

Cassandra: Distributed Access Control Policies with Tunable Expressiveness

Moritz Y. Becker Peter Sewell
Computer Laboratory, University of Cambridge
JJ Thomson Avenue, Cambridge, United Kingdom
{moritz.becker, peter.sewell}@cl.cam.ac.uk

Abstract

We study the specification of access control policy in large-scale distributed systems. Our work on real-world policies has shown that standard policy idioms such as role hierarchy or role delegation occur in practice in many subtle variants. A policy specification language should therefore be able to express this variety of features smoothly, rather than add them as specific features in an ad hoc way, as is the case in many existing languages.

We present Cassandra, a role-based trust management system with an elegant and readable policy specification language based on Datalog with constraints. The expressiveness (and computational complexity) of the language can be adjusted by choosing an appropriate constraint domain. With just five special predicates, we can easily express a wide range of policies including role hierarchy, role delegation, separation of duties, cascading revocation, automatic credential discovery and trust negotiation. Cassandra has a formal semantics for query evaluation and for the access control enforcement engine. We use a goal-oriented distributed policy evaluation algorithm that is efficient and guarantees termination. Initial performance results for our prototype implementation have been promising.

1. Introduction

The emergence of wide-area network-based services poses new and challenging problems to security management. The networks in question are generally heterogeneous, decentralised and large-scale, with possibly millions of autonomous entities (which may be individuals, agents, organisations or other administrative domains) that wish to share their resources in a secure and controlled fashion. Collaborating entities may be mutual strangers at first, thus access control cannot be based on identity, as it is the case in traditional approaches.

In the *trust management* approach [5], authorisation is based on credentials, digitally signed certificates asserting

attributes about entities holding them. In systems supporting *trust negotiation* [19], peers establish trust between each other by exchanging sets of suitable credentials. A *policy specification language* is used to define a system's security policy, a set of rules specifying the security goals in a high-level language. This approach separates policy from implementation, simplifies security administration and facilitates policy evolution.

The diversity of emerging applications with widely differing security requirements has led to the development of a variety of increasingly expressive policy specification languages (e.g. [5, 6, 9, 11, 14, 13, 12, 7]). Existing ones are extended to accommodate more complex policies. For example, the role-based trust management language RT_0 [14] was extended to RT_1 to handle parameterised roles, and to RT^T to express separation of duties [13]. Another extension of RT , RT_1^C [12], provides constructs for limiting the range of role parameters using constraints. However, adding constructs to a language in an ad hoc fashion to increase its expressiveness has several disadvantages. Firstly, it is unlikely that the extension will cover all policies of interest; secondly, the semantics and implementations of the language have to be changed; thirdly, languages with many constructs are harder to understand and to reason about; and lastly, policy evaluation usually becomes computationally more expensive with increasing expressiveness (in some cases, the language is even Turing-complete).

We have designed a trust management system, Cassandra, in which the expressiveness of the policy specification language can be adjusted by selecting an appropriate *constraint domain*. The advantage of this approach is that the expressiveness (and hence the computational complexity) can be chosen depending on the requirements of the application, and can easily be changed without having to change the language semantics. In our prototype implementation of Cassandra, a constraint domain is a separate module that can be plugged into the policy evaluation engine. We have identified a condition on constraint domains, *constraint compactness*, which ensures that policy evaluation is decidable and guaranteed to terminate.

By factoring out the constraint domain, the language

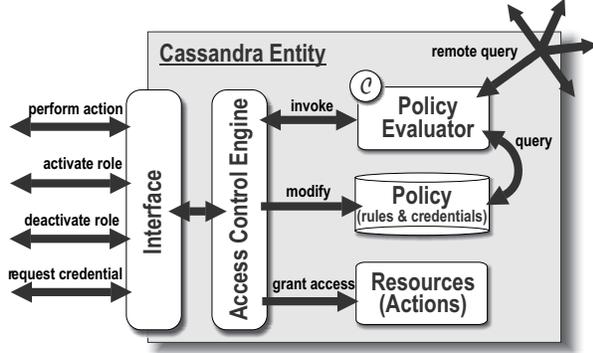


Figure 1. Cassandra components.

syntax and semantics are kept small and simple. In particular, Cassandra has no explicit provisions for standard policy idioms such as role hierarchy, separation of duties or delegation; instead, it is truly policy-neutral in that it can encode such idioms (and many variants). Its expressiveness suffices for policies found in highly complex real-world applications; this has been shown by our work on a large-scale security policy for a national electronic health record system [3].

In §2 we give an informal overview of Cassandra’s policy specification language. Unlike most other systems, Cassandra not only formally specifies the policy language but also the access control semantics governing the dynamic behaviour of an entire Cassandra network. This operational semantics is described in §3. §4 shows how examples of standard policies, including role validity periods, role hierarchy, separation of duties, role delegation and trust negotiation policies, can be expressed in Cassandra. The policy specification language and semantics are formally defined and an algorithm for policy evaluation is given in §5. §6 briefly discusses our case study on security policies for a national electronic health record system. We also discuss our prototype implementation and preliminary experimental results. Finally we discuss related work and conclude.

2. Policy specification overview

Cassandra is a trust management system allowing a potentially large network of entities to share their resources under well-defined restrictions, specified by local access control policies, even if they are mutual strangers. Every entity runs its own copy of a Cassandra service, which acts as a protective layer around the resources. Figure 1 shows the internal components of a Cassandra service. Interaction with other entities is done via the interface that defines requests for performing an action (i.e. accessing a resource), activating and deactivating a role, and request-

ing a credential that can be used to support another request somewhere else. The *access control engine* handles the request by invoking the *policy evaluation engine*, which in turn queries the local Cassandra policy. The expressiveness of the policy specification language depends on the globally chosen constraint domain, \mathcal{C} , an independent module that is plugged into the policy evaluation engine. As policies can refer to policies of other entities, policy evaluation may trigger queries of remote policies (possibly the requester’s) over the network. The answer of the policy evaluation engine is used by the access control engine to decide whether the request is to be granted. As a result of a request, the local policy may be modified. For example, if a role is activated, this new fact is put into the policy; similarly, deactivation of roles causes facts to be removed from the policy.

Cassandra’s policy specification language is based on $\text{Datalog}_{\mathcal{C}}$, a generic extension of negation-free Datalog (Prolog without function symbols) where the expressiveness can be tuned by varying the constraint domain parameter \mathcal{C} [10]. A $\text{Datalog}_{\mathcal{C}}$ rule is of the form

$$p_0(\vec{e}_0) \leftarrow p_1(\vec{e}_1), \dots, p_n(\vec{e}_n), c$$

where the p_i are predicate names and the \vec{e}_i are (possibly empty) expression tuples (that may contain variables) matching the parameter types of the predicate. $p_0(\vec{e}_0)$ is the *head* of the rule, and the sequence of predicates on the right hand side of the arrow is the *body* of the rule; c is a *constraint* on the parameters occurring in the rest of the rule. Intuitively, to deduce the head of a rule, all body predicates must be deducible in such a way that the constraint is also satisfied. A set of $\text{Datalog}_{\mathcal{C}}$ rules can then be interpreted as the deductive closure of the set.

The constraint of a rule, c , is a formula from some fixed *constraint domain* \mathcal{C} , a language of first order formulae containing at least true, false and the identity predicate “=” between \mathcal{C} -expressions (variables, entities and possibly other constructs). It must be closed under variable renaming, conjunction (\wedge) and disjunction (\vee). Furthermore, it must be equipped with an interpretation that defines when formulae are satisfied.

The expressiveness of $\text{Datalog}_{\mathcal{C}}$ depends on the chosen constraint domain \mathcal{C} . For example, the least expressive constraint domain is the one where the only atomic constraints are equalities between variables and constants. Choosing this trivial constraint domain reduces the expressiveness of the language to standard Datalog or Horn clauses without function symbols. More powerful constraint domains often include boolean, arithmetic and set constraints, and make use of more complex expressions such as tuples, set expressions and (side-effect free) function applications (e.g. to access the current time). The computational complexity of evaluating $\text{Datalog}_{\mathcal{C}}$ programs increases with expressiveness: with set constraints it is already possible to encode

the Hamiltonian cycle problem, and thus all NP-complete problems. Care must be taken not to choose a constraint domain that is too expressive as this can result in programs in which queries are undecidable. We will later introduce the notion of *constraint compactness* to restrict constraint domains to those that guarantee termination of queries.

In Cassandra, access control is role-based, and roles, as well as actions, are parameterised. Role-based access control (RBAC) [17, 8] was initially introduced to simplify security administration of large enterprises. In the context of distributed trust management, roles can more generally be used as a representation of authenticated subject attributes in decentralised access control [13]. Formally, a *role* is a typed role name applied to an expression (that may contain variables) of a matching type, e.g. `Manager(Sales-dept)`. Similarly, an *action* is an action name applied to an expression, e.g. `Read-file(file)`. For the remainder of the paper, variables will be written in small letters and italics (e.g. *file*), generic constants in italics but capitalised (e.g. some entity *E*), and concrete constants in typewriter font (e.g. `Sales-dept`).

Policies are specified by rules defining predicates that govern access control decisions: `permits` defines who can perform which action; `canActivate` specifies who can activate which roles (and thus implicitly defines the role membership relation); `hasActivated` specifies who is currently active in which role; `canDeactivate` specifies who can revoke which role; `isDeactivated` is used to define automatically triggered role revocation; and finally, `canReqCred` rules specify the conditions to be satisfied before the service is willing to issue and disclose a credential. User-defined auxiliary predicates are also allowed.

In the trust management approach, access control decisions are based on credentials asserting properties about the holders. In Cassandra, the properties asserted by credentials are (constrained) predicates. Therefore, in order to satisfy a predicate in a rule body, either the predicate can be deduced from the local policy or it is asserted by a foreign credential issued and signed by some other entity. Such credentials are either already stored locally, or are submitted to the service, or automatically fetched by the service from some other entity. To put constraints on the issuer and the storage location of credentials, each Cassandra predicate has an *issuer* and a *location* (constant or variable) parameter, and is written $loc@iss.p(\vec{e})$. For example, `Alice@UCam.canActivate(Alice, Student(Maths))` is a predicate asserting that Alice is a Maths student. If this predicate is part of a rule body, Cassandra can contact Alice over the network (unless this is Alice’s local policy) and request the corresponding credential issued by the University of Cambridge.

We will often write $iss.p(\vec{e})$ as shorthand for $E@iss.p(\vec{e})$ and $p(\vec{e})$ for $E@E.p(\vec{e})$, if *E* is clear from the

context. Intuitively, if a predicate $loc@iss.p(\vec{e})$ appears in the body of a rule in *E*’s policy, and *loc* is equal to *E*, it is deduced locally from *E*’s policy (if *iss* is not equal to *E*, this must be a foreign credential). If, however, *loc* is not equal to *E*, this means that the authority over the predicate is delegated to the remote entity *loc*, so *E* requests a credential $iss.p(\vec{e})$ from *loc* over the network. *loc* will allow this only if her local policy lets her deduce both `canReqCred(E, iss.p(\vec{e}))` and $iss.p(\vec{e})$. If these conditions are met, a credential containing $iss.p(\vec{e})$ (issued and signed by *iss*) is sent back to *E*. A more formal treatment of the language semantics is given in §5.1.

3. Access Control Semantics

Cassandra acts as a protective layer around the shared resources, allowing network access only through an interface. This interface defines requests for performing an action, activating a role, deactivating a role, and for requesting a credential. Incoming requests are checked by the access control engine against the local policy (Figure 1). Entities can support their requests by submitting credentials to the service; the service will then use the assertions in the credentials along with its own local policy to evaluate the query. Granting a request can have side-effects on policies, e.g. when a role is activated, a corresponding `hasActivated` credential rule is added to the policy.

We have formally specified the operational semantics of the access control engine by a labelled transition system where the labels are the requests and the transitions are between sets of policies of all entities. Due to lack of space, we will only give a brief overview of the request definitions.

Performing an action. Suppose the requester *E* attempts to perform the (parameterised) action *A* on *S*’s Cassandra service. *E*’s request is granted if `permits(E, A)` is deducible from *S*’s policy (and submitted credentials).

Role activation. Suppose *E* attempts to activate the (parameterised) role *R* on *S*’s Cassandra service. The request is granted if the role has not already been activated and if `canActivate(E, R)` can be deduced from *S*’s policy (and submitted credentials). As a result of this transition, the corresponding `hasActivated` credential rule is added to *S*’s policy.

Role deactivation. Suppose *E* requests to deactivate *V*’s role *R* on *S*’s Cassandra service. The request is granted if *V* is really currently active in the role *R* and if `canDeactivate(E, V, R)` is deducible from *S*’s policy (and submitted credentials). Depending on the local policy rules, this deactivation may also trigger the deactivation of other role activations in *S*’s policy (local cascading deactivation). For this purpose, we need to compute the set of all

hasActivated credential rules in S 's policy for which a corresponding isDeactivated credential can be derived under the assumption isDeactivated(V, R). The role activations in this set are then removed from S 's policy.

Requesting Credentials. Suppose E requests the credential $I.p(\vec{x}) \leftarrow c$ (a digital certificate asserting $p(\vec{x}) \leftarrow c$, issued and signed by I) from S . S 's service first computes the answer to the query $\text{canReqCred}(E, I.p(\vec{x})) \leftarrow c$. The answer is a constraint c_0 restricting the values that \vec{x} can take.

If I and S are identical, the answer c_1 of the query $p(\vec{x}) \leftarrow c_0$ is computed, and, if c_1 is satisfiable, the new credential $S.p(\vec{x}) \leftarrow c_1$ is issued and sent to E . If I and S are different, this means that the requested credential is a foreign credential held by S , so it cannot be freshly issued and signed. In this case, S sends E all her credentials of the form $I.p(\vec{x}) \leftarrow c_2$ such that c_2 is at least as restrictive as c_0 .

4. Standard policies

Unlike other policy specification languages, Cassandra does not have special constructs for expressing standard policies such as role hierarchies, separation of duties or delegation. Indeed, we can show that Cassandra, equipped with a sufficiently powerful constraint domain, can express these policies in a concise and readable way. Having no constructs in the language for specific policy idioms not only keeps the language and its semantics small and simple; it also avoids the necessity of having to constantly extend the language. Furthermore, our work on policies for a national electronic health record infrastructure has shown that, in large-scale real-world applications, these “standard” policies occur in many variants and combinations with subtle but significant semantic differences [3]. Cassandra was designed in such a way that the whole range of policy variants can be expressed without additional features. It should be noted that Cassandra was designed specifically for authorisation policies; in particular, we do not deal with obligation policies specifying the automatic triggering of actions (as in [7]).

In the following, we show how standard policies can be written in Cassandra.

Role validity periods. In the following rule, a certified doctor (with certification issued at time t) is also member of the role $\text{Doc}()$ if t is at most one year ago. This is an example where the freshness requirement of a certification is set by the acceptor, not by the certificate issuer (as recommended in [16]). The chosen constraint domain must contain a (side-effect free) built-in function that returns the current time, and integer order constraints.

```
canActivate(x, Doc()) ←
  canActivate(x, CertDoc(t)),
  CurTime() - Years(1) ≤ t ≤ CurTime()
```

Auxiliary roles. Sometimes a role is used solely to express some property about its members and can be used without prior activation. In this rule, a logged-in user can read a file provided that the system can deduce she is the owner of that file. Ownership is here expressed with the auxiliary `Owner` role that need not be activated.

```
permits(x, Read(file)) ←
  hasActivated(x, Login()),
  canActivate(x, Owner(file))
```

Role hierarchy. In this variant of parameterised role hierarchy, members of a superior role (`Engineer` working in some department) are automatically also members of a more basic role (`Employee` working in the same department).

```
canActivate(x, Employee(dep)) ←
  canActivate(x, Engineer(dep))
```

Separation of duties. In this common example for separation of duties, a payment transaction requires two phases, initiation and authorisation, which have to be executed by two different people. The rule implements the dynamic and parameterised variant of separation of duties: an `Authoriser` of a payment must not have activated the `Init` role for the same payment. This restriction is implemented by the user-defined `countInitiators` predicate. Its definition is given by the second rule, an example of an *aggregate* rule. The `count{z}` aggregate operator counts how many different values of z satisfy the body. Therefore, the parameter n is 0 only if x has not activated the `Init` role for the same payment.

```
canActivate(x, Authoriser(payment)) ←
  countInitiators(n, x, payment), n = 0
countInitiators(count{z}, x, payment) ←
  hasActivated(z, Init(payment)), z = x
```

Role delegation. Here, an administrator can delegate her role to somebody else by activating the `DelegateAdm` role for the delegatee. The delegatee can then subsequently activate the administrator role. The first parameter of the administrator role specifies who the delegator was. The second parameter n is an integer for restricting the length of the delegation chain: the delegatee can activate the administrator role only with a “rank” n' that is strictly less than the delegator's rank n but must be at least 0. Setting the parameter to 1 for non-delegated administrators (i.e. those at the top of a delegation chain) amounts to non-transitive delegation. Removing the constraint on n in the second rule results in unbounded delegation chains.

```
canActivate(x, DelegateAdm(y, n)) ←
  hasActivated(x, Adm(z, n))
canActivate(y, Adm(x, n')) ←
  hasActivated(x, DelegateAdm(y, n)), 0 ≤ n' < n
```

With the following rule, the delegated role is automatically revoked if the delegation role of the delegator is deactivated.

$$\begin{aligned} \text{isDeactivated}(y, \text{Adm}(x, n')) \leftarrow \\ \text{isDeactivated}(x, \text{DelegateAdm}(y, n)) \end{aligned}$$

However, we need to specify who is allowed to deactivate a delegation role. In grant-dependent revocation (first rule below), only the delegator herself has this power. In grant-independent revocation (second rule below), every administrator (who has at least as high a rank as the delegator) can deactivate the delegation.

$$\begin{aligned} \text{canDeactivate}(x, z, \text{DelegateAdm}(y, n)) \leftarrow x = z \\ \text{canDeactivate}(x, z, \text{DelegateAdm}(y, n)) \leftarrow \\ \text{hasActivated}(x, \text{Adm}(w, n')), n \leq n' \end{aligned}$$

A rather paranoid policy may specify cascading revocation: if a delegated administrator is revoked from her role, all her delegation must also be revoked recursively.

$$\begin{aligned} \text{isDeactivated}(x, \text{DelegateAdm}(y, n)) \leftarrow \\ \text{isDeactivated}(z, \text{DelegateAdm}(x, n')) \end{aligned}$$

The trust management system Oasis [21] has a language construct for role appointment, a generalisation of role delegation. Our work on real-world policies suggests that variants of general appointment are indeed far more frequent than role delegation [3]. Appointment and other stateful policies can be expressed in Cassandra in a very similar way as shown above for delegation.

Automatic trust negotiation & credential discovery. Suppose the following rule is part of the policy of a server holding the electronic health records (EHR) for some part of the UK's population. To activate the doctor role, x must be a certified doctor in some health organisation org , and furthermore the organisation must be a certified health organisation. Both requirements must be satisfied in the form of credentials signed by some entity $auth$ belonging to a locally defined set of registration authorities.

$$\begin{aligned} \text{canActivate}(x, \text{Doc}(org)) \leftarrow \\ \text{auth.canActivate}(x, \text{CertDoc}(org)), \\ \text{org}@auth.\text{canActivate}(org, \text{CertHealthOrg}()), \\ \text{auth} \in \text{RegAuthorities}() \end{aligned}$$

In the rule above, there is no location prefix in front of the first body predicate, so the doctor certification credential is required to already be in the local policy or have been submitted by x together with the role activation request. No automatic credential requests are issued the credential is not found. On the other hand, there is a location prefix org in front of the second body predicate: the health organisation credential is automatically requested from org , or, more precisely, the entity the variable org stands for during actual evaluation. However, the health organisation (say, Addenbrooke's Hospital) will allow this retrieval request only if its `canReqCred` policy allows it. With the following rule, Addenbrooke's specifies that it is willing to reveal

its `CertHealthOrg` credential, signed by the registration authority of East England, to certified EHR servers.

$$\begin{aligned} \text{canReqCred}(x, y.\text{canActivate}(z, \text{CertHealthOrg}())) \leftarrow \\ x@auth.\text{canActivate}(x, \text{CertEHRServ}()), \\ y = \text{RegAuthEastEngland} \wedge z = \text{Addenbrookes}, \\ \text{auth} \in \text{RegAuthorities}() \end{aligned}$$

The $x@auth$ prefix specifies that the required credential must be signed by some registration authority and that it is to be retrieved automatically from x ; in this case, x will have been instantiated to be the EHR server. The EHR server will in turn have `canReqCred` policy rules specifying to whom its `CertEHRServ` credential may be disclosed. As this example shows, a simple request can trigger multiple phases of credential exchanges between two or more entities over the network until a sufficient level of mutual trust has been established.

5. Language semantics and evaluation

This section defines the syntax and semantics of Cassandra's policy specification language. We also describe a goal-oriented algorithm for evaluating policy queries that is sound and complete with respect to the language, and discuss a condition for guaranteed termination of query evaluation.

5.1. Language Semantics

Each entity E_{loc} on the network protects its resources with a (possibly empty) Cassandra *policy*, a finite set of Cassandra *policy rules* of the form

$$\begin{aligned} E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow \\ \text{loc}_1@iss_1.p_1(\vec{e}_1), \dots, \text{loc}_n@iss_n.p_n(\vec{e}_n), c. \end{aligned}$$

The *location* and the *issuer* of the rule, E_{loc} and E_{iss} , are entity constants, and the loc_i and iss_i are entities or entity typed variables. The $p_i(\vec{e}_i)$ are well-typed predicates, and c is a constraint from the globally chosen constraint domain \mathcal{C} .

A rule with empty body of the form

$$E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow c$$

is called a *credential rule* or just a *credential*. (These correspond to *facts* in Logic Programming.) If it is sent over the network, it can be thought of as a certificate asserting $p_0(\vec{e}_0)$, signed and issued by E_{iss} , and belonging to and stored at E_{loc} . The location and the issuer of a rule are usually identical; only in the case of a credential rule can they be different, as E_{loc} may hold a *foreign* credential signed by a different entity E_{iss} .

We will omit the prefix E_{loc} from a rule if it is clear from the context, and also E_{iss} , loc_i and iss_i if they are equal to E_{loc} .

Access control decisions are based on policy *queries* which have the same form as credentials: $E_{loc}@E_{iss}.p_0(\vec{e}_0) \leftarrow c$. The answer to a query is a set of constraints c_i such that $E_{iss}.p_0(\vec{e}_0) \leftarrow c \wedge c_i$ can be deduced from E_{loc} 's policy. For example, the query

$$\text{UCam@UCam.canActivate}(x, \text{Student}(\text{subj})) \leftarrow \\ \text{subj} = \text{Maths}$$

may return the constraints $\{x = \text{Alice}, x = \text{Bob}\}$, and the query

$$\text{UCam@UCam.canActivate}(x, \text{Student}(\text{subj})) \leftarrow \\ x = \text{Alice} \wedge \text{subj} = \text{Maths}$$

would simply return $\{\text{true}\}$.

The semantics of a policy is defined by the set of all credentials that can be deduced from it. To formally define the notion of deduction, we extend the notion of *consequence operator* known from constraint logic programming [18]. We construct a consequence operator $T_{\mathcal{P}}$, where \mathcal{P} is the finite union of the policies of all entities. Given a set of credentials \mathcal{I} (which we distinguish only up to variable renaming), $T_{\mathcal{P}}(\mathcal{I})$ returns the set of all credentials that can be deduced from \mathcal{I} and the policies in \mathcal{P} in one step.

The definition of $T_{\mathcal{P}}$ assumes the existence of two computable operations on \mathcal{C} -constraints, \exists^c and \Rightarrow^c . $\exists^c x. (c)$ computes the existential quantifier elimination of x and returns the set of conjuncts in the disjunctive normal form (DNF) of the result. If V is a set of variables, we also write $\exists^c_{-V}(c)$ for the set of conjuncts in the DNF of c , with all free variables apart from the ones in V existentially eliminated. (This is in effect a projection of c onto the variables V .)

\Rightarrow^c is a computable subsumption relation on \mathcal{C} -constraints: if $c_1 \Rightarrow^c c_2$ returns true then c_1 is *subsumed* by c_2 , i.e. all substitutions that satisfy c_1 also satisfy c_2 .

Then the consequence operator $T_{\mathcal{P}}(\mathcal{I})$ is defined to contain all credentials of the form $E_{loc}@E_{iss}.p(\vec{x}) \leftarrow c_0$ (for some entities E_{loc} , E_{iss}) if \mathcal{I} contains no other credential that already subsumes it: if $E_{loc}@E_{iss}.p(\vec{x}) \leftarrow c'_0 \in \mathcal{I}$ and $c_0 \Rightarrow^c c'_0$ then $c_0 = c'_0$; and furthermore, if there is some matching rule

$$E_{loc}@E_{iss}.p(\vec{x}) \leftarrow P_1, \dots, P_n, c$$

in \mathcal{P} (i.e. in the policy of E_{loc}) such that there is a constraint c_0 with the following property:

$c_0 \in \exists^c_{-\vec{x}}(c_1 \wedge \dots \wedge c_n)$, and c_0 is satisfiable, for some constraints c_1, \dots, c_n , such that each c_i is a contribution from P_i . We say c_i is a *contribution* from $P_i \equiv y_{loc}@y_{iss}.q(\vec{y})$ if one of the following two cases hold.

Either y_{loc} is taken to be local, so P_i has to be deduced from E_{loc} 's own local policy. This means that c_i must be

equal to some

$$(c'_i \wedge y_{loc}=E_{loc} \wedge y_{iss}=E'_{iss})$$

such that $E_{loc}@E'_{iss}.q(\vec{y}) \leftarrow c'_i$ is already in \mathcal{I} .

Alternatively, y_{loc} may refer to some remote entity $E'_{loc} \neq E_{loc}$, so P_i has to be deduced from E'_{loc} 's policy. As this amounts to a credential request and E'_{loc} 's credentials are protected by `canReqCred` rules, the corresponding `canReqCred` predicate must also be satisfied, as well as P_i itself. In this case, c_i is some constraint in

$$\exists^c x_e. (c'_i \wedge c''_i \wedge y_{loc}=E'_{loc} \wedge \\ y_{iss}=E'_{iss} \wedge x_e=E_{loc})$$

such that both credentials

$$E'_{loc}@E'_{loc}.canReqCred(x_e, y_{iss}.q(\vec{y})) \leftarrow c'_i \text{ and} \\ E'_{loc}@E'_{iss}.q(\vec{y}) \leftarrow c''_i \text{ are already in } \mathcal{I}.$$

The consequence operator $T_{\mathcal{P}}(\mathcal{I})$ is continuous on the powerset of credentials and thus has a unique least fixed-point $\bigcup_{n \geq 0} T_{\mathcal{P}}^n(\emptyset)$ which we call the fixed-point semantics of \mathcal{P} . It coincides with our intuitive notion of deductive closure of the policy rules.

Sometimes we need to know not only whether a predicate can be satisfied but also how often. For example, it is often necessary to know that nobody has activated a certain role, i.e. the corresponding `hasActivated` predicate can be satisfied 0 times. For these purposes, we define rules with *aggregation operators* [15]. (These require the constraint domain \mathcal{C} to contain equalities over set and integer constants and variables.) A Cassandra *aggregation rule* is of the form

$$E_{loc}@E_{loc}.p(\text{aggop}(x), \vec{y}) \leftarrow E_{loc}@E_{iss}.q(\vec{x}), c$$

where the aggregation operator `aggop` is either `group` or `count`. The predicate $q(\vec{x})$ is required to be one that can be satisfied with only finitely many different parameters on E_{loc} , and \vec{x} must contain x . If the operator is `group`, the first argument of p stands for the finite set of all different values of x such that the rule body can be satisfied. If the operator is `count`, it stands for the cardinality of that set. For example,

$$\text{getSetOfActiveDoctors}(\text{group}(x), \text{spcty}) \leftarrow \\ \text{hasActivated}(x, \text{Doctor}(\text{spcty}))$$

finds the set of all active doctors with specialty `spcty`.

5.2. Evaluation

Recall that the access control engine makes access control decisions by invoking the policy evaluation engine, which queries the local policy. We now describe the algorithms used in the policy evaluation engine.

In deductive databases, queries are usually evaluated against a model that is pre-computed with a bottom-up algorithm that, starting from basic facts, iteratively adds derived facts until the fixed-point semantics is reached.

This would not be an acceptable evaluation strategy for Cassandra: firstly, the constraints may contain (side-effect free) function calls that depend on the environment, for example for getting the current time, and therefore cannot be pre-computed; secondly, the fact that rule bodies can refer to remote predicates would require a distributed form of bottom-up evaluation which would be highly impractical; and thirdly, the model would have to be re-computed after every activation or deactivation of roles as role activation and deactivation modify policies.

The standard SLD top-down resolution algorithm known from Logic Programming (e.g. Prolog) is not suitable either as it may run into infinite loops even when the fixed-point semantics is finite. Instead, Cassandra uses a modified version of Toman’s memoing algorithm for evaluating constraint extensions of Datalog [18]. Based on SLG resolution, it combines advantages of both the top-down and the bottom-up approaches: it is goal-oriented and yet preserves the termination properties of the bottom-up algorithms by memoing (tabling) already seen subgoals and their answers. To solve a subgoal for which a table entry already exists, the algorithm uses the tabled answers as solutions; whenever new answers are added for the entry, they are automatically propagated to other waiting evaluation branches. If no relevant entry exists for the subgoal, a new table entry is created and populated. We have extended the algorithm in [18] to deal with goals referring to remote entities.

Suppose the query $E_{loc}@E_{iss}.p_0(\vec{x}_0) \leftarrow c_0$ is to be evaluated by the Cassandra service of E_{loc} . Evaluation is started by calling the Clause Resolution procedure on the query.

Clause Resolution. Find all policy rules with a matching head, i.e. of the form

$$E_{loc}@E_{iss}.p_0(\vec{x}_0) \leftarrow P_1, \dots, P_n, c_1.$$

For all such c_1 , compute $c_2 \equiv c_0 \wedge c_1$ if the result is satisfiable. If the rule body is non-empty ($n \geq 1$), call the Query Projection procedure on the list P_1, \dots, P_n, c_2 . Otherwise call the Answer Projection procedure on the combined constraint c_2 .

Query Projection. This procedure operates on a list of predicates P_1, \dots, P_n and a constraint c . Using the \exists^c operation, project the constraint onto the free variables of the first predicate P_1 in the list and compute the DNF constraint set. For all c_i from this set, call the Answer Propagation procedure on $P_1 \leftarrow c_i$, and the (possibly empty) list of remaining predicates, P_2, \dots, P_n .

Answer Propagation. This procedure operates on a subgoal $P \leftarrow c$, and a list of remaining predicates P_2, \dots, P_n . Check whether we have already encountered a query $P \leftarrow c'$ such that $c \Rightarrow^C c'$, in which case the current goal can be solved using answers from that query. For each already

existing answer d , combine it with the current constraint and call the Clause Resolution procedure on the remaining predicates in the list, or the Answer Projection procedure, if the remaining list is empty. We also need to store the information that this query waits for answers from the proof of $P \leftarrow c'$.

If, however, no such $P \leftarrow c'$ exists yet, we need to spawn a new query for $P \leftarrow c$ and wait for its answers. If the location of P is remote, a credential request is sent to the remote entity. The remote entity will then call its Query Projection procedure on the list containing $\text{canReqCred}(E_{loc}, P)$ and P with the constraint c .

Answer Projection. This procedure is called when the list of body predicates is empty. The remaining constraint is then projected onto the free variables of the query predicate. The resulting constraints are stored in the answers table and propagated to all queries currently waiting for such answers, and execution is resumed there. If the waiting party is a remote entity, the answers are sent to it over the network in the form of credentials. The remote entity will then invoke its Answer Projection procedure on these answers.

On exit, the table entry for the original query will be populated with all its answers. The algorithm is sound and complete with respect to the language semantics.

As in other database applications, we require query evaluation to always terminate. Clearly, if the chosen constraint domain \mathcal{C} is too expressive, it is possible to write policies and queries that are uncomputable. Often, the features that make it too expressive seem rather innocuous at first glance. For example, constraint domains with untyped tuple constructors or with negative gap-order constraints of the form $x - c < y$ (where c is a positive integer constant) enable the construction of undecidable policies.

Constraint compactness [18] is a sufficient condition on constraint domains to guarantee a finite and hence computable fixed-point semantics for any finite global policy set \mathcal{P} . A constraint domain \mathcal{C} is said to be constraint compact if any infinite set of \mathcal{C} -constraints in which only finitely many variables and constants occur has a finite subset subsuming the entire set, that is, for every constraint c in the infinite set there is a constraint c' in the finite set such that $c \Rightarrow^C c'$.

Unfortunately, constraint compactness severely restricts the expressiveness of the constraint language and is also often hard to prove. We use *static groundness analysis* [1] to restrict policies in such a way that variables occurring in specific constructs will always have been grounded (so a unique value can be deduced for each) by the time existential quantifier elimination is performed on them, given the query patterns from §3 (e.g. `canActivate` queries are always fully grounded), so these constructs can be ignored.

We also use static groundness analysis to ensure that the location prefix of body predicates becomes ground by the time we evaluate it: otherwise the evaluator would have to query many different entities (all, in the worst case), which is clearly unpractical.

6. Discussion

EHR case study. Cassandra’s design process was partially guided by our case study [3] on an access control policy for a national electronic health record (EHR) system. The background of the case study is the British National Health Service’s current plan to develop an electronic data spine that will contain “cradle-to-grave” medical data for all patients in England. The project is highly risky and challenging for several reasons: it is extremely large-scale with 100 million records and billions of accesses per year; the requirements are likely to change frequently, in particular those concerning access control; and it is inherently distributed with interacting health organisations, registration authorities and the data-spine. These challenges can best be met by a distributed trust management system that allows policies to be specified in a sufficiently expressive high-level language.

In our case study, we propose a distributed three-level infrastructure to cope with the large scale. Based on official specification documents, we have developed Cassandra policies for the entire infrastructure. Our proposed policies contain a total of 310 rules, define 58 parameterised roles and implement all the required access control rules.

The requirements are not only highly complex but also contain principles unseen in traditional access control models. For example, the policies need to handle explicit patient consent, third-party disclosure consent, individualised access decisions (e.g. a patient could prohibit access to record items concerning a certain medical subject to a specific doctor), appointment of agents acting on a patient’s behalf and workgroup-based access control (e.g. based on ward or consultant team membership).

One of the main lessons learnt from the case study is that standard policy idioms such as role appointment occur in many different variants. We thus had to design Cassandra in such a way that it could express all of these elegantly. Our approach was to identify the small number of underlying primitives concerning role membership, activation and deactivation, and to base the language solely on those. The distributed nature of the EHR policies also necessitated features for automatic credential discovery and credential protection (automatic trust negotiation).

For the case study, we devised a sufficiently expressive constraint domain containing tuple expressions and projections, disequalities, integer order inequalities, built-in functions to access state-dependent data and set inclusion

constraints[3]. It is constraint-compact and thus guarantees query termination, but its relatively high expressiveness still makes it possible in principle to write policies that are prohibitively expensive to evaluate. However, such policies do not seem to occur in practice, as the recursion depth is usually small and variables are instantiated to ground values early on.

Implementation and performance. A prototype of Cassandra has been implemented in OCaml. The code is factored into independent modules as depicted in Figure 1. In particular, constraint domain implementations can be plugged into the policy evaluation engine as separate modules, as long as they provide fundamental operations of projection, satisfiability and subsumption checking. We have implemented the constraint domain used for the EHR case study, including a type inference mechanism that allows us to omit explicit variable typing.

At the time of writing, role deactivation and credential requests and the static groundness analyser are still in the process of being implemented. Furthermore, the current prototype only simulates the distributed system, and issued credentials are implemented without encryption and public key signatures.

The prototype was tested with the policies from the EHR case study. The system behaved as expected and handled all requests, including the most complex ones, within fractions of a second. The preliminary results suggest that Cassandra is indeed suitable for large-scale real-world application. Of course, authoritative results can only be produced after completion of a more complete and optimised implementation and under more realistic settings; we have for example so far only tested the system with up to 10,000 patients [3].

Our experiments have highlighted another requirement for policy-based trust management systems that neither our nor existing systems currently fulfil: human users expect textual justifications of access control decisions, especially if their request is denied; they feel rather frustrated and helpless if the answer is simply “request denied”, especially if the policy is complex or unknown to the user. Such explanations could be collected from annotations of policy rules used during deduction. The problem is non-trivial as deduction proofs can be long and access denials can have many and far-reaching reasons. More worryingly, the textual justification may reveal more (and perhaps, sensitive) information than could have been deduced from the fact of request denial alone: consider, for example, a response such as “access denied because your daughter has prohibited you from accessing all her records with the subject ‘abortion’ ”.

Related work. A large amount of work has been done on security policy specification in a non-trust-management

context. For instance, Barker [2] uses constraint logic programming to encode RBAC policies in a non-distributed environment; as such, his approach does not deal with credentials, trust management and trust negotiation. Policy-Maker [5] introduced the trust management paradigm, and its successor, KeyNote [4] defined the first policy specification language. Since then, many other trust management systems have been proposed for policy specification and distributed access control (e.g. SPKI/SDSI [6], QCM [9], SD3 [11], RT [13], Oasis [21], Ponder[7]).

The Cassandra policy specification language was inspired by Oasis, a role-based trust management system in which Datalog-based rules specify which credentials are prerequisite for role activation and deactivation [21]. Oasis has a special construct for role appointment, which was introduced as a useful generalisation of the delegation mechanisms found in many other languages. Our case study supports the claim that role appointment (and its variants) is a very useful policy idiom. Oasis is the only other system we are aware of that supports cascading role revocation. Its revocation mechanism works even across the network between collaborating entities. This is implemented using a distributed event infrastructure. Another difference is that in Oasis, revocation is triggered whenever a specified subset of the role activation prerequisites ceases to hold. In contrast, role deactivations in Cassandra are allowed to be triggered by conditions that have nothing to do with the role activation prerequisites. Oasis does not deal with automatic credential discovery and trust negotiation. It also does not possess a full formal semantics and does not guarantee termination of queries.

The RT family of role-based trust management languages [13] bears some similarities to our system. In RT, the Datalog-based rules, or credentials, as they are called, specify only the role membership relation: either directly, by role hierarchy, by (direct or attribute-based) delegation of authority, or any combination of these. The subjects of the rule head and the body conditions are implicitly the same, which is sufficient to express delegation but not convenient for appointment policies. In RT's youngest offspring, RT_1^C [12], rules are translated into Datalog_C. Constraints are used only to define a range on each role parameter; constraints between two parameters are not permitted in order to keep policies more comprehensible and to guarantee tractability. We find that a more liberal use of constraints is useful and necessary, as our EHR policy shows, and can still be efficient in practice. RT roles are prefixed with the issuing entity, just like Cassandra's predicates are, but do not specify the location where a matching credential may be found. RT solves this by statically specifying for each role name whether credentials defining such roles are stored with the issuer or the subject. Our EHR policy has rules in which predicates have locations different

from both issuer and the subject entity. A distinctive feature of the RT framework is that RT credentials contain a link to a so-called Application Domain Specification Document (ADSD) that defines a common vocabulary (types of role parameters, natural language descriptions of role names etc.) for collaborating entities.

SD3 is another Datalog-based trust management system [11]. Similar to Cassandra, SD3 predicates can be prefixed with an issuer (a public key), thereby delegating authority of predicate definition to that key. A predicate can further be tagged with an IP address which is used to refer to a remote policy. SD3 is a very general system that does not specify any access control meaning for any predicates and can be viewed as Cassandra without constraints, roles and access control semantics. SD3 passes the proof tree from its highly optimised policy evaluation engine through a simple and small proof checker to reduce the size of its trusted computing base. This would be a technique that could also be applied to Cassandra.

The problem of trust negotiation has been addressed in [19], where various different negotiation strategies (which, when and in which order credentials are disclosed) are discussed. Their Credential Access Policy (CAP) corresponds to Cassandra's `canReqCred` rules specifying the prerequisites for credential disclosure. Cassandra's uniform treatment of rules during evaluation gives us trust negotiation almost "for free", with a negotiation strategy similar to their "Parsimonious Strategy". It has been pointed out that this strategy can leak information about possession of credentials without actually disclosing them. The "Eager Strategy" does not have this problem but is less efficient. [20] prevents the problem by adding another policy protection layer. [22] argue that entities should be given the freedom to choose their own negotiation policy. They identify a large family of strategies that are mutually compatible.

Conclusions and future work. We have developed a trust management system, Cassandra, with a role-based policy specification language in which the expressiveness can be tuned according to need by choosing an appropriate constraint domain. Apart from management of role permissions, activations and (cascading) deactivations, the system also uniformly provides flexible automatic credential retrieval and automatic trust negotiation. With the constraint domain we devised for the EHR case study, Cassandra's expressiveness surpasses that of existing systems while preserving a strong termination property. The policy language is small, simple and devoid of any redundant constructs such as delegation or hierarchies and yet it can express a wide variety of policies. Cassandra, including the language, the access control engine and the goal-oriented distributed policy evaluation algorithm, is fully and formally specified and thus amenable to formal reasoning.

We plan to use Cassandra’s formal framework to prove security properties about specific policies. Along the same lines, we wish to formalise a low-level model of Cassandra that specifies the underlying network protocols, the public key infrastructure and the design of certificates. We will also investigate possibilities for making answers to requests more descriptive and user-friendly without leaking sensitive information.

To gather more reliable test results, we need to build a complete prototype that is truly distributed and uses digital certificates for sending credentials over the network. We hope to improve efficiency by using a standard relational database for policy rule lookups. Such an implementation will enable us to test real-world policies in a more realistic setting, with millions of role activations and entities that interact via an unreliable network.

Acknowledgments We acknowledge support from a Gates Cambridge Scholarship (Becker), a Royal Society University Research Fellowship (Sewell), EPSRC grant GRN24872, and EC FET-GC project IST-2001-33234 PEPITO. The authors thank Arne Heizmann for corrections and comments. We also thank the reviewers for their valuable comments.

References

- [1] N. Baker and H. Sondergaard. Definiteness analysis for CLP(R). In *Australian Computer Science Conference*, pages 321–332, 1993.
- [2] S. Barker and P. J. Stuckey. Flexible access control policy specification with constraint logic programming. *ACM Transactions on Information and System Security*, 6(4):501–546, 2003.
- [3] M. Y. Becker and P. Sewell. Cassandra: Flexible trust management, applied to electronic health records. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, June 2004. To appear.
- [4] M. Blaze, J. Feigenbaum, and A. D. Keromytis. KeyNote: Trust management for public-key infrastructures (position paper). *Lecture Notes in Computer Science*, 1550:59–63, 1999.
- [5] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, pages 164–173, 1996.
- [6] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.
- [7] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Policy Workshop*, 2001.
- [8] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. 4, (3):224–274, 2001.
- [9] C. A. Gunter and T. Jim. Policy-directed certificate retrieval. *Software - Practice and Experience*, 30(15):1609–1640, 2000.
- [10] J. Jaffar and M. J. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
- [11] 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.
- [12] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, pages 58–73, 2003.
- [13] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust management framework. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.
- [14] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *ACM Conference on Computer and Communications Security*, pages 156–165, 2001.
- [15] P. Revesz. *Introduction to constraint databases*. Springer Verlag, 2002.
- [16] R. L. Rivest. Can we eliminate certificate revocations lists? In *Financial Cryptography*, pages 178–183, 1998.
- [17] R. Sandhu. Rationale for the RBAC96 family of access control models. In *Proceedings of the 1st ACM Workshop on Role-Based Access Control*, 1997.
- [18] D. Toman. Memoing evaluation for constraint extensions of datalog. *Constraints*, 2(3/4):337–359, 1997.
- [19] W. Winsborough, K. Seamons, and V. Jones. Automated trust negotiation. In *DARPA Information Survivability Conference and Exposition*, volume 1, pages 88–102, 2000.
- [20] W. H. Winsborough and N. Li. Towards practical automated trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 92–103, 2002.
- [21] W. Yao, K. Moody, and J. Bacon. A model of OASIS role-based access control and its support of active security. *ACM Transactions on Information and System Security*, 5(4), 2002.
- [22] T. Yu, M. Winslett, and K. E. Seamons. Supporting structured credentials and sensitive policies through interoperable strategies for automated trust negotiation. *ACM Transactions on Information and System Security*, 6(1):1–42, 2003.