An Empirical Assessment of Approaches to Distributed Enforcement in Role-Based Access Control (RBAC)

Marko Komlenovic
mkomlenovic@uwaterloo.ca

Mahesh Tripunitara
tripunit@uwaterloo.ca

Toufik Zitouni
tzitouni@engmail.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

We consider the distributed access enforcement problem for Role-Based Access Control (RBAC) systems. Such enforcement has become important with RBAC's increasing adoption, and the proliferation of data that needs to be protected. We assess six approaches, each of which has either been proposed in the literature, or is a natural candidate for access enforcement. The approaches are: directed graph, access matrix, authorization recycling, CPOL, Bloom filter and cascade Bloom filter. We consider encodings of RBAC sessions in each, and propose and justify a benchmark for the assessment. We present our results from an empirical assessment of time, space and administrative efficiency based on the benchmark. We conclude with inferences we can make regarding the best approach to access enforcement for particular RBAC deployments based on our assessment.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access Controls*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed Applications*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms

Security, Performance

Keywords

Role-Based Access Control, Access Enforcement, Efficiency, Empirical Assessment

1. INTRODUCTION

Modern enterprises generate and archive large amounts of data. Such data needs to be protected by access control systems. Access control deals with the provision of regulated accesses to resources by principals and is one of the most important aspects of security. The proliferation of data requires access control systems to scale to tens of thousands of resources and permissions [1].

CODASPY'11, February 21–23, 2011, San Antonio, Texas, USA.

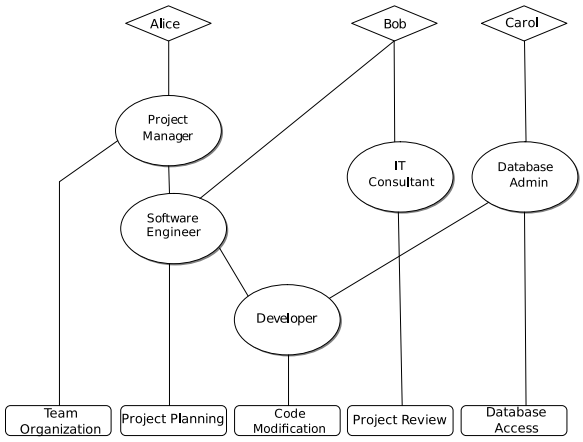

Figure 1: An Example RBAC policy. Users are shown in diamonds, roles in ovals and permissions in rectangles. Edges represent user-role, role-role and role-permission assignments. In the example, the user Alice is assigned to the role Project Manager and is therefore authorized to the permission Team Organization. She is also authorized to the role Developer, and therefore to Code Modification.

An important aspect of this scalability issue is the efficiency of access enforcement. Access enforcement is the process by which an entity called a reference monitor makes an 'allow' or 'deny' decision when a principal requests access to a resource. We consider access-enforcement in the context of Role-Based Access Control (RBAC) [2, 3], which is increasingly becoming the de-facto standard for access control in enterprise settings. In RBAC, rather than assigning a user directly to a permission, we assign a user to roles, and the roles to permissions. Also, the roles are associated with one another in a partial ordering called a role-hierarchy. An example of an RBAC policy is shown in Figure 1. There has been considerable research on RBAC. However, to our knowledge, there is very little work on efficient, scalable access-enforcement.

An approach to the problem is to distribute access enforcement across several reference monitors. With such an approach, a single, monolithic reference monitor is no longer a performance bottleneck. Such distributed enforcement, however, can be at odds with what is touted as one of the main benefits of RBAC – the ease of administration. Wei et al. [4] have proposed an architecture for distributed en-

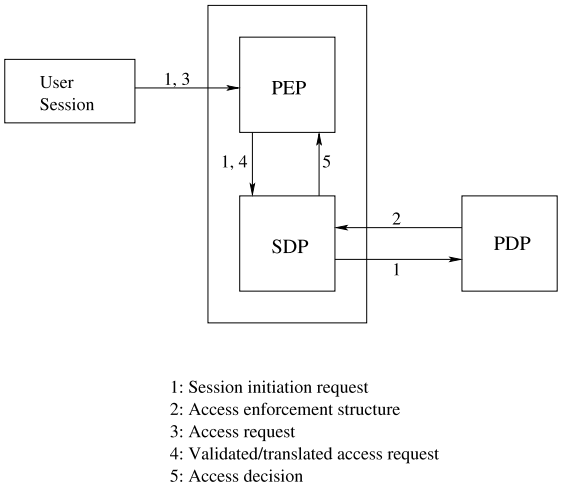

Figure 2: An architecture, reproduced from prior work [4, 5], for distributed access-enforcement in RBAC, and an associated process-flow. Our focus is on the Access enforcement (data) structure that is received by the SDP in Step 2, and that it uses to make access decisions.

forcement that attempts to reconcile these two issues (see Figure 2, which is a reproduction of corresponding figures from prior work [4, 5]).

In the architecture, the Policy Decision Point (PDP) is a centralized entity at which the RBAC policy is maintained. This centralization eases administration. Enforcement is performed at Policy Enforcement Points (PEPs). PEPs are aided by Secondary Decision Points (SDPs). An SDP can be seen as a cache of a portion of the RBAC policy from the PDP. In Figure 2 we show a typical chronological flow of events. In Step 1, a user activates a session at a PEP/SDP. In RBAC, users exercise permissions in sessions. A session is associated with a set of roles to which the user is authorized in the RBAC policy. In the example in Figure 1, users Alice and Bob may activate sessions s_a and s_b respectively. Alice may associate session s_a with the role Software Engineer, which authorizes s_a to the permissions Project Planning and Code Modification. Bob may associate s_b with the roles Software Engineer and IT Consultant, which authorizes s_b to the permissions Project Planning, Code Modification and Project Review.

The request to activate a session propagates to the PDP, which makes the decision on whether it is allowed. If it is, in Step 2, the PDP communicates a data structure to the SDP that the latter uses in Steps 3, 4 and 5 to make decisions on access requests that pertain to that session, that are communicated to it by the PEP.

The question we seek to answer is: what are the data structure and associated algorithms we should use in an SDP? There is evidence that "general purpose" approaches, such as storing an access control policy in a database and using the querying capabilities of the database, do not lend themselves to efficient access enforcement [6]. Consequently, it is necessary to carefully consider the approach that is used. In answering the question, we evaluate an approach along the following three axes.

Time efficiency – our primary goal is to make the SDP time efficient. An access check should be fast.

Space efficiency – we consider also the space that a particular data structure may take at the SDP. Space and time efficiency can be at odds; this is the classical time-space trade-off.

Administrative efficiency – with this, we ask whether a particular data structure at the SDP lends itself to easy administration in the propagation of administrative changes that are made at the PDP, to the SDP. We quantify this as the time it takes to update the SDP when an administrative change is made to the RBAC policy at the PDP.

Our approach to answering the question is to empirically assess six candidates. A challenge in conducting an empirical assessment is the lack of a meaningful benchmark. The establishment of meaningful benchmarks is seen as an important milestone in several settings in computing. We propose and adopt a benchmark in this paper (see Section 4). Our objective is for what we propose to serve as a macro-benchmark [7] – one that has RBAC policies and session profiles that are realistic.

In summary, our contribution is an assessment of six approaches to distributed access enforcement in the context of RBAC using a meaningful benchmark that we propose. In Section 3.1, we justify our choice of the six approaches.

The remainder of the paper is organized as follows. In the next section, we discuss related work. In Section 3, we describe the six approaches that we assess. In Section 4, we describe our benchmark and rationalize it. In Section 5, we present our assessment and results from it. We conclude in Section 6 with a "good," "fair" and "poor" rating of each of the six approaches along the three axes we mention above.

2. RELATED WORK

There is large amount of research in distributed access control, and in distributed RBAC in particular. However, there is relatively little work on efficient access enforcement in these contexts. To our knowledge, CPOL [6] is the state of the art in access enforcement in distributed settings. CPOL employs caching and a structure called an AccessToken that is application-specific to speed-up access enforcement. The work on CPOL points out also that simply using database querying does not suffice for fast access enforcement. Our work is close also to those of Wei et al. [4], Tripunitara and Carbunar [5] and Liu et al. [8], that address the access enforcement problem in RBAC. Wei et al. [4] propose the architecture that we adopt in this paper (see Figure 2). In that context, they propose authorization recycling which is one of the approaches that we assess. Tripunitara and Carbunar [5] adopt the architecture of Wei et al. [4] and propose an approach called the cascade Bloom filter for access checking. Their focus is fast and space efficient access checking for RBAC in low-capability devices. Liu et al. [8] propose a technique that they call transformations for access checking in RBAC. We see a transformation as encoding RBAC in an access matrix; it is one of the approaches that we assess.

3. THE APPROACHES

We compare six approaches in this work. They are: directed graph (Section 3.2), access matrix (Section 3.3), CPOL (Section 3.4), authorization recycling (Section 3.5) and the Bloom filter and the cascade Bloom filter (Section 3.6). For each approach, we discuss how we encode RBAC sessions in the particular data structure. We begin in Section 3.1 with a justification of the six choices.

3.1 Basis for the Choice of Approaches

Our primary objective is to compare four approaches from the research literature that have been proposed for distributed access enforcement. These are the access matrix [8], CPOL [6], authorization recycling [4] and the cascade Bloom filter [5]. Of these, the last two have been proposed specifically in the context of distributed access enforcement for RBAC. Their performance has been assessed only in isolation, and not relative to other approaches.

The access matrix [9, 10] is a well-established syntax for access control, with a long history. Liu et al. [8] propose its use (somewhat indirectly) for access control in RBAC. They do not consider RBAC sessions; we devise and adopt a particular encoding (see Section 3.3). In the work on CPOL [6], only an encoding of a trust management language in CPOL has been presented, and its performance has been assessed in that context only. The main elements of CPOL are sufficiently general that it can be used for RBAC (see Section 3.4 for our encoding). Consequently, we argue that it is an important candidate for access enforcement in RBAC.

The directed graph and the Bloom filter are two other candidates we consider. A natural representation of RBAC is as a directed graph. Consequently, it behooves us to consider it. The Bloom filter is also a meaningful candidate given that the cascade Bloom filter [5] is an extension to it, and an empirical argument over the use of the cascade Bloom filter over the "vanilla" Bloom filter has not been made before.

Certainly, one can think of other data structures that may be used for access-enforcement in RBAC. For example, one could encode RBAC sessions as Access Control Lists (ACLs) for the purpose of enforcement. However, we argue that our work gives a broad coverage of the possible approaches with valuable insights for other possible approaches as well. For example, an ACL is an encoding of an access matrix, which is one of the candidates we consider. There are also similarities between ACLs and the directed graph in the context of RBAC – both approaches are linear from the standpoint of time efficiency.

3.2 Directed Graph

An RBAC policy can be seen as a directed graph with a particular structure – it is acyclic, and its vertices can be partitioned into three sets (users, roles and permissions), with constraints on edges between the sets (e.g., the only outgoing edges from users are to roles). A natural data structure to use in the SDP, consequently, is a directed graph. When a session is activated, the PDP communicates to the SDP, in Step 2 of Figure 2, a directed graph, G.

Let $\widehat{G}$ be the complete RBAC policy at the PDP perceived as a directed graph. Let $S = \{s_1, \ldots, s_n\}$ be the set of sessions that are active at a PEP, and $R_i = \{r_1, \ldots, r_{m_i}\}$ be the set of roles that are associated with the session s_i. Let $P = \{p_1, \ldots, p_k\}$ be the set of permissions that are reachable in $\widehat{G}$ from the roles in $\bigcup_i R_i$. Then the vertices of G are $S \cup P \cup R_1 \cup \ldots \cup R_n$. The edges of G are $\{\langle s_i, r_j\rangle : r_j \in R_i\} \cup E(I)$ where $E(I)$ is the set of edges of the subgraph I of $\widehat{G}$ that is induced by the vertices in $P \cup R_1 \cup \ldots \cup R_n$. That is, G is similar to a subgraph of $\widehat{G}$, except with sessions in place of users, and the edges induced by the vertices that are relevant to the sessions.

The access $\langle s, p\rangle$ is allowed if and only if the vertex p is reachable from s in G. We represent G as an adjacency list, which is a standard representation of a graph [11]. As an example, consider the sessions s_a and s_b from Section 1 for the RBAC policy in Figure 1. The session s_a is activated by Alice and is associated with the role Software Engineer. The session s_b is activated by Bob and is associated with the roles Software Engineer and IT Consultant. The resultant directed graph at the SDP is shown in Figure 3.

3.3 Access Matrix

The access matrix [9, 10] is a canonical and intuitively appealing representation for an access control policy. Consequently, we consider it a natural candidate as the data structure in an SDP. Our encoding of RBAC sessions in an access matrix is straightforward. Rows in the matrix are indexed by sessions, and columns are indexed by permissions. An entry in the matrix is a bit. An access request, $\langle s, p\rangle$ is an index into the matrix, and can be checked in constant-time. The access matrix that results at the SDP for our example sessions s_a and s_b from the previous section is shown in Figure 3.

3.4 CPOL

CPOL [6] is an approach to distributed access enforcement that has been proposed in the context of trust management. In trust management, the policy is distributed as well. Also, the syntax of policies is different from RBAC. Consequently, we need to provide an encoding of RBAC sessions in CPOL.

In CPOL, an AccessToken is used to determine whether access should be granted or not. In the original design [6], an AccessToken is opaque – its structure is specific to an application. A policy comprises Rules; each Rule contains an AccessToken. To check whether a particular access should be granted, we check the set of Rules and determine whether any of them contains an AccessToken that grants the access. For faster access enforcement, it is possible to aggregate AccessTokens in a Cache which is a keyed table.

Our encoding of RBAC in CPOL is as follows; we argue that this is the most natural encoding. We implement the SDP as a CPOL Cache. The key into the Cache is a session identifier. Each AccessToken is a set of permissions to which the session is authorized. Our study of the original implementation of CPOL suggests that there are two important aspect that affect the time efficiency of access checking: the manner in which the set of permissions is implemented within an AccessToken, and caching.

In our example of the sessions s_a and s_b for the RBAC policy from Figure 1 from the previous sections, we would have two CPOL Rules, Rule_a and Rule_b, which contain AccessToken_a and AccessToken_b respectively. AccessToken_a is {Project Planning, Code Modification}. AccessToken_b is {Project Planning, Code Modification, Project Review}. The cache has keys s_a and s_b, for the two access tokens.

In our reimplementation of CPOL, we have adhered closely to the original implementation. In Section 5, we discuss

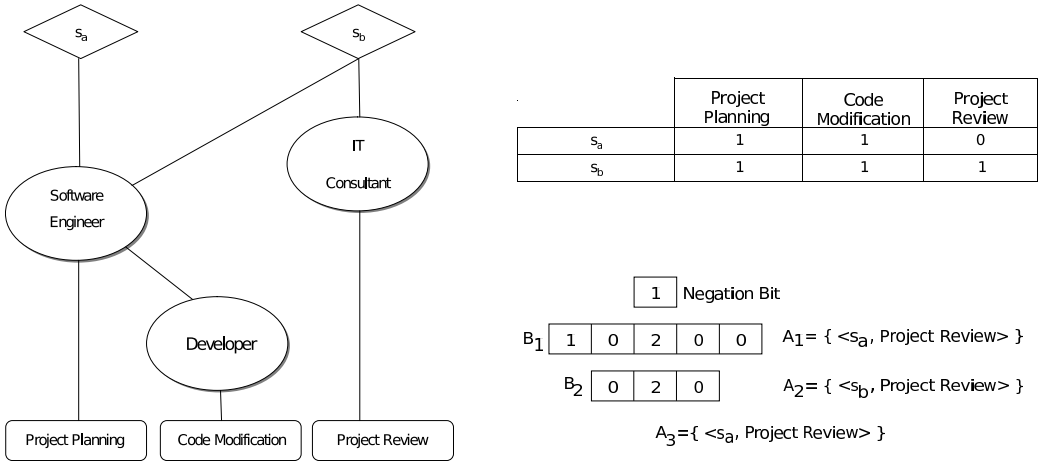

Figure 3: The directed graph, access matrix and cascade Bloom filter with 2 levels for our example sessions s_a and s_b that are discussed in the text, for the RBAC policy in Figure 1. (We discuss the encodings for CPOL and authorization recycling in Sections 3.4 and 3.5 respectively.) For the cascade Bloom filter, we assume that three indexing functions are used for Level 1, and two are used for Level 2. The Negation Bit is set, which indicates that A_1 is the set of authorizations that are disallowed. A_2 is the set of false positives in B_1, and $A_3 \subseteq A_1$ is the set of false positives in B_2.

why we have based our assessment on a new implementation. Our implementation compares in performance to the original (see Section 5). For ease of administration, we also maintain a set of Rules that are associated with an SDP, and a Condition for each entry of the Cache, as specified in the original design.

3.5 Authorization Recycling

Authorization recycling [4] is an approach to distributed access-enforcement in RBAC that was proposed in conjunction with the architecture in Figure 2. In this approach, two caches are maintained at the SDP, Cache$^+$ and Cache$^-$. The entries in Cache$^+$ indicate authorizations that are allowed, and the entries in Cache$^-$ indicate authorizations that are disallowed. An entry in a cache is $\langle R, p\rangle$, where R is a set of roles and p is a permission.

The access-enforcement algorithm is as follows. An access request $\langle s, p\rangle$ is mapped to $\langle R, p\rangle$, where R is the set of roles with which the session s is associated. We first check whether there exists an entry in Cache$^-$ of the form $\langle R', p\rangle$ such that $R \subseteq R'$. If there is, we deny the access request. This is because $R \subseteq R'$ implies that no role in R is authorized to p. Otherwise, we check whether there exists an entry in Cache$^+$ of the form $\langle R', p\rangle$ such that $R \supseteq R'$. If there is, we allow the request. This is because $R \supseteq R'$ implies that some role in R is authorized to p.

It is possible that neither of the above two conditions is met by $\langle R, p\rangle$ that corresponds to an access request $\langle s, p\rangle$. In this case, we need to consult the PDP. Unlike with the other approaches, in authorization recycling, no data structure is communicated from the PDP to the SDP in Step 2 of Figure 2. Rather, when an access request is made, if there is a match in neither Cache$^-$ nor Cache$^+$, the SDP communicates with the PDP and then updates its Cache$^-$ and/or Cache$^+$. We refer the reader to Wei et al. [4] for the algorithm that is used to update the two caches.

In our example of the two sessions, s_a and s_b for the RBAC policy from Figure 1 from the previous sections, both Cache$^+$ and Cache$^-$ are empty before a request for access takes place. Assume that Bob, in session s_b, requests permissions Project Planning and Database Access. The former succeeds and the latter fails, and for each attempt, the SDP acquires new state from the PDP. Cache$^+$ contains $\langle \{\text{Software Engineer}, \text{IT Consultant}\}, \text{Project Planning}\rangle$ after the request for Project Planning, and Cache$^-$ contains $\langle \{\text{Software Engineer}, \text{IT Consultant}\}, \text{Database Access}\rangle$ after the request for Database Access. Now, when Alice, in session s_a, requests permission Project Planning, the request succeeds based on the contents of Cache$^-$ and Cache$^+$.

3.6 (Cascade) Bloom Filter

A Bloom filter [12] is a probabilistic time- and space-efficient data structure for encoding a set A, and checking membership in A. We assume a universe, U of which A is a subset. A Bloom filter B is an array of m bits, with indices 0 through $m-1$. It is associated with k indexing functions $h_1, \ldots, h_k$ each of which maps every $e \in U$ to some integer between 0 and $m-1$. To represent that $e \in A$, we set the bit to which each h_i maps. To check whether $e \in A$, we check whether the bit to which every h_i maps is set. A counting Bloom filter [13] makes removal from a Bloom filter easier by associating a counter with each index rather than a bit.

As k and m are constants in the size of A (and U), B is represented in constant-space, and we can check whether $e \in A$ in constant-time. However, this check can result in a false positive; i.e., a check may return 'true' in some cases for which $e \notin A$. Consequently, a Bloom filter trades-off a probability of false positives for time- and space-efficiency.

The cascade Bloom filter [5] is an extension of the Bloom filter and has been proposed for the context with which we deal in this paper – distributed enforcement for RBAC. This approach uses several Bloom filters and associates each with a level, $l \geq 1$. Let the set encoded by the Bloom filter at

level i, B_i, be called A_i. Then, $A_0 = U$, $A_1 = A$, and for $i > 1$, A_i comprises elements of A_{i-2} that are false positives in B_{i-1}. False positives of B_l are represented as a list.

The encoding of RBAC sessions in cascade Bloom filters that has been proposed [5] is as follows. Let $S = \{s_1, \ldots, s_n\}$ be the set of active sessions at a PEP-SDP, and P be the set of permissions such that $p \in P$ if any $s_i \in S$ is authorized to P. Then, U is $S \times P$, and A is the smaller (in cardinality) of $A_p = \{\langle s, p\rangle : s \text{ is authorized to } p\}$ and $A_n = \{\langle s, p\rangle : s \text{ is not authorized to } p\}$. A bit is maintained with the cascade Bloom filter to indicate which of A_p or A_n is encoded by it. The Bloom filter is a special case of the cascade Bloom filter, with the number of levels, $l = 1$. We adopt the same encoding of RBAC sessions for the Bloom filter as the cascade Bloom filter.

For the example of sessions s_a and s_b for the RBAC policy of Figure 1 from the previous sections, we show an example cascade Bloom filter with 2 levels in Figure 3.

4. A BENCHMARK

In this section, we discuss a benchmark for access-enforcement in RBAC that we have devised. The benchmark has two components: RBAC policies (Section 4.1) and session profiles (Section 4.2). We have designed and implemented programs to generate data sets for the benchmark. The programs are written in Java, and take as input arguments that correspond to the categorizations we discuss in Sections 4.1 and 4.2. We have made the programs available publicly [14].

4.1 RBAC Policies

The RBAC policies that comprise our benchmark are from prior research in RBAC, and experience with RBAC deployments that have been documented in books and the research literature. We present a summary in Table 1. We categorize RBAC policies along the following axes.

Source We have two sources, "Literature," and "Synthetic." By Literature, we mean that we have directly acquired particular kinds of policies from literature that documents research and experience with RBAC. Our sources for these can be classified into three. (1) top-down design of RBAC policies [3, 15, 16, 17], (2) role mining and engineering [18, 19, 20, 21, 22, 23, 24, 25], and, (3) evaluation of approaches to access-enforcement [4, 5, 8]. We also have created some new kinds of policies based on policies from the literature. We call these Synthetic policies.

Number of users, roles and permissions The numbers of users, roles and permissions are typically co-dependant in RBAC policies from the literature. In Table 1, we show the number of users, and the corresponding numbers of roles and permissions for policies from the literature, and for Synthetic policies. We point out that roles do not grow, for example, linearly, with users, but more as a step function.

We point out also that the number of permissions range from a fraction of the number of users, to a somewhat significant multiple. The reason for this range is that RBAC is deployed in one of two contexts. One is for high-level policies in which permissions are abstract. Another is at a much lower level, in which resources that are protected are individual files or email messages; in such systems, there can be a considerable number of permissions. (It is common for a permission to be a pair $\langle o, r\rangle$, where o is the object or resource that is protected, and r is a privilege or right. However, this is not the only encoding as a permission that is meaningful; see, for example, the work of Crampton [26].)

For our Synthetic policies, we consider numbers for typical enterprises that we have not already considered under Literature. The number of employees of an enterprise can be up to 1.6 million as of the writing of this paper [27]. If such enterprises deploy RBAC, we anticipate that they will want to model each employee as an RBAC user. For such policies, we anticipate that the number of roles will be in the same proportion to the number of users as for the largest range for users from the literature. We do not anticipate that the number of permissions will increase significantly. Consequently, we adopt for permissions similar numbers as the largest ranges from the literature.

Role Hierarchy (RH) and connectivity There are three categories we consider for the structure of RBAC policies. As Table 1 indicates, these are RH Depth, RH Model and Connectivity. By RH Depth, we mean the maximum path-length from a role to a permission. In our survey of the literature, the RH Depth does not exceed 5.

We consider two RH Models, Stanford and Hybrid. In the Stanford model [3], roles are layered, and a role at layer i directly inherits roles only in layer $i + 1$, and is inherited directly only by roles in layer $i - 1$ (or by users, for the topmost layer of roles). The Stanford model arises in the top-down design of RBAC policies. Realizing the Stanford model in an enterprise typically results in 4 or 5 layers of roles [3]. The hybrid model arises in both the top-down design of RBAC policies and in role mining. In the hybrid model, the role hierarchy is some partial ordering, and not layered as in the Stanford model. A special case of the two models is when there is no role-role relationship. This is called Core RBAC and arises in role mining [8, 21].

Activation	Access checks
• Intra-session	• Number
– Number of roles	• Nature
– Number of permissions	
– Nature of roles	
– Nature of permissions	
• Inter-session	
– Number of sessions	
– Arrival rate	

Table 2: Session profile categories in our benchmark.

4.2 Session Profiles

There is some prior work which has datasets on session profiles [5, 8, 28]. We augment those datasets with our own. We categorize session profiles into two: activation and access checks; we summarize in Table 2.

Activation Under this category, we consider attributes associated with the activation of a session. We consider both intra- and inter-session attributes. An intra-session attribute is the *number of roles* in the session. For the number, we may specify a constant, or a range. Another attribute is the *nature of roles*. For this attribute, we may specify, for example, that only roles to which a user is directly assigned are activated. Another example is that only roles that do not violate some separation-of-duty condition may be activated [3]. The other two attributes are the *number* and *nature* of permissions.

Source	# Users	# Roles	# Permissions	RH Depth	RH Model	Connectivity
Literature	500-999	10-200	10-3000	1–5	Stanford Hybrid Core	Constant (range) to roles, Constant (range) to permissions, Distributions (e.g., Zipf, uniform)
	1000-1999	200	1000-3500			
	2000-2999	100	100-2000			
	3000-3999	200-250	1500-11000			
	5000-6000	200	1500-2000			
	10000-40000	120-1300	100-11000			
Synthetic	40001-400000	1600-16000	1500-2000			
	400001-1600000	16000-64000	1500-11000			

Table 1: A categorization of RBAC policies in the benchmark.

We have two inter-session attributes. One is the number of sessions that are activated. The other is the arrival rate of sessions. We consider two kinds of arrivals: bursty and uniform. By bursty arrival, we mean that session activations are interspersed with relatively long "quiet" periods in which we have no session activation. In the interim, we have access checks for the sessions that exist. In uniform session arrival, session activations are uniformly interspersed with access checks. We conjecture that bursty arrivals are likely with sessions that are directly used by humans, and uniform arrivals are possible if there are automated processes with which sessions are associated.

Access checks Our second category under session profiles relates to access checks. We have two broad attributes: the *number* and *nature* of access checks. Under number of access checks, we characterize how many access checks are made in the session. Under the nature of access checks, we characterize the permissions for which access checks needs to be made. For example, a session may exercise a large subset of the permissions to which it is authorized, and may do so multiple number of times.

5. ASSESSMENT

In this section, we discuss our assessment of the six approaches using the benchmark that we discuss in the previous section. In Section 5.1, we discuss our methodology for statistically sound data collection and evaluation. In Section 5.2, we discuss our assessment of time efficiency for inter-session attributes. In Section 5.3 we discuss our assessment for intra-session attributes. In Section 5.4, we assess the space efficiency of the approaches, and in Section 5.5, we assess how administratively efficient each approach is.

We have implemented all six approaches in Java; our implementations are available publicly [14]. For CPOL [6] and the (cascade) Bloom filter [5], we have acquired the original implementations. The code for the latter is already in Java, and we have made some minor modifications for our assessment. The original implementation of CPOL is in C++. We have reimplemented it in Java so we have "apples for apples" comparisons with the other approaches. In doing so, we have attempted to adhere to the original implementation as closely as possible. In particular, the manner in which the AccessToken (see Section 3.4) is implemented is crucial to the time-efficiency of CPOL. We observe that the timing measurements we obtain with our re-implementation (see Table 3) are close to those of Borders et al. [6].

5.1 Methodology

Meaningful empirical assessment is a significant challenge in computing [29]. For Java programs, non-determinism in making empirical observations can result from various factors, such as dynamic compilation and garbage collection. The methodology we adopt overcomes such non-determinism and is statistically rigorous. It is based on the work of Georges et al. [30].

Java programs run within an instance of a Virtual Machine (VM). We collect the average time across multiple VM invocations, as there can be variation across such invocations. Within a VM invocation, we need to avoid skew from the effects of starting up the VM and reach what is called steady-state [30]. For each VM invocation, we determine the number of benchmark iterations that we need to perform by finding at least k consecutive steady-state values for which the coefficient of variation (CoV) is less than some preset value (we have chosen 2%). The value of k starts at some value (4, in our case) and increases so long as the CoV decreases, upto the threshold. We record the mean of the k values for each VM invocation. Our final benchmark time is the mean across all VM invocations.

To minimize the effects from garbage collection, we keep the heap size constant across VM invocations. Apart from the mean, we also compute confidence intervals. Our objective is for the confidence intervals to not overlap, as then, with a certain confidence (95%, in our case), we can assert that the two values are statistically distinct. All the values we report and graph in this paper are statistically distinct from other values.

We have conducted our experiments on an isolated Intel dual core E8400 PC that runs at 3 GHz, has 3.5 Gbytes of RAM and runs the Ubuntu Linux operating system. Our Java version is 1.6.0_18, and we run the OpenJDK Runtime Environment.

5.2 Time Efficiency – Inter-Session

We have two inter-session attributes: the number of sessions, and the arrival rate. In Table 3 and Figure 4, we present our results for time efficiency, with the inter-session attributes as parameters. We also consider an intra-session attribute, the nature of RH. We discuss the results that pertain to that in the next section. In each dataset we have 25 users, each authorized to different numbers of roles and permissions. We have 100 roles in total, and 250 permissions. Our objective is to understand the behavior of each approach as the two inter-session attributes change. Conse-

		Direc. graph	Access matrix	Auth. recycl.	CPOL	Bloom filter	Cas. Bl. filter
Bursty	Stanford	32.70	0.79	2928.80	2.14	56.03	18.07
	Hybrid	9.41	0.80	94.07	3.12	60.15	32.40
	Core	5.17	0.74	10.18	2.87	50.41	29.94
Uniform	Stanford	29.47	0.62	1220.43	1.50	53.73	22.00
	Hybrid	8.45	0.62	49.73	1.44	55.57	25.54
	Core	5.93	0.60	4.44	1.51	55.62	26.16

Table 3: Average access check times in μs with the inter-session attributes, and one intra-session attribute (nature of RH), as parameters. The averages are across number of sessions from 2 through 15, for a given RBAC policy that comprises 25 users, 100 roles and 250 permissions. For the Stanford RBAC policy, we have adopted 5 layers, which is the maximum in Table 1. For the Hybrid RBAC policy, the depth varies between 1 to 5. We give the standard deviations for the bursty case in Figure 4.

quently, we consider from 2 through 15 sessions, and both bursty and uniform arrivals for the sessions. There are several observations we make from our results.

Arrival rates We observe from Table 3 that none of the approaches, except authorization recycling, is impacted by the session arrival rate (burst vs. uniform). The reason is that in authorization recycling, all the work is during access checking; session activation does not involve any exchange from the PDP to the SDP (except validation of the initiation). Consequently, authorization recycling can be impacted by bursty session arrival, which results in a number of access checks in a short periods.

Number of sessions The graphs in Figure 4 show the impact of the number of sessions on each approach. We observe that all six approaches are resilient to an increase in the number of sessions from the standpoint of time efficiency. That is, access check time does not necessarily grow with the number of sessions. We expect this to be the case, so long as the PEP/SDP is not stressed by adding too many sessions. None of the approaches has an access check algorithm whose time-complexity is parameterized by the number of sessions.

It is not our objective to stress a PEP/SDP by considering large numbers of sessions. Indeed, the number of sessions a PEP/SDP can support without significant impact on its performance depends on its resources such as hardware. Our objective is gain broader insights into the six approaches, notwithstanding the resources available to a PEP/SDP, assuming some realistic model of computation (the "Random-Access Machine" model, for example [11]).

Efficiency The access matrix is very time-efficient; in our tests, an access check takes less than 1 μs. This is unsurprising as an access check is done in constant time with minimal additional overhead. CPOL is only slightly less efficient; for this particular dataset, we can perceive the number of permissions to which a session is authorized as constant. Consequently, the manner in which a CPOL AccessToken is realized does not impact time-efficiency. The directed graph is highly efficient for Core RBAC. This is because a path from a session vertex to a permission vertex is exactly 2; consequently, it is highly efficient when we have only up to a few hundred roles. The cascade Bloom filter and the Bloom filter are also efficient. The major overhead with them are the computation of the indexing function (in our implementation, this is the cryptographic hash function, SHA-1 [31]), and searching a set in the worst case. Authorization recycling is efficient for Core RBAC, but its performance degrades when we add a hierarchy. The reason is that the first time a permission is accessed, the SDP must communicate with the PDP; this must happen for every permission that is accessed. We study the impact on time efficiency from intra-session attributes (e.g., a large number of permissions in a session) in the next section.

Jitter By jitter, we mean the variation in access check times as the number of sessions changes. We can quantify this as the percentage error in the mean; that is, the ratio of the standard deviation to the mean. We observe from Figure 4 that this is quite small for the directed graph, access matrix and CPOL. It is higher for the cascade and Bloom filter, and very high for the Stanford RH for authorization recycling. The cascade and Bloom filter are affect by the heterogeneity of the permissions; if the union of permissions to which all sessions are authorized is larger, this can result in a deeper cascade or a larger set of false positives that must be maintained explicitly. Authorization recycling is affected by the heterogeneity of the roles that a user may activate in a session. In our datasets, a user is directly assigned to the same number of roles across each of the Stanford, Hybrid and Core policies. Consequently, there is more heterogeneity in the roles that a user may activate in the Stanford policy than in the other two.

5.3 Time-Efficiency – Intra-Session

We have studied the impact of intra-session attributes on time-efficiency. We vary three parameters in our experiments in this context: the number of roles per session, the number of permissions per session and the nature of RH (Stanford, Hybrid and Core). Figures 4 shows the impact of the last attribute on time efficiency, and 5 shows average access check times in μs for Core RBAC, for which the number of roles and permissions range from small (10) to large (10,000). Such numbers are consistent with Table 1.

Role hierarchy Table 3 and the graphs in Figure 4 show the impact of Stanford vs. Hybrid vs. Core as the choice for RH. Only for the directed graph and authorization recycling do we see an impact from the choice of RH. For the directed graph, a deeper RH results in an increased access check time as we need to traverse a longer path from a session vertex to a permission vertex. For authorization recycling, a deeper RH gives a user more choices of roles he may activate. This is reflective of our dataset – a user is directly assigned to the same number of roles for all three of the Stanford, Hybrid

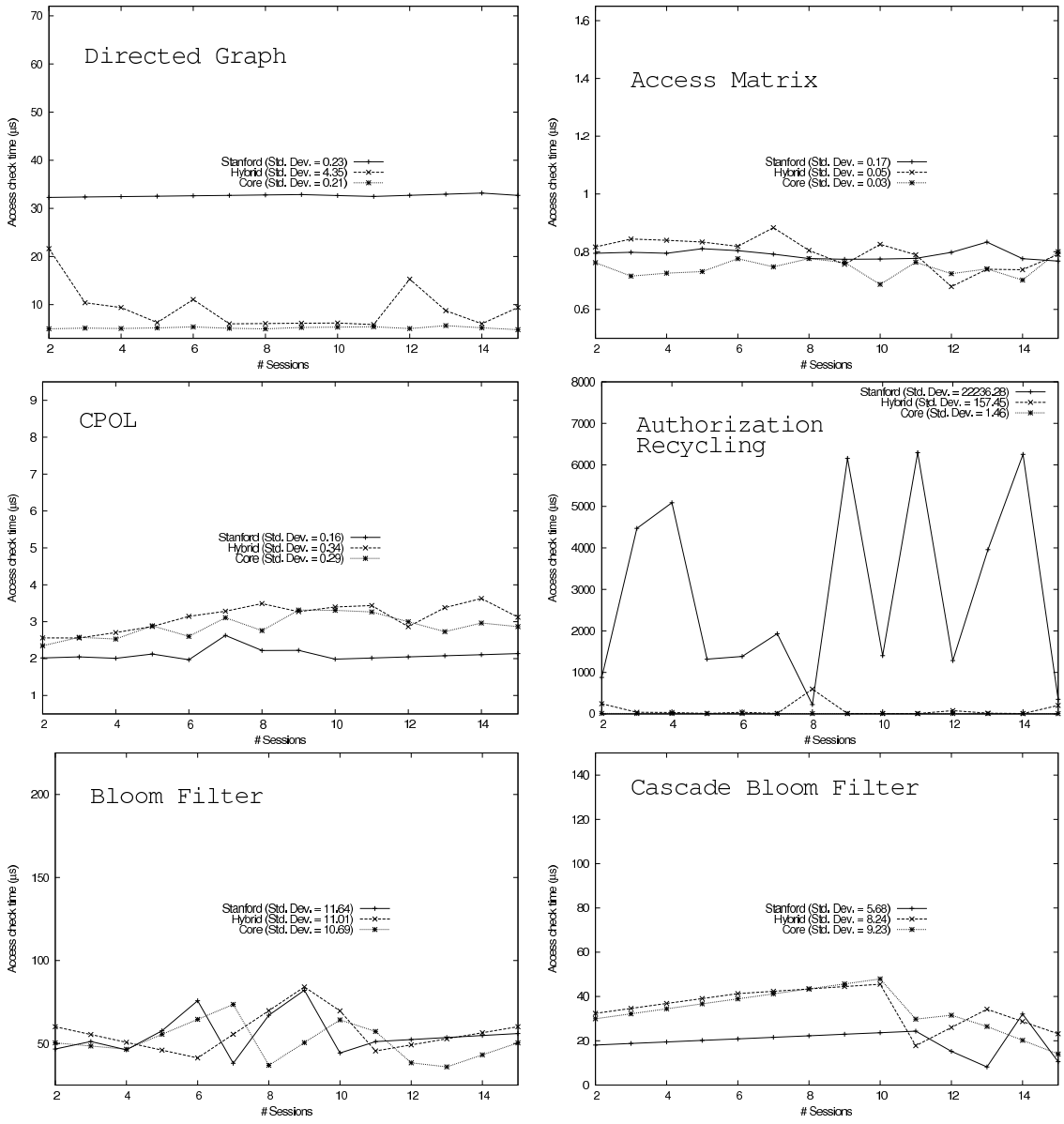

Figure 4: Average access check times in μs and the corresponding standard deviations for our six approaches as the number of sessions changes, for the three different kinds of role hierarchies.

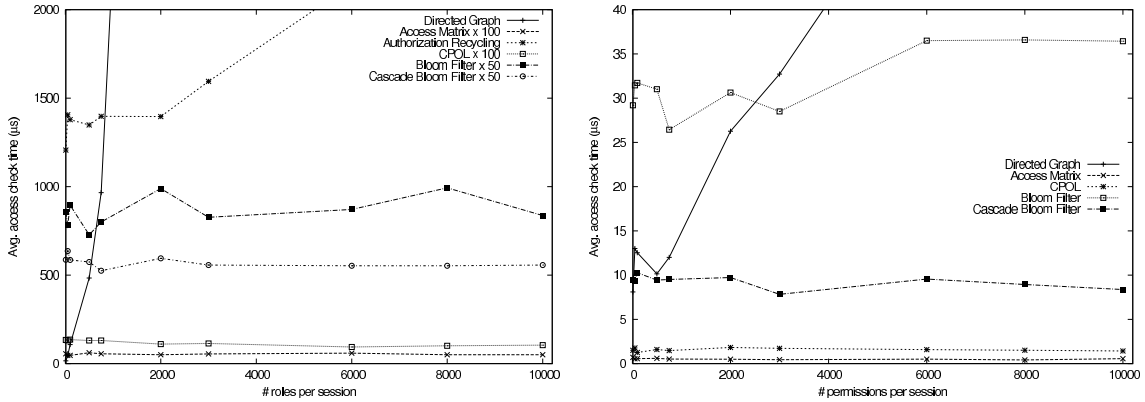

Figure 5: Time efficiency for small (10) to large (10,000) numbers of roles and permissions in a session. In the graph to the right, we do not plot authorization recycling as the numbers are much larger than the ones for the other approaches.

and Core RBAC policies. However, in the Stanford policy, he is authorized to more roles as a result of the deep RH. Consequently, the size of Cache^- and Cache^+ is larger.

Scalability We observe from Figure 5 that the directed graph scales poorly as we increase the number of roles. The reason is that access checking for the directed graph is vertex reachability, which is linear in the size of the graph. Authorization recycling fairs somewhat poorly as we need to cache almost every role for each permission, and this results in it being linear in the number of roles, though with a smaller constant than for the directed graph. For the cascade and Bloom filter, access matrix and CPOL, the time for access checking is independent of the number of roles in a session. In this respect, they scale well with the number of roles.

The cascade and Bloom filter, access matrix and CPOL scale well also with the number of permissions, as Figure 5 indicates. This is somewhat surprising as an AccessToken in CPOL is linear in the number of permissions in a session. As we mention earlier, it is crucial to the time efficiency of CPOL that this encoding be efficient. Also, for the cascade and Bloom filters, the optimal values of the number of indices and the number of indexing functions changes with the number of permissions. (It may decrease for the cascade Bloom filter owing to the negation bit.) Notwithstanding this, upto 10,000 permissions, these issues appear to have no tangible impact on the time efficiency of these approaches. The directed graph fares poorly in this context as well. This is because the adjacency list approach often requires a linear search to find a vertex (permission, in this case). We do not plot authorization recycling in the graph for permissions in Figure 5 as the numbers for it are much higher than for the other approaches. It scales poorly with the number of permissions, as the number of entries in the two caches is linear in the number of permissions in a session.

5.4 Space-Efficiency

In this section, we analyze the space-efficiency of the six approaches. We base our assessment on what we have observed from our implementations, and an analysis of the data structures. The space needed for a directed graph grows linearly with the number of sessions. In the worst-case, it can also grow linearly with the number of permissions and roles per session. However, on average, the size of the directed graph is constant in the number of permissions and roles. This is because we expect roles and permissions to be shared by several sessions.

The access matrix is highly space inefficient. The reason is that it grows quadratically with the number of sessions and the number of permissions to which any session is authorized. CPOL is linear in the number of sessions. It is linear also in the number of permissions per session, and therefore not as space efficient as the directed graph. It is agnostic to the number of roles in a session. Authorization recycling is linear in the number of sessions. However, it can be quadratic in the number of roles and permissions, in the worst case. The reason is that an entry in the Cache^- or Cache^+ is a role set-permission pair.

The Bloom filter and the cascade Bloom filter are non-constant in space relative to the number of sessions and the number of permissions per session. The reason is that the optimal values for the number of indices and the number of indexing functions changes with the number of sessions and permissions. For the cascade Bloom filter, the number of indices may decrease with an increase in the number of session or permissions per session as a consequence of the negation bit (see Section 3.6 and Figure 3). Our code implements the algorithms for insertion and deletion that have been proposed by Tripunitara and Carbunar [5]. Consequently, the relationship between space and the number of sessions or permissions per session is a step function. The cascade and Bloom filter are agnostic to the number of roles per session.

In Figure 6, we present graphs that capture the above discussion. The graphs have been generated based on our implementations. The reason the access matrix is highly space-efficient for small numbers of sessions is that it is a bit matrix. However, as the number of sessions and permissions per sessions grow, its (quadratic) growth quickly negates the fact that each entry in the matrix is only a bit.

5.5 Administrative Efficiency

An administrative change is the addition or deletion of a user-role, role-role or permission-role relationship in an RBAC policy. The addition of a user at the PDP has no impact on an SDP. However, the removal of a user may im-

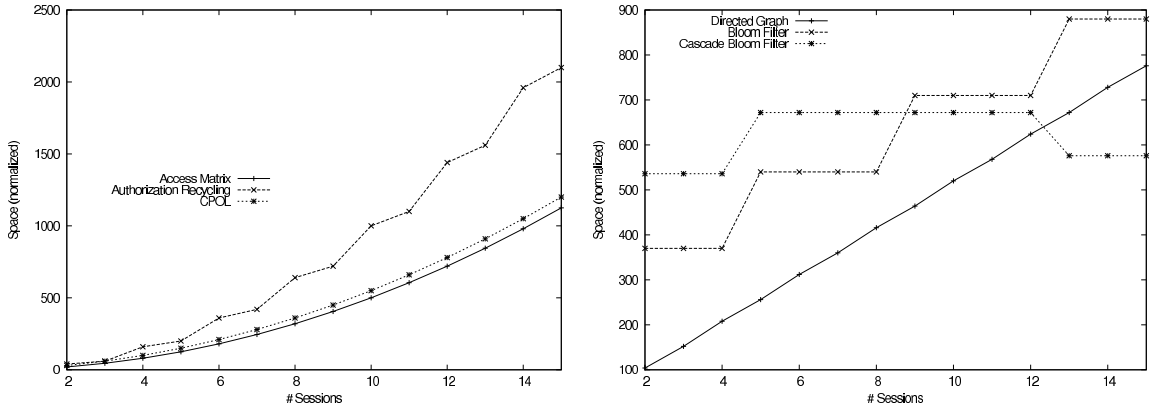

Figure 6: The space efficiency of our approaches. We show the three space inefficient approaches (access matrix, authorization recycling and CPOL) to the left, and the three space efficient approaches (directed graph, cascade and Bloom filter) to the right. In our dataset that we used to generate these graphs, The number of roles and permissions grows by a constant factor per session.

pact an SDP, as that user's sessions need to be removed. This impact is linear in the number of sessions in the worst case for the directed graph, quadratic in the number of sessions and permissions in the worst case for the access matrix, constant-time for authorization recycling, linear in the number of sessions in the worst case for CPOL, and quadratic in the number of sessions and permissions in the worst case for the cascade and Bloom filter.

The addition or removal of a permission can impact an SDP. The impact is constant-time for the directed graph, linear in the worst case in the number of sessions for the access matrix, quadratic in the number of sessions and roles in the worst case for authorization recycling, linear in the number of sessions for CPOL, and linear in the number of sessions in the worst case for the cascade and Bloom filter. The addition or removal of a role can authorize or forbid a session, respectively, to several permissions. We can infer the impact on the six approaches from our discussions on permissions.

In Table 4, we show the results of a proportional mix of administrative changes. The research literature on RBAC administration has focussed mostly on user-role changes, presumably because these are the most frequent in real-world deployments. We assume that 75% of the changes are to user-role relationships. We conjecture that permission-role changes are the next most frequent (20%) and changes to roles are infrequent (5%). In our experiments, sessions overlap with one another in terms of permissions and roles to a constant factor.

6. CONCLUSIONS

We summarize our conclusions in Table 5. For each approach, we rate it "good," "fair" or "poor" along the three axes that we adopt in this paper. For time efficiency, we split our rating into inter- and intra-session. The table allows us to choose the most appropriate approach for an RBAC deployment. For example, if the deployment is small in the size of the RBAC policy (e.g., only up to 100's of roles and permissions), then the access matrix is a good choice. If the deployment is larger, however, space considerations can dominate, and the cascade Bloom filter is a good choice. If there is a need to balance reasonable space and access check time with ease of administration, then the directed graph is a good choice.

Summary We have assessed six approaches to distributed access enforcement in RBAC. Our approach is empirical, and we have proposed and used a benchmark as the basis. Based on our quantitative results, we are able to provide guidance on the best approach from the among the six for particular RBAC deployments.

Future work A validation of our conclusions in real-world RBAC deployments is an important piece of future work. Another issue is whether our approaches can apply to emergent access control models other than RBAC. We can also go back and ask this question in older contexts such as trust management. For example, it will be interesting to assess whether an encoding of trust management systems in the access matrix will give better performance than CPOL [6].

7. REFERENCES

[1] Personal Communication, Open Text Corporation, Aug. 2010.

[2] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, "Role-based access control models," *IEEE Computer*, vol. 29, pp. 38–47, February 1996.

[3] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli, *Role-Based Access Control*. Artech House, Apr. 2003.

[4] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu, "Authorization Recycling in RBAC Systems," in *Proceedings of the 13th ACM Symposium on Access Control, Models and Technologies (SACMAT'08)*, pp. 63–72, 2008.

[5] M. Tripunitara and B. Carbunar, "Efficient Access Enforcement in Distributed Role-Based Access Control (RBAC) Deployments," in *Proceedings of the 14th ACM Symposium on Access Control, Models and Technologies (SACMAT'09)*, pp. 155–164, 2009.

[6] K. Borders, X. Zhao, and A. Prakash, "Cpol: High-performance policy evaluation," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'05)*, pp. 147–157, 2005.

	Direc. graph	Access matrix	Auth. recycl.	CPOL	Bloom filter	Cas. Bl. filter
100	13.45	2934.00	2294.20	321.75	1006.95	2530.05
200	22.20	9003.60	2757.00	1644.00	927.60	5559.30
300	39.15	1741.05	8085.60	5439.30	9760.50	2753.55
400	45.80	3748.80	524.00	5053.00	13755.00	4891.60
500	38.50	25097.25	8781.50	12567.25	3568.00	15980.00
600	87.30	20488.20	3272.10	3492.00	16399.50	7051.80
700	108.85	18676.70	19598.25	1737.05	7383.60	12406.45
800	142.00	33686.40	17520.40	7352.00	7076.80	1967320
900	151.65	17543.25	44167.05	17145.00	33234.30	41891.85
1000	158.50	31068.00	685.00	6800.00	19080.50	32351.50

Table 4: The administrative overhead on a Core RBAC policy. We assume a proportion of 75% changes to user-role relationships, 20% to role-permission relationships, and 5% to role-role relationships. The number of sessions is 1000, and every user has at least one session.

		Direc. graph	Access matrix	Auth. recycl.	CPOL	Bloom filter	Cas. Bl. filter
Time	Inter-session	fair	good	fair	good	fair	fair
	Intra-session	poor	good	poor	good	good	good
Space		fair	poor	poor	poor	fair	good
Admin		good	poor	poor	poor	poor	poor

Table 5: Our rating of "good," "fair" or "poor" for each approach that we assess. While we argue that these ratings follow from our quantitative observations, they are somewhat subjective.

[7] S. Wilson and J. Kesselman, *Java Platform Performance: Strategies and Tactics*. Prentice Hall, May 2000.

[8] Y. Liu, C. Wang, M. Gorbovitski, T. Rothamel, Y. Cheng, Y. Zhao, and J. Zhang., "Core role-based access control: efficient implementations by transformations," *PEPM'06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial Evaluation and semantics-based Program Manipulation*, pp. 112–120, May 2006.

[9] G. S. Graham and P. J. Denning, "Protection — principles and practice," in *Proceedings of the AFIPS Spring Joint Computer Conference*, vol. 40, pp. 417–429, AFIPS Press, May 16–18 1972.

[10] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in operating systems," *Communications of the ACM*, vol. 19, pp. 461–471, Aug. 1976.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. The MIT Press, 3 ed., Sept. 2009.

[12] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[13] L. Fan, P. Cao, J. Almeida, and A. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking*, vol. 8, no. 3, pp. 281–293, 2000.

[14] Marko Komlenovic, Mahesh Tripunitara and Toufik Zitouni, "A platform for assessing approaches to distributed Role-Based Access Control (RBAC) enforcement," 2010. Available from `http://code.google.com/p/dist-rbac-eval/`.

[15] A. Kern, M. Kuhlmann, A. Schaad, and J. Moffett, "Observations on the role life-cycle in the context of enterprise security management," *7th ACM Symposium on Access Control Models and Technologies*, June 2002.

[16] A. Schaad, J. Moffett, and J. Jacob., "The role-based access control system of a european bank: A case study and discussion," *proceedings of ACM Symposisum on Access Control Models and Technologies*, pp. 3–9, May 2001.

[17] A. Kern, "Advanced features for enterprise-wide role-based access control," *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 333–343, December 2002.

[18] D. Zhang, K. Ramamohanarao, S. Versteeg, and R. Zhang., "Rolevat: Visual assessment of practical need for role based access control," *ACSAC*, pp. 13–22, 2009.

[19] J. Vaidya, V. Atluri, and J. Warner, "Roleminer: mining roles using subset enumeration," *Proceedings of the 13th ACM conference on Computer and communications security (CCS'06)*, pp. 144–153, 2006.

[20] D. Zhang, K. Ramamohanarao, T. Ebringer, and T. Yann, "Permission set mining: Discovering practical and useful roles," *ACSAC'08: Proceedings of the 2008 Annual Computer Security Applications Conference*, pp. 247–256, 2008.

[21] I. Molloy, N. Li, T. Li, Z. Mao, Q. Wang, and J. Lobo, "Evaluating role mining algorithms," *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, pp. 95–104, 2009.

[22] C. Blundo and S. Cimato, "A simple role mining algorithm," *Proceedings of the 2010 ACM Symposium on Applied Computing*, pp. 1958–1962, 2010.

[23] M. Frank, A. Streich, D. Basin, and J. Buhmann, "A probabilistic approach to hybrid role mining," *Proc. 16th ACM conference on Computer and Communications Security (CCS)*, pp. 101–111, 2009.

[24] I. Molloy, H. Chen, T. Li, Q. Wang, N. Li, E. Bertino, S. Calo, and J. Lobo, "Mining roles with semantic meanings," *Proc. ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2008.

[25] M. Jafari, A. Chinaei, K. Barker, and M. Fathian, "Role mining in access history logs," *Journal of Information Assurance and Security*, 2009.

[26] J. Crampton, "On permissions, inheritance and role hierarchies," in *Proceedings of the Tenth ACM Conference on Computer and Communications Security (CCS-10)*, pp. 27–31, ACM Press, Oct. 2003.

[27] "Global 500." Fortune Magazine, 2010. Available from `http://money.cnn.com/magazines/fortune/global500/2010/`.

[28] Q. Yao, A. An, E. Terzi, and X. Huang, "Finding and analyzing database user sessions," *Proceedings of the 10th International Conference on Database Systems for Advanced Applications (DASFAA)*, 2005.

[29] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney, "Producing wrong data without doing anything obviously wrong!," in *Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*, pp. 265–276, 2009.

[30] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *Proceedings of OOPSLA'07*, pp. 57–76, May 2007.

[31] F. I. P. Standards, "Secure hash standard," 2002. Available from `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.