ that partially maps the input specification onto the output.

The definition of creds_{C_i} is intentionally kept abstract and general to cover a wide range of possible implementations. In practice, C_i would decide according to a local issuance and disclosure policy which credentials are returned by creds_{C_i}. Any authorization mechanism, including SecPAL, could be used to implement such a policy; in fact, the policy decision may even involve human interaction (see Section 5). We only assume that the returned triples in creds_{C_i} contain the largest, least instantiated and least constrained assertion sets that conform to the local issuance and disclosure policy and partially match the input. For example, if the input is

$$(\{\texttt{Alice}\ \mathsf{says}:\ \texttt{Bob}\ \mathsf{can}\ \texttt{read}\ f,$$
$$\texttt{Bob}\ \mathsf{says}:\ \texttt{Charlie}\ \mathsf{can}\ \texttt{read}\ f\},\ \texttt{True})$$

and the provider's policy allows the disclosure of the first assertion in that set without further constraints, then it should also return it without instantiating the variable f to a more concrete value than necessary. If the provider returned an assertion with f bound to some concrete value, then the shared variable f in the remaining second assertion would also be bound to the same value, and subsequent credential providers may not be willing or able to provide a credential with that particular value for f. The protocol thus attempts to defer the instantiation of variables to the latest possible step, when C_N, the final provider in the path, has been reached.

This requirement introduces two problems. Firstly, the returned atomic assertions $\mathcal{A}'_{atm}$ may be unsafe (safe atomic assertions must not contain variables) and hence cannot be directly used within a SecPAL evaluation. Secondly, it is generally not in C_i's interest to issue a blanket assertion with uninstantiated variables; rather, it should be made sure that the variables will be bound to concrete values by the end of the protocol run, and that these values can only be chosen by credential providers down the path of this particular protocol run (provided that downstream providers are trusted by upstream).

Our solution to both of these problems is to guard each variable x occurring in any unsafe atomic assertion in $\mathcal{A}'_{atm}$ with a conditional fact $\mathsf{inst}(\texttt{hash}_x, x)$. This makes the assertion safe, because now all variables in the concluding fact also occur in a conditional fact. The function $\mathsf{addInst}$ is responsible for adding these conditional facts to the assertions returned by creds_{C_i} before they are actually issued by issue and added to the acquired credentials.

To address the second problem, C_i also delegates authority over the fact $\mathsf{inst}(\texttt{hash}_x, x)$ to C_{i+1} and adds a new requirement that C_{i+1} should instantiate the fact. Both the delegation and the requirement are handed down the path, so it is only when C_N is reached that concrete values for the uninstantiated variables are chosen and all outstanding inst facts issued. The details of this process are described in the following.

The purpose of PROCESS-TEMPLATE-SET($\mathcal{T}$) is to partially satisfy the templates in $\mathcal{T}$ using locally stored or freshly issued credentials which can then be removed from the set of requirements. We assume that when the procedure starts, C_i knows the identity of, and can communicate with, C_{i+1}.

First, an empty template set $\mathcal{T}'$ is initialized which acts as an accumulator for the new templates to be sent to the next credential provider in the path, C_{i+1} (Line 1). The procedure then loops through all triples $\langle \mathcal{A}; \theta; c' \rangle$ returned by creds_{C_i} that match any template in $\mathcal{T}$ (Lines 2,3). The purpose of the code inside the loop is to construct a new template to be added to $\mathcal{T}'$. The constraint c'' of this new template is the conjunction of the original constraint c (renamed by θ) and c' (Line 4). As a first step towards constructing the new set $\mathcal{A}'_{req}$ of requirements, $\mathcal{A}$ is removed from the original requirements (Line 5) and in exchange issued and added to the new set of acquired credentials (augmented by inst-conditions, Line 10). All original inst-requirements (which, by construction, are of the form C_i $\mathsf{says}:$ $\mathsf{inst}(\mathtt{hash}_x, x)$ for some x) are removed as well (Line 7) and replaced by identical assertions said by C_{i+1}. Similar inst-requirements are also added for each inst-condition in $\mathsf{addInst}(\mathcal{A})$. This finalizes the new set of requirements (Lines 8). Finally, C_{i+1} must also be given authority over these inst-requirements; the corresponding delegation credentials are issued and added to the set of acquired credentials (Lines 9,10). In essence, Lines $7-9$ implement the process of deferring instantiation of unsafe variables in $\mathcal{A}$. The new template is added to $\mathcal{T}'$ (Line 11), and at the end of the loop, $\mathcal{T}'$ is sent to C_{i+1} at time step T_{i+1}.

Each application of PROCESS-TEMPLATE-SET conserves the original property from Proposition 3.6, namely that any set of credentials satisfying a template in $\mathcal{T}$ will be a sufficient set of supporting credentials for an instance of the original query. At time step T_N, when the final credential provider C_N is reached (and we assume that C_N is aware of this fact), C_N executes PROCESS-FINAL-TEMPLATE-SET($\mathcal{T}$). We assume that at this point C_N knows the identity of and is able to communicate with U_{acc} (in most cases, C_N and U_{acc} are in fact identical). Furthermore, we assume that C_N knows the final access query q_{acc}.

PROCESS-FINAL-TEMPLATE-SET($\mathcal{T}$) also starts by partially satisfying the templates in $\mathcal{T}$ (Lines $1-4$). However, the goal now is not to produce a new template set, but to find one template which can be fully satisfied. This must be a template with requirements $\mathcal{A}_{req}$ which, after removal of $\mathcal{A}$ (Line 4), only contains inst-requirements (Line 5). Moreover, a ground variable substitution γ has to be found that satisfies the constraint c''. It must also be ensured that the resulting instance $\gamma(\alpha)$ of the original query q_{abd} is an instance of the actual access query q_{acc} made by U_{acc} at time step T_{acc} (Line 6). If these conditions are met, C_N can instantiate all inst-requirements using γ (Lines 8,9) and assemble the final set of acquired credentials $\mathcal{A}_{res}$ (Line 10) that is then sent to U_{acc} (Line 11). If the conditions are not met by any of the templates, the protocol fails.

Due to the invariance conserved by the protocol, the resulting set of credentials $\mathcal{A}_{res}$ is guaranteed to be a sufficient set of supporting credentials for the access query q_{acc} at time T_{acc}, granted that U_{srv}'s local policy has not changed in the meantime.

5 EHR Scenario

This section illustrates the abductive credential gathering protocol in the context of a simple scenario based on electronic health records (EHR). In this scenario, clinician Alice wishes to access patient Bob's sensitive data on the EHR server which holds patient-identifiable health data of all patients across a community. Alice initiates the credential gathering protocol prior to her access, to make sure that she will possess all required credentials when she needs them.

EHR policy The EHR service's policy states that access to a patient y's sensitive data is granted to a principal x if x is a clinician, x is treating y, and y has given consent to this access. The policy also requires that the validity time span of the consent is contained in the time span of the clinical relationship.

$\texttt{EHR}$ says : x can access y's data if
  x is a $\texttt{clinician}$,
  x is treating y (from t_1 until t_2),
  x has y's consent (from t_3 until t_4)
where $t_1 \leq t_3 \land t_4 \leq t_2$

EHR delegates authority over role membership definitions (expressed by facts of the form $\langle e_1$ is a $e_2\rangle$) to the National Health Service (NHS). Thus if the NHS says that a principal is e.g. a clinician or a hospital, EHR will say it as well. As clinical relationships (expressed by $\langle e_1$ is treating e_2 (from e_3 until e_4)$\rangle$) are not managed centrally, EHR also delegates this task to individual hospitals. Similarly, patient consent (expressed by $\langle e_2$ has e_1's consent (from e_3 until e_4)$\rangle$) is not managed by the EHR either, but by a separate patient health portal (PP) at which patients can, among other actions, register their consent for other people to access their sensitive data. EHR therefore delegates authority over consent facts to PP but requires that the validity time span be at most one year.

$\texttt{EHR}$ says : $\texttt{NHS}$ can say$_0$ x is a r

$\texttt{EHR}$ says : x can say$_0$ y is treating z (from t_1 until t_2) if
  x is a $\texttt{hospital}$,
  y is a $\texttt{clinician}$

$\texttt{EHR}$ says : $\texttt{PP}$ can say$_0$
    y has x's consent (from t_1 until t_2)
where $t_2 - t_1 \leq 365\ \texttt{days}$

Template Set Generation In this scenario, the initiating party and the accessing party are identical: $U_{ini} = U_{acc} = \texttt{Alice}$. The protocol starts by initiating an abductive query on the EHR service. The EHR service allows all atomic assertions to be abducible apart from those issued by EHR itself. This definition of abducibility is useful in the common situation where the principal performing the abduction has complete local knowledge about all self-issued credentials.

Alice submits the abductive query

$q = \texttt{EHR}$ says : $\texttt{Alice}$ can access $\texttt{Bob}$'s data together with her NHS-issued clinician credential $\langle$NHS says : Alice is a clinician$\rangle$. The answer is a template set containing one template $\langle q; \mathcal{A}_{req}; \mathcal{A}_{acq}; c \rangle\}$ where $\mathcal{A}_{acq} = \{$NHS says : Alice is a clinician$\}$, and $\mathcal{A}_{req}$ consists of

NHS says : x is a hospital

x says : Alice is treating Bob (from u_1 until u_2)

PP says : Alice has Bob's consent (from u_3 until u_4)

The constraint c is equal to $u_1 \leq u_3 \ \wedge u_4 \leq u_2 \ \wedge u_4 - u_3 \leq 365$ days.

Since the answer is not empty (which would mean that the access is not supported no matter which additional credentials were provided), and the missing-credential specification $\mathcal{A}_{req}$ is not empty (which would mean that Alice already possess all necessary credentials), the protocol proceeds by gathering credentials matching $\mathcal{A}_{req}$ and the constraint c.

Credential Gathering Alice is treating Bob in a local hospital whose credential providing service (HOSP) is behind a firewall, and can thus be directly accessed only by staff. In particular, it cannot be accessed by EHR, so server-side pull-based approaches to credential gathering are not applicable.

Alice forwards the returned template set to C_1 = HOSP which executes PROCESS-TEMPLATE-SET. The hospital's credential disclosure policy allows the disclosure of the locally stored NHS-issued credential stating that HOSP is a hospital. Furthermore, since Alice has started treating Bob on the date 2008-10-07, with the therapy lasting six months, creds$_{\text{HOSP}}$ returns a triple $\langle \mathcal{A}; \theta; c' \rangle$, where $\mathcal{A}$ is the set

$\{$NHS says : HOSP is a hospital,

HOSP says : Alice is treating Bob (from v_1 until v_2)$\}$,

θ is the substitution $[u_1 \mapsto v_1,\ u_2 \mapsto v_2]$ and c' the constraint 2008-10-07 $\leq v_1 \ \wedge v_2 \leq$ 2009-04-06. This gives rise to a new template set $\mathcal{T}'$ containing a single template $\langle q; \mathcal{A}'_{req}; \mathcal{A}'_{acq}; c'' \rangle$. The new set of acquired credentials $\mathcal{A}'_{acq}$ consists of $\mathcal{A}_{acq}$ unioned with

NHS says : HOSP is a hospital

HOSP says : Alice is treating Bob (from v_1 until v_2) if
  inst(hash$_{v_1}$, v_1),
  inst(hash$_{v_2}$, v_2)

HOSP says : PP can say$_\infty$ inst(hash$_{v_1}$, v_1)

HOSP says : PP can say$_\infty$ inst(hash$_{v_2}$, v_2)

The new set of requirements $\mathcal{A}'_{req}$ consists of

PP says : Alice has Bob's consent (from u_3 until u_4)
PP says : inst(hash$_{v_1}$, v_1)
PP says : inst(hash$_{v_2}$, v_2)

The new constraint c'' is equal to $\theta(c) \wedge c'$, hence $c'' = v_1 \leq u_3 \wedge u_4 \leq v_2 \wedge u_4 - u_3 \leq 365\ \texttt{days} \wedge \texttt{2008-10-07} \leq v_1 \wedge v_2 \leq \texttt{2009-04-06}$.

The new template set is sent to PP, which, being the last credential provider in the path, executes PROCESS-FINAL-TEMPLATE-SET. Assuming that Bob has given consent for Alice to access his sensitive data without specifying restrictions on the time span, $\mathsf{creds}_{\texttt{PP}}$ returns a triple containing $\{\texttt{PP}\ \mathsf{says}: \texttt{Alice}\ \mathsf{has}\ \texttt{Bob}\text{'s}\ \mathsf{consent\ (from}\ w_1\ \mathsf{until}\ w_2)\}$, the substitution $[u_3 \mapsto w_1,\ u_4 \mapsto w_2]$, and the constraint $\texttt{True}$. In the case where Bob has not given consent yet, the execution of $\mathsf{creds}_{\texttt{PP}}$ may involve sending a notification to Bob and waiting for him to give or deny consent. Again, pull-based approaches do not cope well with such situations where credentials are not immediately available, or where not all parties are online simultaneously.

Having satisfied the only requirement in $\mathcal{A}'_{req}$ that does not involve inst, PROCESS-FINAL-TEMPLATE-SET proceeds by attempting to find any ground variable assignment γ that satisfies the constraint. One such solution gives rise to the final set of acquired credentials $\mathcal{A}_{res}$ consisting of $\mathcal{A}'_{acq}$ unioned with

$\texttt{PP}$ says : $\texttt{Alice}$ has $\texttt{Bob}$'s consent (from $\texttt{2008-10-07}$
until $\texttt{2008-11-06}$)

$\texttt{PP}$ says : $\mathsf{inst}(\texttt{hash}_{v_1}, \texttt{2008-10-07})$

$\texttt{PP}$ says : $\mathsf{inst}(\texttt{hash}_{v_2}, \texttt{2008-11-06})$

These are sent back to Alice who can eventually use them to support her access query q.

Alternatively, Alice could also have submitted an abductive query with the patient parameter left uninstantiated: $\texttt{EHR}$ says : $\texttt{Alice}$ can access x's data. The template set resulting from this query could then have been reused by Alice for future, similar accesses to patients' sensitive data.

6 Implementation

Much of the system described in this paper has been implemented as an extension to the SecPAL research prototype. SecPAL's logic engine was extended to support the abductive algorithm described in Section 3. A grid computation system was then constructed in order to prototype and test the credential gathering algorithm described in Section 4. The system included an FTP service, secured using SecPAL, and a matching abductive query service with access to the FTP service's policy. A client-side application was then built which interfaces with a compute cluster scheduler (also secured using SecPAL) in order to schedule a compute job. In our test scenario, the client application acts as the initiator of the credential gathering process for a two-level delegation; first to the scheduler and then to the compute node. The compute node then uses the gathered credentials to access the FTP service to retrieve data for the computation and later to publish results for pickup. RSA cryptography is used throughout the system for signing credentials and identifying principals (by public keys). Communication in the system uses SOAP and XML serialization for credentials and template sets.

Each provider in our prototype distributed system involved in the credential gathering process is equipped with a local credential store and an issuance mechanism. The local creds functions are implemented by searching the local store for matching credentials, and issuing any credentials for which the provider has authority. In a real implementation, credentials should be returned based on a disclosure and issuance policy (e.g., as in Cassandra [4]).

Aggressive pruning of the proof search space helps to reduce running times of the abduction algorithm dramatically. The subsumption check (Definition 3.4) is one important place where such pruning occurs. But there is a tradeoff between the time spent checking subsumption between nodes (in order to dispose of redundant proof branches) and the time spent computing redundant answers. In the presence of complex constraints such as regular expressions, a complete, accurate subsumption check would be both expensive and hard to implement. Therefore, our implementation opts for an approximate subsumption check that uses a number of fast heuristics (the details of which are beyond the scope of this paper).

Another useful technique for reducing the search space is the use of the abducibility check in the RESOLVE-CLAUSE procedure (Line 6). This check defines the class of atomic assertions that the algorithm may insert into the set of abductive assumptions. Commonly it is useful to exclude assertions said by the resource guard, based on the premise that the resource guard already has complete local knowledge of any such assertions. Additionally, our implementation defines assertions involving $\mathsf{can\ say}_\infty$ to be non-abducible because the recursive nature of $\mathsf{can\ say}_\infty$ causes the search space to grow rapidly. In general, policies using depth-limited delegation (with $\mathsf{can\ say}_0$) tend to behave better under abduction than policies using infinite depth-delegation.

A preliminary scalability test was run on a set of scenarios deliberately constructed to produce an exponentially increasing number of answers. The parameters and computation times are given in Fig. 3. The underlying policy delegates authority over a number of roles (R) to a number of certificate authorities (CA). The policy requires that a principal possess all the roles in order to access some specific resource, resulting in CA^R possible ways this query could be satisfied ("Number of Answers" in the table). In addition to this basic policy, the test context was populated with policy for 200 irrelevant resources, as well as role membership assertions for 200 irrelevant principals, in order to test if they have any significant impact on evaluation time. The data indicates that the running time of the abduction algorithm scales roughly quadratically with the number of possible ways to build a proof for the given query.

7 Related Work

Previous work on credential gathering has focused on server-side pull methods, which are not always applicable if communication cost is high, or credential providers are unknown or unavailable to the resource guard. This section briefly reviews these works.

QCM [8] and Cassandra [4] are two example of such languages that facilitate on-demand credential retrieval during the authorization proof. In QCM, credential providers work either in online or in offline signing mode. In the former, providers create and sign requested credentials on the fly; in the latter, they return only cached credentials. (Our definition of creds abstracts away from this distinction.) If a provider is unavailable at access time, QCM and Cassandra return approximate answers. QCM's expressiveness is equivalent to non-recursive Datalog. SD3 [10] extends QCM with recursion and has a similar expressiveness to Cassandra's. In SD3, credential providers can return intensional answers (policy rules) as opposed to simple facts

CA	Roles (*R*)	Number of Answers	Time (s)
1	1	1	0.01
1	2	1	0.02
1	3	1	0.02
1	4	1	0.05
2	1	2	0.02
2	2	4	0.07
2	3	8	0.19
2	4	16	0.62
3	1	3	0.02
3	2	9	0.16
3	3	27	0.88
3	4	81	6.36
4	1	4	0.03
4	2	16	0.34
4	3	64	3.33
4	4	256	47.06

Figure 3: Abduction scalability test results under Windows Server 2003 on an Intel Pentium 4 at 3.4 GHz with 1 GB of RAM.

[11]. In effect, the provider can tell the requester that the answer depends on certain other facts issued by other principals.

Our abstract protocol does not specify how to determine where a missing credential is stored. QCM and SD3 assume credentials to be always stored with the issuer. In Cassandra, facts can be tagged with any arbitrary location. In RT [17], a type system on role names that constrains the storage locations of role credentials is used. Well-typedness of credentials guarantees that the location of any missing credential will be instantiated to concrete values at deduction time, so it can be fetched on-the-fly from that location.

Bauer et al. [1] present a technique for constructing a distributed authorization proof that is closely related to our abduction algorithm in Section 3. Their proof technique presupposes a resource guard that can communicate with all credential providers, but it delays expensive choice points (for example those requiring remote communication or human interaction) during the proof, so that these can be fetched collectively at the end of the proof. Their technique thus addresses one of the shortcomings of previous approaches, namely that providers may have to be queried multiple times and are asked for credentials that may not lead to a successful proof. Their techniques are applicable to authorization languages that are at most as expressive as Datalog without constraints. One of the chief challenges in the current paper was to design the algorithms in the context of a more expressive language that supports arbitrary constraints.

Abduction has been applied to access control in previous work. Becker and Nanz [3] propose the use of abduction for generating user-friendly explanations in the case of access denial and for debugging authorization policies. The algorithm presented in that paper only works with authorization logics based on Datalog (without constraints), and is therefore considerably simpler than the one presented in this paper. They also identify sufficient conditions for finiteness of abductive answers; it would be interesting to see if their results can be applied to our, more general, policy language.

Koshutanski and Massacci [13] develop a (centralized, on-demand) abduction-based framework for interactive access control in which the server requests missing credentials from the client if the ones submitted by the client are not sufficient for granting access. In their framework, clients can define a disclosure policy specifying which credentials they are willing to submit. This ties in with work on automated trust negotiation [21, 20], where credentials are exchanged in a multi-step disclosure process. The policies considered in their framework are written in Datalog without constraints and with variables ranging over a finite domain; this is too restrictive for decentralized authorization, where constraints and infinite-domain variables are vital. In contrast, our algorithm could be used with any authorization language that can be translated into constrained Datalog, including SD3 [10], Binder [6], Cassandra [4], RT [16, 15] and Delegation Logic [14].

8 Discussion

The credential gathering protocol described in this paper was developed to address some of the problems related to previous work in which credentials are fetched by the resource guard at authorization time, whenever the proof process requires a credential that is not locally stored. This centralized, on-demand approach is communication-intense (because credential providers may be visited multiple times during the proof), requires connectivity between the resource guard and relevant credential providers, and requires that all relevant credential providers to be simultaneously online at authorization time.

In contrast, our protocol makes very weak assumptions on connectivity and requires only that each credential provider can communicate with the next one along some linear path. It thus supports scenarios with distributed knowledge of credential locations, or where some credential providers are behind firewalls or other restrictions prevent a star-shaped connection topology. Because the protocol proceeds in one single pass without any participating party having to store any state, the credential gathering process can be done when some providers are intermittently offline. Communication overhead is minimized, thus the protocol is applicable in environments where each single credential fetch request may take a long time, for example when human interaction is involved.

However, these properties are achieved at the price of higher computational complexity and algorithms that are harder to implement. There are two sources of potential intractability within the protocol. Firstly, certain types of policies can cause abduction to be very expensive; future work may attempt to characterize such policies and to find alternative policy idioms that are "abduction-friendly". Secondly, if each credential provider has multiple ways of partially satisfying each template in the template set, the number of templates to be passed along the path can grow exponentially. In practice, this might not be a problem as credential gathering paths tend to be very short. Moreover, the template set processing can be optimized by making some additional assumptions. For example, if the final access query is known by intermediate credential providers, then a fully satisfiable template may be identified before the end of the path is reached. Also, under the assumption that credentials are always stored with the issuer, early failure is possible if a credential provider C finds that all templates in the template set require some credential issued by C which C is not able or willing to provide.

Another potential problem of our approach is the possibility for the requester (and the participating credential providers) to gain detailed knowledge about the resource guard's (U_{srv}) policy through the the template set. This is problematic if some part of the policy is considered to be confidential, but the same level of information could be gained by collaborating credential providers with the pull-based approach. A related problem is that any credential provider in the path gets to know all credentials that have been collected so far. But since neither PROCESS-TEMPLATE-SET nor PROCESS-FINAL-TEMPLATE-SET uses those credentials, it is possible to encrypt them with U_{srv}'s public key before they are sent to the next credential provider.

Conclusions We have presented a novel protocol for gathering authorization credentials in a distributed system, in the context of constrained authorization and delegation policies written in SecPAL. The protocol proceeds in two stages: in the first stage, abductive reasoning is used to distill a SecPAL policy and a generalized query into a template set representing a complete specification of missing credentials. In the second stage, the template set is passed along a path of credential providers each of which attempt to partially satisfy the templates by providing matching credentials. Instantiation of (possibly constrained) parameter variables in the set of acquired credentials is deferred and delegated down the path using SecPAL's can say delegation construct. The protocol is designed for environments in which the relying party cannot directly and simultaneously communicate with all credential providers, or where each remote credential request is costly.

Acknowledgements We thank Greg Fee, Jason Hogg, Larry Joy, Brian LaMacchia, and John Leen for their contributions to the SecPAL libraries and the implementation of the prototype for the decentralized credential gathering system. We are also grateful for the helpful comments from the anonymous reviewers of an earlier version of this paper.

References

[1] L. Bauer, S. Garriss, and M. K. Reiter. Efficient proving for practical distributed access-control systems. In *European Symposium on Research in Computer Security*, 2007.

[2] M. Y. Becker, C. Fournet, and A. D. Gordon. Design and semantics of a decentralized authorization language. In *IEEE Computer Security Foundations Symposium*, 2007.

[3] M. Y. Becker and S. Nanz. The role of abduction in declarative authorization policies. In *10th International Symposium on Practical Aspects of Declarative Languages*, Lecture Notes in Computer Science, pages 84–99. Springer, 2008.

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

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

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

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

[8] C. Gunter and T. Jim. Policy-directed certificate retrieval. *Software: Practice and Experience*, 30:1609–1640, 2000.

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

[10] 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.

[11] T. Jim and D. Suciu. Dynamically distributed query evaluation. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems*, pages 28–39. ACM Press, 2001.

[12] A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324, 1998.

[13] H. Koshutanski and F. Massacci. Interactive access control for web services. In *International Information Security Conference*, pages 151–166, 2004.

[14] 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.

[15] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Practical Aspects of Declarative Languages*, pages 58–73, 2003.

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

[17] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management. *Journal of Computer Security*, 11(1):35–86, 2003.

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

[19] H. Tamaki and T. Sato. OLD resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. Springer-Verlag, 1986.

[20] W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *IEEE International Workshop on Policies for Distributed Systems and Networks*, 2002.

[21] W. H. Winsborough, K. E. Seamons, and V. E. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume 1, 2000.