Implementation and Performance Analysis of the Role-Based Trust Management System, RTC

Tyler L. Hobbs, William H. Winsborough

▶ To cite this version:

Tyler L. Hobbs, William H. Winsborough. Implementation and Performance Analysis of the Role-Based Trust Management System, RTC. 4th IFIP WG 11.11 International on Trust Management (TM), Jun 2010, Morioka, Japan. pp.184-199, 10.1007/978-3-642-13446-3_13 . hal-01061327

HAL Id: hal-01061327
https://hal.inria.fr/hal-01061327

Submitted on 24 Nov 2017

Implementation and Performance Analysis of the Role-based Trust Management System, RT^C

Tyler L. Hobbs[1] and William H. Winsborough[2]

[1] University of Texas at Austin, thobbs@cs.utexas.edu
[2] University of Texas at San Antonio, wwinsborough@acm.org

Abstract. We present representations and algorithms for the implementation of RT^C, a role-based trust management language, and announce an open-source implementation available to the public. We also design and perform large-scale performance tests on policies closely modeled after possible applications of *RT* in the real world. These tests aim to determine the viability of *RT* as an authorization solution for large and potentially complex policies in a decentralized environment; the results of the tests are analyzed to identify what policy characteristics most strongly affect the performance of *RT* and develop strategies to achieve the rapid response times required in real-world authorization systems.

1 Introduction

The term "trust management" (TM) has been used in a variety of ways over the years. Today it is often taken to refer to semi-automated techniques for estimating the likely trustworthiness of individuals, organizations, and software agents. This paper focuses instead on the classical notion of TM introduced by Blaze, Feigenbaum and Lacy [1], which focuses on placing into the hands of individuals and organizations security management decisions that are appropriate to their expertise and to their exposure as stakeholders. This has the effect of decentralizing authority, which serves to place decisions in the hands of experts and interested parties.

Several TM systems have received a great deal of attention in the literature [1–8]. The aim of the current work is not to advance the underlying theory of TM, but rather to analyze the adequacy of one rich TM system as an authorization framework in highly decentralized and distributed environments. We have developed a Java-based system implementing RT^C, a modern TM system. RT^C uses a variant of the classical notion of role in which roles are owned by principals that have control over how the membership of their roles are defined. These roles can be used to express any property or characteristic of principals that role membership is intended by the role owner to assert. In particular, the roles can take parameters, such as the members date of birth or the path to a subdirectory to which the member has authorized access.

In RT^C, extensive use of delegation of authority is used, so that, for instance, experts on one characteristic of principals are not expected to be experts on all characteristics. For instance, a university might own roles that capture the characteristic of being enrolled as a student, while an accrediting board might own roles that capture the characteristic of being an accredited university. Someone wishing to define a role of their own such that the role's members are students at an accredited university might delegate to the accrediting board authority to define which principals are universities and to those universities, authority to define which principals are students.

RT^C is a member of the RT family of languages [6, 9, 10]. These languages introduce authority management abstractions that are given precise semantics via translation to the well understood subset of first-order predicate calculus, Datalog. RT^C in particular is translated into Datalog with constraint systems that admit efficient evaluation [11]. Constraints are logical formulas that serve to specify valuations over variables appearing as parameters in RT^C roles. For instance, constraints would be helpful in our university student example if the aim is to define a role whose members graduated in the last 10 years.

The current paper introduces techniques for the implementation of RT^C, including constraint representation and sketch an algorithm that performs the most interesting operation on them, namely conjunction. We have validated these techniques by building a distributed implementation, which is available to the public under the open source BSD license; all code and related documentation is available for public use at `http://sourceforge.net/projects/rtcredential`. This represents the first of the two present contributions.

The second contribution is a comprehensive investigation of the performance of our implementation. We investigate scenarios in which large numbers of credentials must be retrieved from widely distributed repositories, as well as assess the considerable improvements that can be obtained when credential caching is exploited. We do this in a variety of scenarios that we believe are representative of environments in which RT^C would be most helpful. The thesis we aim to support is that TM languages such as RT^C represent an important component in security solutions in the ever-widening need for decentralized authorization policy management.

The remainder of the paper is organized as follows: Section 2 gives a brief summary of RT^C [9,12]. Section 3 describes the RT Credential Toolkit. Section 4 presents and analyzes performance results for RT^C on large, real world scenarios and analyzes the results, and Section 5 concludes.

2 Brief Overview of RT^C

A main feature of RT^C is its ability to use parameterized roles with constraints [12]. Parameters allow us to convey information about principals that could not be easily represented by role membership alone, such as values in a large domain. This in turn enables policy to define authorization with much finer granularity than when constraints are not supported. For instance, in the university student example, constraints easily capture the requirement that graduation must have occurred within the last 10 years. Without constraints, each of the

10 years has to be treated as a separate case. In general, constraints greatly facilitate giving concise definitions of roles that involve parameters that range over large sets of possible values.

The aim of an RT^C [9,12] policy is to assign *principals* (also called entities) to *roles*. A principal may be a user, an organization, a virtual organization (VO), or a software or hardware agent. A role takes the form $A.r(x_1, \cdots, x_k)$ in which A is an principal, $r(x_1, \cdots, x_k)$ is called a *rolename*, and $x_1, \cdots, x_k$ are sorted[3] variables. We call r a *role identifier*, and intuitively it resembles a predicate over the owner, role member, and $x_1, \cdots, x_k$. We use capital roman letters A, B, D to range over principals and r, possibly with subscripts, to range over role identifiers.

A policy in RT^C is given by a set of statements, which are described just below. We index variables x_i^j occurring in a statement so that j indicates the role the variable occurs in and i indicates the parameter position with that role. We denote vectors of variables by using bold face, so $\boldsymbol{x}^j$ denotes the vector of variables appearing as parameters to the jth role in the statement. All variables occurring in a given statement are distinct from one another.

A *valuation* is a mapping from variables to values in the carrier appropriate to their sort. In the treatment given here, *primitive constraints* are predicates applied to variables that are given an intended interpretation over the specified carriers. For instance, if two distinct variables x_i^j and x_ℓ^k are intended to assume the same value, a primitive constraint $x_i^j = x_\ell^k$ is used to express this. Different constraint systems include different carriers and different primitive constraints (*interpreted* predicates). A *constraint* is a conjunction of primitive constraints, possibly with existential quantifiers. A constraint is satisfiable if there exists a valuation that makes each primitive constraint true. A constraint is said to be *over* any set of variables that includes the variables that occur free in it.

There are four types of policy statements in RT^C, each of which includes a constraint, denoted by $\psi(\boldsymbol{x})$, in which $\boldsymbol{x}$ is the list of variables occurring in the remainder of the statement:

- *Type-1:* $A.r(\boldsymbol{x}^0) \longleftarrow B; \psi(\boldsymbol{x}^0)$
  This statement defines B to be a member of A's role $r(\boldsymbol{x}^0)$ under all valuations that satisfy $\psi(\boldsymbol{x}^0)$.
- *Type-2:* $A.r_1(\boldsymbol{x}^0) \longleftarrow B.r_2(\boldsymbol{x}^1); \psi(\boldsymbol{x}^0, \boldsymbol{x}^1)$
  This statement defines $A.r_1(\boldsymbol{x}^0)$ to include all members of $B.r_2(\boldsymbol{x}^1)$
- *Type-3:* $A.r(\boldsymbol{x}^0) \longleftarrow A.r_1(\boldsymbol{x}^1).r_2(\boldsymbol{x}^2); \psi(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2)$
  The body of this statement is referred to as a *linked role.* Under each valuation that satisfies $\psi(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2)$, the statement says that for each B in $A.r_1(\boldsymbol{x}^1)$, $B.r_2(\boldsymbol{x}^2)$ is a subset of $A.r(\boldsymbol{x}^0)$.
- *Type-4:* $A.r(\boldsymbol{x}^0) \longleftarrow B.r_2(\boldsymbol{x}^1) \cap C.r_3(\boldsymbol{x}^2); \psi(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2)$

[3] Sorts resemble a very simple notion of variable types; each variable is assigned to exactly one sort, which defines the set of values over which it can range. This set is called the *carrier* of the sort.

The body of this statement is referred to as an *intersection*. Under each valuation that satisfies $\psi(\boldsymbol{x}^0, \boldsymbol{x}^1, \boldsymbol{x}^2)$, the statement defines $A.r(\boldsymbol{x}^0)$ to include principals that are members of both roles, $B.r_2(\boldsymbol{x}^1)$ and $C.r_3(\boldsymbol{x}^2)$.

Each of these statements is said to *define* $A.r(\boldsymbol{x}^0)$. A is called the *owner* of the role and the *issuer* of these statements, and has sole authority to define such statements in the policy. The *body* of a statement consists of the right-hand-side, to the right of the arrow. We abuse the terminology by saying that a constraint over the variables that occur in a statement (resp., role) is *a constraint over* that statement (role).

Semantics and Supported Constraint Domains. Extending work by Li *et al.* [10], Li and Mitchell [6] define the semantics of RT^C by translating policy statements into clauses in Datalog with constraints [11]. In this context, queries about role membership are given a standard semantics based on logical entailment. Li and Mitchell [11] identify some classes of constraint domains that efficiently support all the operations that are necessary for evaluating queries under Datalog with constraints. Our implementation techniques and the implementation itself support several such constraint domains. In them, all primitive constraints involve only a single variable, with the single exception of equality. (An equality $x = y$ is satisfied by a valuation only if that valuation assigns both variables the same value.) Other primitive constraints we consider require variable values to belong either to a set identified through enumeration, a numerical range, which may be unbounded on one or both ends, or to bear a hierarchical relation to a node in a tree.

In RT^C, a *query* asks about role membership. There are three kinds of queries. An *all-members* query is given by a constrained role, which takes the form $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$. A *principal solution* to an all-members query is the set of all principal/constraint pairs, $\langle D, \varphi(\boldsymbol{x})\rangle$, such that the Datalog translation logically entails that D is in $A.r(\boldsymbol{x})$ under all valuations that satisfy $\varphi(\boldsymbol{x})$, $\varphi(\boldsymbol{x})$ is satisfiable, and $\varphi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$. An *all-roles* query is given by a principal D and determines a set of constrained roles $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ that the Datalog semantics makes D a member of under all valuations satisfying $\psi(\boldsymbol{x})$; these constrained roles are referred to as *role solutions*. A *membership-decision* query, given a constrained role $A.r(\boldsymbol{x}); \psi(\boldsymbol{x})$ and a principal D, determines the set of constraints $\phi(\boldsymbol{x})$ under which D is a member of $A.r(\boldsymbol{x})$ for all valuations of $\boldsymbol{x}$ satisfying $\phi(\boldsymbol{x})$ and such that such that $\phi(\boldsymbol{x}) \Rightarrow \psi(\boldsymbol{x})$ [12].

2.1 The Credential Chain Discovery Algorithm

Our query evaluation engine implements the algorithm introduced by Mao *et al.* [12], which in turn is based on the algorithm for RT_0 introduced by Li *et al.* [9]. (RT_0 has no role parameters or constraints.) We now briefly summarize that algorithm.

Queries are answered by creating a portion of a directed graph called a *credential graph*. Each node in the graph represents either a role or a principal. The graph contains several types of edges, which correspond to statements in the RT^C policy. For instance, when the policy contains a statement of the form $A.r(\boldsymbol{x}^0) \longleftarrow B.r_2(\boldsymbol{x}^1); \psi_2(\boldsymbol{x}^0, \boldsymbol{x}^1)$, the credential graph contains an *implication edge* from the node representing $B.r_2(\boldsymbol{x}^1); \psi(\boldsymbol{x}^1)$ to the node representing $A.r(\boldsymbol{x}^0); \psi_1(\boldsymbol{x}^0)$, in which the constraints $\psi(\boldsymbol{x}^1)$ and $\psi_1(\boldsymbol{x}^0)$ depend on the manner (direction) in which the (partial) credential graph is constructed. To deal with Type 3 and 4 credentials (and other special cases), other types of edges are created which perform actions in concordance with their type; for example, an *intersection edge* requires principals to be members of both roles in the intersection before it may become a member of the role to which the edge leads.

Search Algorithms. The all-members query is answered through a *backward search* algorithm, which constructs the credential graph by creating a node representing the constrained role given by the query and expanding the constructed graph by traversing edges in a backward direction. Constructing and traversing edges requires the algorithm to access (possibly remote) repositories to retrieve the credentials that define those edges. Backward searches create only role nodes; principals are mentioned only in solutions and do not receive their own node. The *forward search* algorithm answers the all-roles query. It begins by creating a principal node, and expanding the graph by traversing edges in a forward direction. To handle linked roles in the forward search, principal nodes must also be used in addition to role nodes. Although the membership-decision query may be answered by either forward or backward search, a third algorithm referred to as the *bidirectional search* may also be used. A bidirectional search involves both a backward search on the query role and a forward search on the query principal simultaneously; these terminate as soon as the query principal is shown to be a member of the query role under the given constraint.

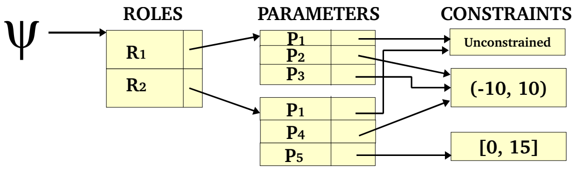
When a query is evaluated, the graph may be reused for any other queries thereafter (maintaining the graph through multiple queries is referred to as *graph caching* in later portions of this paper).

3 RT Credential Toolkit

The RT Credential Toolkit was designed to supply most of the components necessary for an authorization solution. The toolkit is composed of three main components: the credential authoring tool, the query engine, and the credential repository. Each of these components are detailed below.

3.1 Implementation Design Decisions

Representation of Constraints. The representation must support several operations: construction of internal representation from the representation given in credentials; conjunction of constraints; existential quantifier elimination; subsumption of one constraint by another; and duplication of representation. As mentioned above, the only primitive constraints we support that involve more than one variable are equalities, which leads to their having special treatment: all variables that are constrained to be equal refer to a single object representing



$$A.R_1(P_1,P_2,P_3) \longleftarrow B.R_2(P_1,P_4,P_5);$$
$$P_2 \;\epsilon\; (-10,20), \quad P_4 \;\epsilon\; (-20,10), \quad P_5 \;\epsilon\; [0,15], \quad P_2 = P_4, \quad P_3 = P_4$$

Fig. 1. Depiction of a constraint created for the credential shown

the constraint on values they can assume. Due to the simplicity of the constraint domains we consider, any consistent conjunction of non-equality primitive constraints can be reduced to a single primitive constraint. This enables a simple representation, illustrated in Figure 1. The representation of a constraint over a statement (or a role) has an entry for each role in the statement. Each role entry has an entry for each parameter of that role and each of these parameter entries is associated with an object representing a primitive constraint, which may be shared with other parameter entries. The special primitive-constraint object (PCO), *unconstrained*, indicates that all values of the corresponding carrier satisfy the constraint.

Among the operations that must be supported, providing an algorithm to implement conjunction is the least straightforward. We outline the conjunction algorithm in the next section.

Conjunction Algorithm. Our algorithm implementing conjunction needs to handle only the case in which one conjunct, ψ_1, is over some statement, and the other, ψ_2, is over a single role from that statement. The result, ψ_3, must be constructed without modifying ψ_1 or ψ_2. The algorithm begins by duplicating ψ_1. To maintain equality constraints from ψ_1, the duplication procedure uses a table that associates PCOs in ψ_1 with PCOs in the duplicate. (Note that this association is between objects, not the primitive constraints they represent.) Processing each role parameter in turn, when the PCO for that parameter in ψ_1 does not yet have an associated object in the duplicate, a new copy of that PCO is created and entered into the table as well as into the duplicate's parameter entry. When the table already contains an associated object in the duplicate, a reference to that PCO is entered in the duplicate's parameter entry. Thus, the sharing of PCOs in the duplicate is made to correspond precisely to that in ψ_1.

Next, we modify the duplicate, generating the result ψ_3, by conjoining the PCOs in ψ_2 with the corresponding PCOs in (the data structure that will become) ψ_3. Recall that ψ_2 is a constraint over some role in the statement that (the duplicate of) ψ_1 is over. The procedure considers each parameter of that role in turn, constructing a new PCO that represents the conjunction of those in ψ_2 and in the duplicate. This new PCO then replaces the old one in the duplicate. To preserve the equality constraints in the duplicate coming from ψ_1

while introducing the equality constraints from ψ_2, an association table is used to associate PCOs in ψ_2 with PCOs in ψ_3.

When processing a parameter, the table is used as follows. Let us call the PCO in the parameter's ψ_2 entry pco_2 and that in the parameter's ψ_3 entry pco_3. When the table does not contain an association for pco_2, the (simplified) conjunction of pco_3 and pco_2 is computed, entered into the table in association with pco_2, and is used to replace pco_3 in the constraint being constructed (ψ_3). (Note that when conjunctions are unsatisfiable, this is detected and handled at a higher level in the algorithm as this leads to greater efficiency.) When the table *does* contain an associated PCO, call it pco'_3, we check to see whether pco'_3 and pco_3 are one and the same object. If so, no further action is required. Otherwise, the (simplified) conjunction of pco_2, pco'_3, and pco_3 is computed, entered into the table in association with pco_2, and used to replace pco_3 in the constraint being constructed (ψ_3).

Repository Locations and Credential Typing. An acknowledged limitation of the currently available system is that it does not support credential typing of the type described by Li *et al.* [9]. This type discipline enables credentials to be stored with either their issuers or their subjects in a manner that ensures credential chains, if they exist, can be found. However, as a means of supporting distributed sets of credentials, each credential optionally contains the location of one or more credential repositories where other credentials relevant to the issuer, defined role, and body of that credential may be found. The engine is able to use this information during proof construction to find other credentials as needed. Extending the current system to support full credential typing is a possible future activity.

3.2 Components of the Toolkit

Credential Authoring Tool. The credential authoring tool enables users to create RT credentials easily by using a special-purpose graphical user interface. Technical activities such as public key creation, credential signing, and distributing credential to repositories are easy to perform.

Credentials are stored in an XML format with elements for the defined role, body, the credential validity period, and relevant credential repositories; the credentials are cryptographically signed by the issuer of the credential. To support the use of *RT* within an organization, public keys, role names, credentials, and repository locations may be imported along with their metadata from a local or online repositories. This allows one to start with a working set of useful objects, such as public keys for company employees or commonly used role names. The metadata associated with each object allows them to be referred to by short local names instead of their globally unique identifiers and also provides a way to attach and display easy to read descriptions of each object.

The ability to import and search existing credentials allows one to view a policy set that the user may wish to modify or extend; this feature may also help to present a simple or small portion of the overall policy set to organizational members who need to write only a limited set of new policy statements.

Additionally, test queries may be run from the authoring tool to ensure that new policies have the desired effect.

Engine. The engine is responsible for evaluating *RT* credentials to answer queries. Forward, backward, and bidirectional searches are supported. The engine has both Java API and command line interfaces which may be used to pose queries and read results. The engine automatically fetches relevant credentials from the appropriate repositories for processing.

Credential Repository. The credential repository is a relatively simple application which hosts, indexes, and serves credentials to engines. Credentials may be added to a repository remotely or locally. The repository also maintains information about what credentials have been served to an engine during the current connection to avoid sending credentials multiple times.

4 Performance Evaluation

Although other *RT* query evaluation programs have been developed, there has been little investigation into the costs of an *RT* system which is usable by the public, stores credentials in a distributed manner, or deals with policies approaching the size and complexity expected in real world scenarios.

The costs of using *RT* in terms of latency, throughput, network usage, and computational power required need to be assessed. It also must be determined whether *RT* is efficient under certain usage scenarios but performs poorly under others and what factors primarily determine this difference. If performance is poor, new strategies may be necessary to maintain the low latency required for a useful authorization system.

4.1 Policy Benchmarks

Large scale authorization policies used in practice are difficult to obtain. Instead, we base our analysis on automatically generated policies designed to have characteristics likely to arise in realistic scenarios. To this end, we have constructed a policy generator that uses pseudo-random methods to generate policies based on a collection of parameters that can be tuned to yield policies likely to be representative of policies that would be used in a variety of scenarios. We begin by describing the policy generator and its parameters. We then describe scenarios that seem appropriate candidates for application of RT^C.

Policy Set Generator. The policy set generator creates all of the credentials necessary to define a rough hierarchy of roles based on a set of input parameters described below. At a high level, the generator creates the policy set for the hierarchy one level at a time, beginning at the base, and takes the following steps to do so:

1. The generator begins by defining the *base roles*. The base roles are the roles at the bottom level of the hierarchy which directly include the main body of principals and are generally defined by Type-1 policies only.

2. A new level of roles are created above the previous level. The set of policies created in step 3 will define these new roles and will include roles from the previous level in the body. The number of roles defined at each level must be at least one fewer than the level below it.
3. For each of the roles in the previous level, we will create a statement whose type is randomly selected based on the given parameters. If we select Type-2, the role itself is used as the body of the new statement. If we select Type-4, the role appears in an intersection in the body of the statement along with another role randomly chosen from the set of already defined roles. If we select Type-1, we create two policies: a Type-1 with a randomly selected principal as the body and a Type-2 with the role as the body. For Type-3, we do the following:
   (a) Create a linked role of the form $A.r_1.r_2$ where A is a randomly selected principal and r_2 is the role name from the role we are currently working with. This linked role will appear in the body of the new statement.
   (b) Create $(i - 1)$ Type-1 policies of the form $A.r_1 \longleftarrow B$ where B is a randomly selected principal and i is the parameter described below.
   (c) Create one Type-1 statement of the form $A.r_1 \longleftarrow D$ where D is the principal from the role we are currently working with.

   For each statement statement written, we advance use the next available role from the new level as the defined role. When we reach the end of the list of new roles, we start again from the beginning.
4. Repeat from step 2 until we arrive at a level with only one defined role.

Parameters for the Policy Set Generator.

1. Number of principals, n: the number of principals that are eligible to be members of the base roles. Principles defining roles within the tree are also randomly drawn from this range.
2. Number of base roles, r: the number of base roles for the hierarchy.
3. Principal inclusion percentage, p: the probability that any given principal is a member of any given base role. This results in an approximately normal distribution of the number of principals in any base role with mean np and variance $np(1 - p)$.
4. Level-to-level reduction percentage, l: the reduction in the number of roles defined from one level to the level above it. This governs how quickly the hierarchy narrows, controlling the number of levels and correspondingly the maximum height of any credential chain.
5. Chain die-off percentage, d: the percentage of roles at each level which will not appear in the body of any credentials defining the next level of roles. This controls the variability of chain height; a value of 0.0 will result in all chains reaching the maximum height while a value of 0.5 will result in approximately half of the remaining chains ending at each level.
6. Type-1 percentage within the hierarchy, t_1: the percentage of credentials defining roles within the hierarchy that are Type-1. Note that this percentage does not include Type-1 credentials that define the base roles. This parameter

can be used to account for management or other special principals who may be included directly in a mid to high level role.

7. Type-3 percentage within the hierarchy, t_3: the percentage of credentials defining roles within the hierarchy that have linked roles as their body.
8. Number of principals in the intermediate role of every linked role, i: whenever a linked role is created, this many principals are randomly picked from existing roles in the hierarchy and are added to the intermediate role of the linked role. There is no guarantee that one of these principals will define a role whose role name matches the second role name from the linked role.
9. Type-4 percentage within the hierarchy, t_4: the percentage of credentials defining roles within the hierarchy that have intersections as their body.

All credentials within the hierarchy that are not Types 1, 3, or 4 are Type-2.

Scenarios. The following scenarios are possible real world examples which might require policies that *RT* is well suited for describing. These policies may rely heavily on delegation of authority, which is relatively flexible and straightforward in *RT*. Credentials storage is likely to be distributed in two of the scenarios, which *RT* is designed for, while the other two may have more centralized storage.

Parameters and constraints were not used in these scenarios due to the complexity involved their generation. However, our benchmarks show that the performance impact of using parameters throughout a set of policies similar to the government department scenario described below is minimal, generally resulting in less than a 10% increase in time to complete a query, including the retrieval of credentials from remote repositories.

Grid Scenario. Several virtual organizations (VOs) collaborate to create a grid infrastructure. Each VO has a separate internal policy. A small set of policy statements is created to govern access to the various grid tools by VO members; these policies reference one or two top level roles from each VO. Additionally, some principals are members of more than one VO.

Policy sets for five VOs of different sizes were generated, with n ranging from 100 to 500, r ranging from 10 to 20, l ranging from .25 to .4, p ranging from .35 to .25, i ranging from 3 to 5, t_1 equal to .1, t_3 equal to .05, t_4 equal to .15, and d equal to .1. The sets of principals used for the different VOs were nearly disjoint, with an overlap of 20 or fewer principals.

The parameters which vary depending on the size of the VO follow these expectations: first, while the average group size will increase, the overall number of groups will not increase as quickly. Second, as the size of an organization grows, specialization is likely to occur, causing the average number of roles a single principal is a member of to grow more slowly than the total number of roles available. Third, the hierarchy is likely to increase in width more quickly than height.

Credentials representing grid authorization roles were written by hand. These defined a single top level role and three roles below it, each of which included roles from the top and middle of the hierarchies for three of the five VOs.

Government Department Scenario. A large government department creates a policy that roughly resembles a single hierarchy with many management roles. Credential storage is mostly centralized.

The following parameters were used: $n = 10000$, $p = .1$, $r = 100$, $l = .5$, $t_1 = .2$, $t_3 = .1$, $i = 10$, $t_4 = .15$, and $d = .1$. These represent a more complex policy, with a greater number of linked roles and intersections. The maximum chain height was seven.

E-Bookstore. An electronic bookstore wishes to offer discounts to students of all small accredited universities nearby. Each university maintains its own simple policy which groups students by department and describes membership for several of the larger extra-curricular roles.

Policy sets for 13 universities were generated, each with the following parameters: $n = 5000$, $p = .1$, $r = 10$, $l = .6$, $t_1 = 0$, $t_3 = .02$, $i = 3$, $t_4 = .1$, and $d = .1$. These create very simple policies with a maximum chain height of three. The top level role for each university was $\mathit{University}_i$*.student()*, where $1 \leq i \leq 13$. The following credentials were created by hand for the E-bookstore:

1. *E-Bookstore.discount()* ⟵ *AccredBoard.university().student()*
2. *AccredBoard.university()* ⟵ $\mathit{University}_i$

Social Network. A new social network is created. Every principal defines a set of friends, and each principal can see their extended network, which includes friends of friends, and their 2nd extended network, which includes friends of extended friends.

Policy sets for 10000 principals were created. Each principal randomly added principals to their *friends()* role from a range of 1000 principals, with each principal having a 10% chance of inclusion in the role. The range of principals that friends were selected from was occasionally shifted to give the effect that principals nearby are likely to share many friends, while principals far apart share few or no friends. Each principal, P, also created the following credentials:

1. *P.extendedFriends()* ⟵ *P.friends().friends()*
2. *P.secondExtendedFriends()* ⟵ *P.extendedFriends().friends()*

Repository Distribution. The repositories were spread between the University of Texas at Austin the University of Texas at San Antonio. The bandwidth available to the repositories was great enough to not be a significant limiting factor during performance testing.

For the grid and E-bookstore scenarios, all credentials for each VO or university were stored in a single repository, with all repositories being used. This simulates what one is likely to find in real world scenarios where credentials are likely to be stored centrally per organization, but the organizations themselves are likely to be distributed.

For the government department and social networking scenario, all credentials were split evenly between two repositories near the engine. For these scenarios, we felt that centralized storage of the credentials was more likely in

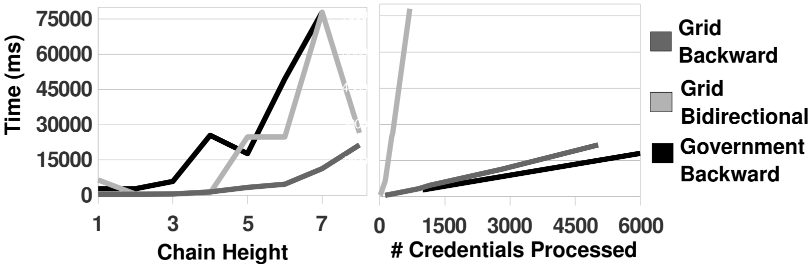


Fig. 2. Total Query Processing Time for: Backward Search in the Government Scenario; Backward and Bidirectional in the Grid Scenario.

real world examples. Users of such facilities seem unlikely to avail themselves of repositories not operated by the sites in questions.

4.2 Results

The credential repositories were run on machines of varying hardware, including a 2.13GHz Core2 with 2GB of RAM, a 4-core 2.66GHz Xeon with 16GB of RAM, an 8-core 2.0GHz Xeon with 8GB of RAM, and 8-core 3.0GHz Xeon with 32GB of RAM that was also used for the engine.

All results shown below are for searches with no portion of the graph cached unless stated otherwise. With a fully cached graph (meaning all nodes have been processed), the time to execute any query, regardless of query type or which roles and principals the query was performed on, was less than 10ms.

Grid Scenario. As seen in Fig. 2, the total time required to evaluate a query by using backward search in the Grid Scenario was linearly proportional to the number of credentials processed. Because of the hierarchical nature of the policies used in this scenario, the number of credentials processed increased exponentially with respect to the maximum chain height of the query role. The average time required per credential processed was 4.25ms. The time required to perform a forward search was also approximately linearly proportional to the number of credentials processed, but the average time required per credential processed was much higher, at 250ms.

Like the backward and forward search, bidirectional search exhibited a linear increase in time proportional to the number of credentials processed. The number of credentials processed with respect to the length of the chain between the query role and principal was far fewer than backward search: anywhere from .2 to .06 the number of credentials processed during a backward search of the same height. The average time required per credential processed was in between that of backward and forward search, at 88ms.

Backward search in the grid scenario generated a maximum of 400 KB/s of download traffic and 200 KB/s of upload traffic for the engine, while for-

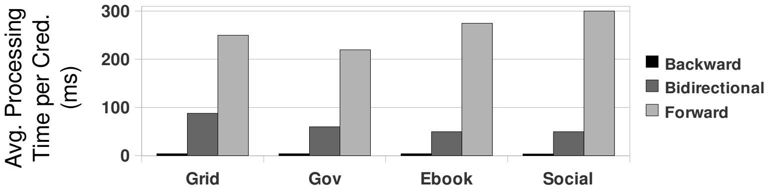


Fig. 3. Average processing time per credential among the three query types.

ward search generated only 45 KB/s down and 80 KB/s up. Bidirectional search generated 80 KB/s of download traffic and 80 KB/s up.

CPU utilization on the engine remained at or above 90% during backward search. During forward search it was much lower: usually less than 10%; CPU utilization during forward search also had a downward trend as a search progressed. Although bidirectional searches initially showed greater CPU utilization than forward searches, they progressed more slowly as the search continued and CPU utilization became only slightly better than that of forward search.

Government Department Scenario. As seen in Fig. 2, backward search performance in the Government Department Scenario was similar to that of the Grid Scenario, but with a slightly lower time required per credential processed at 4ms. CPU utilization and network traffic was similar to that of the Grid Scenario. Bidirectional searches took an extremely long time to complete. In the case that the query principal was not a member of the query role, a single search took about 90 minutes and processed about 90000 credentials, averaging about 60ms per credential processed.

E-Bookstore Scenario. Backward searches in the E-Bookstore scenario completed in 9 seconds on average when performed on *University.student()* roles with an average of 3ms required per credential processed; when the query role was *E-Bookstore.discount()*, the time required per credential increased to 6ms. Forward and search performed more poorly than in the grid and government department scenarios, requiring 275ms per credential processed. Bidirectional search was slightly better than the government dept. scenario, at 50ms per credential processed.

Social Networking Scenario. For backward search, queries on *friend()*, *extendedFriend()*, and *secondExtendedFriend()* queries took .5, 28, and 490 seconds on average, and average time per credential processed was was 5.6, 3.5, and 2.8ms, respectively. CPU and network utilization remained extremely high, similar to the other scenarios.

Forward search performed extremely slowly, with low CPU utilization and network usage. The completion time for a single query was roughly two hours. After extended periods of time, the forward search slowed to a rate of three credentials processed per second.

Bidirectional search performed moderately in the case that the query principal was a member of the query role; for *friend()* and *extendedFriend()* roles, queries completed in about 10 seconds, while *secondExtendedFriends()* queries completed in about 20. When the query principal was not a member of the query role, bidirectional searches initially had moderate CPU utilization and network traffic before degenerating into performance slightly better than forward search.

4.3 Analysis

As the results from the performance tests show, the difference in response times for queries with and without graph caching is staggering. At the scale of these tests, it is clear that caching all or most of the credential graph makes an enormous difference to the expected query response time. When the graph is fully cached, query response times are consistently extremely low. We suggest techniques for maintaining such a cache in the next section.

The amount of time required to complete a query increased nearly linearly with respect to the number of credentials processed for all three types of queries across all scenarios. The average processing time per credential did decrease slowly for backward search as the number of credentials grew, while the average time per credential increased slowly for forward and bidirectional searches.

Backward search performed very similarly regardless of the scenario. Its performance appears to be almost entirely CPU bound, although network traffic could become a limiting factor in some cases. The primary factor which affected backward search performance was the number of roles overall; the percentage of Type-3 and Type-4 credentials also impacted performance to a lesser degree.

The differences in performance between forward and backward search were unexpected and quite large. There are two primary differences which may account for the disparity:

First, backward searches create only role nodes, while forward searches create both principal nodes and role nodes. In these scenarios, the number of principals greatly exceeds the number of roles. The large increase in the number of nodes requires much more memory and substantially increases the rate of cache misses, which results in the extremely low CPU utilization seen during forward searches.

Second, backward searches need to perform extra work only to deal with linked roles and intersections when a Type-3 or Type-4 credential is encountered while processing a role node. By contrast, to handle linked roles a forward search must create new nodes and perform several actions for each node processed.

For these reasons, forward search is unable to effectively utilize the CPU or generate network traffic; memory access time is the limiting factor.

Because bidirectional search runs both a forward and a backward search, it suffers from the same performance problems as forward search. Although the number of credentials processed are often low when the query is a success, the average time per credential processed is still high. Therefore, it cannot be depended upon as a means of graph creation.

The average processing time per credential grows faster with the number of credentials in the forward search than in the backward search. This is because

Type-1 credentials tend to constitute above 90% of the number of credentials in a policy, processing a greater number credentials in a forward search generally results in the creation of many additional principal nodes, and consequently, poorer performance. On the other hand, an increase in the number of credentials processed in a backward search results in the creation of relatively few role nodes whose performance impact does not outweigh the loss of performance overhead during earlier portions of graph construction.

4.4 Optimizations

Because the performance of an *RT* system is largely dependent on fully caching the graph, optimizations should concentrate on caching the graph quickly and efficiently. Due to the performance differences between forward, backward, and bidirectional search, the quickest way to build a complete graph is through backward searches on all roles that are used for authorization purposes. By avoiding forward and bidirectional searches entirely, performance is significantly enhanced. However, for this strategy to work, all credentials chains must be backward-traversable [9]. This requirement is not particularly onerous, but may be difficult to maintain in loosely organized, widely distributed scenarios. Without this guarantee, designing an efficient caching strategy is more complex.

Additionally, because backward search is primarily CPU bound, converting the currently single-threaded engine to a multi-threaded model is expected to improve performance significantly.

Multiple Engines. Because revocation and expiration become frequent for large policies, their effect on query response times must be minimized. Currently, when a credential expires or is revoked, our only available strategy is to rebuild the entire graph without that credential. An engine that is busy constantly rebuilding the graph will have no way to respond to queries in a timely manner. Although one possible solution to this problem is to devise a way to alter the graph so that the work required to handle credential invalidation is lessened, the frequency of these events and the expected time to deal with them still prevents the engine from responding to queries quickly.

An alternative strategy runs multiple engines simultaneously. One engine is dedicated to answering all queries with the latest complete version of the graph. For each credential invalidation, a separate engine constructs a new version of the graph in the background; when the new graph is complete, that engine assumes the role of servicing queries while the engine it replaces waits to build a new version of the graph after the next credential invalidation. The number of engines running concurrently could grow to accommodate increased workloads.

5 Conclusion

In this paper, we announced the availability of an open source implementation of RT^C and used it to test and analyze the performance of RT^C systems in a large, distributed environment. The results of these tests show that low response times for role membership queries may only be achieved through thorough caching of the graph. Furthermore, the characteristics of the backward search algorithm make it a much better candidate for graph creation than either forward or bidirectional search due to more efficient memory usage. Utilizing this knowledge, we have proposed strategies for cached-graph management that will allow RT^C to achieve the query evaluation-response times necessary to make it a viable authorization solution in distributed environments that benefit from easy delegation of authority.

References

1. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (1996) 164–173
2. Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.D.: The KeyNote trust-management system, version 2. IETF RFC 2704 (1999)
3. Clarke, D., Elien, J.E., Ellison, C., Fredette, M., Morcos, A., Rivest, R.L.: Certificate chain discovery in SPKI/SDSI. Journal of Computer Security **9**(4) (2001) 285–322
4. Gunter, C.A., Jim, T.: Policy-directed certificate retrieval. Software: Practice & Experience **30**(15) (2000) 1609–1640
5. Jim, T.: SD3: A trust management system with certified evaluation. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (2001) 106–115
6. Li, N., Mitchell, J.C.: RT: A role-based trust-management framework. In: The Third DARPA Information Survivability Conference and Exposition (DISCEX III), IEEE Computer Society Press (2003)
7. Becker, M.Y., Sewell, P.: Cassandra: Distributed access control policies with tunable expressiveness. In: POLICY '04: Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY'04), Washington, DC, USA, IEEE Computer Society (2004) 159
8. Czenko, M., Tran, H., Doumen, J., Etalle, S., Hartel, P.H., den Hartog, J.: Nonmonotonic trust management for p2p applications. CoRR **abs/cs/0510061** (2005)
9. Li, N., Winsborough, W.H., Mitchell, J.C.: Distributed credential chain discovery in trust management. Journal of Computer Security **11**(1) (2003) 35–86
10. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management framework. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy, IEEE Computer Society Press (2002) 114–130
11. Li, N., Mitchell, J.C.: Datalog with constraints: A foundation for trust management languages. In: Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages (PADL 2003). Number 2562 in LNCS, Springer (2003) 58–73
12. Mao, Z., Li, N., Winsborough, W.H.: Distributed credential chain discovery in trust management with parameterized roles and constraints (short paper). In: ICICS. (2006) 159–173