Authorization Recycling in Hierarchical RBAC Systems

QIANG WEI, CCB International (Holdings) Ltd.
JASON CRAMPTON, Royal Holloway, University of London
KONSTANTIN BEZNOSOV and MATEI RIPEANU, University of British Columbia

3

As distributed applications increase in size and complexity, traditional authorization architectures based on a dedicated authorization server become increasingly fragile because this decision point represents a single point of failure and a performance bottleneck. Authorization caching, which enables the reuse of previous authorization decisions, is one technique that has been used to address these challenges.

This article introduces and evaluates the mechanisms for authorization "recycling" in RBAC enterprise systems. The algorithms that support these mechanisms allow making precise and approximate authorization decisions, thereby masking possible failures of the authorization server and reducing its load. We evaluate these algorithms analytically as well as using simulation and a prototype implementation. Our evaluation results demonstrate that authorization recycling can improve the performance of distributed-access control mechanisms.

Categories and Subject Descriptors: K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; C.4 [**Performance of Systems**]: *Reliability, availability, and serviceability*

General Terms: Security, Design, Performance, Reliability

Additional Key Words and Phrases: SAAM, RBAC, access control, authorization recycling

ACM Reference Format:
Wei, Q., Crampton, J., Beznosov, K., and Ripeanu, M. 2011. Authorization recycling in hierarchical RBAC systems. ACM Trans. Info. Syst. Sec. 14, 1, Article 3 (May 2011), 29 pages.
DOI = 10.1145/1952982.1952985 http://doi.acm.org/10.1145/1952982.1952985

1. INTRODUCTION

Modern access control solutions [Borders et al. 2005; DeMichiel et al. 2001; Entrust 1999; Karjoth 2003; Netegrity 2000; OMG 2002; Securant 1999; Spencer et al. 1999; Oracle 2008] are based on the request-response model as illustrated in Figure 1. In this model, a policy enforcement point (PEP) intercepts application requests, obtains access control decisions (also known as *authorizations*) from a policy decision point (PDP), and enforces these decisions.

Research on $SAAM_{RBAC}$ by Q. Wei and K. Beznosov has been partially supported by the Canadian NSERC Strategic Partnership Program, grant STPGP 322192-05.
Authors' addresses: Q. Wei, CCB International (Holdings) Limited, 35/F, Two Pacific Place, 88 Queensway, Admiralty, Hong Kong; email: weiqiang@ccbintl.com; J. Crampton, Department of Mathematics, Royal Holloway, University of London, Egham Hill, Egham, Surrey, TW20 0EX, U.K.; email: Jason.Crampton@rhul.ac.uk; K. Beznosov, Department of Electrical and Computer Engineering, University of British Columbia, 4047-2332 Main Hall, Vancouver, BC, V6T 1Z4, Canada; email: beznosov@ece.ubc.ca; M. Ripeanu, Department of Electrical and Computer Engineering, University of British Columbia, 4033-2332 Main Hall, Vancouver, BC, V6T 1Z4, Canada; email: matei@ece.ubc.ca.

DOI 10.1145/1952982.1952985 http://doi.acm.org/10.1145/1952982.1952985

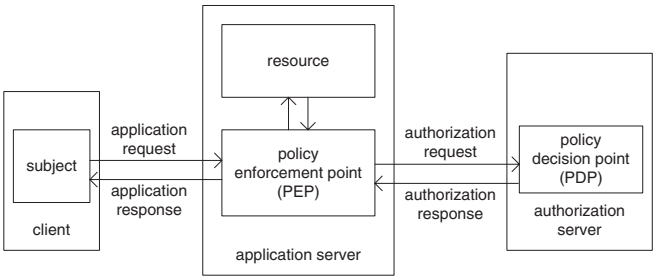


Fig. 1. Access control based on request-response model.

In the large enterprise systems currently deployed, PDPs are commonly implemented as logically centralized authorization servers. This design provides two benefits: consistent policy enforcement across multiple PEPs and reduced administration cost for authorization policies. Like all centralized approaches, however, this architecture has two critical drawbacks: the PDP is a single point of failure and a potential performance bottleneck.

The single point of failure property of the PDP leads to reduced availability: the authorization server may not be reachable due to a failure (transient, intermittent, or permanent) of the network, of the software located in the critical path (e.g., the operating system), of the hardware, or even as a result of a misconfiguration of the supporting infrastructure. A conventional approach to improving the availability of a distributed infrastructure is failure masking through redundancy (either information, time, or physical [Johnson 1996]). However, redundancy and other general-purpose fault-tolerance techniques for distributed systems scale poorly, and become technically and economically infeasible when the number of entities in the system reaches thousands [Kalbarczyk et al. 2005; Vogels 2004]. At the same time, large-scale commodity computing is becoming a reality, with eBay having 12,000 servers and 15,000 application server instances [Strong 2007], and Google estimated to have "more than 450,000 servers spread in at least 25 locations around the world" [Markoff and Hansell 2006].

In a massive-scale enterprise system with nontrivial authorization policies, making authorization decisions is often computationally expensive due to the complexity of the policies involved and the large size of the resource and user populations. Thus the centralized PDP often becomes a performance bottleneck [Nicomette and Deswarte 1997]. Additionally, the communication delay between the PEP and the PDP can make the authorization overhead prohibitively high.

The state-of-the-practice approach to improving overall system availability and reducing the authorization processing delays observed by the client is to cache authorizations at each PEP—what we refer to as *authorization recycling*. Existing authorization solutions commonly provide PEP-side caching [Borders et al. 2005; Entrust 1999; Oracle 2008; Netegrity 2000; Spencer et al. 1999]. These solutions, however, only employ a simple form of authorization recycling: a cached authorization is reused only if the authorization request in question exactly matches the original request for which the authorization was made. We refer to such reuse as *precise recycling*.

To improve authorization system availability and reduce delay, we previously proposed the Secondary and Approximate Authorization Model (SAAM) [Crampton et al. 2006]. SAAM adds a secondary decision point (SDP) to the request-response model, as shown in Figure 2. The SDP is collocated with the PEP and can resolve authorization requests not only by precise recycling but also by computing *approximate authorizations* from cached authorizations. SAAM is independent of the specifics of the application

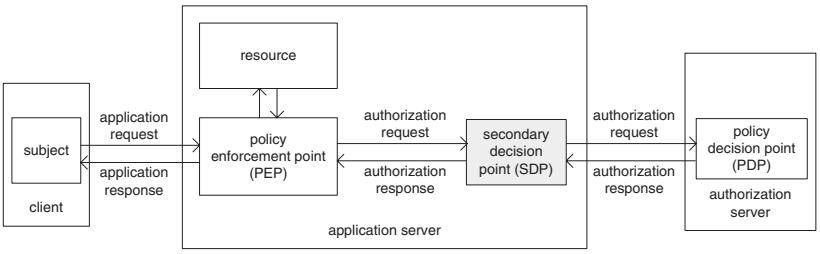


Fig. 2. SAAM adds a secondary decision point (SDP) to the request-response model.

and access control policy. For each class of access control policies, however, specific algorithms for inferring approximate responses need to be designed.

This article proposes $\text{SAAM}_{\text{RBAC}}$—the SAAM authorization recycling algorithm for role-based access control (RBAC) model. Introduced more than a decade ago, RBAC [Ferraiolo and Kuhn 1992; Sandhu et al. 1996] has been deployed in many organizations for access control enforcement and eventually matured into the ANSI RBAC standard [ANSI 2004]. In RBAC, instead of directly assigning permissions to users, the users are assigned to roles and the roles are mapped to permissions. A role normally represents the organizational position that is responsible for certain job functions. Users are assigned roles according to their qualifications. Permissions are a set of many-to-many relations between objects and operations. Roles describe the relationship between users and permissions through user-to-role assignment (UA) and permission-to-role assignment (PA). Our inference algorithm uses this structure to infer approximate authorizations for new requests.

Compared to approaches that proactively pull or push the entire PA set (or even the whole policy) to each SDP, our approach—based on on-demand caching of authorization responses—offers two advantages. First of all, if the working set of the PEP is a significantly smaller subset of the whole policy, it may well be the case that the SDP cache is "warm" enough and hence able to answer a significant proportion of authorization requests more quickly than the PDP (since the cache size is significantly smaller than the whole RBAC policy and the SDP is collocated with the PEP). Thus, depending on application workload and deployment scenario, our approach offers the possibility of rapid response times without the need for large caches. Second and more importantly, our approach allows for the PEP and PDP remain unchanged. That is, the middleware used for PEP-to-PDP communications can be reconfigured in such a way that the SDP is interposed between the PEP and PDP. As a result, the SDP can act as a PDP's proxy for the PEP, without requiring any modification at the PEP or at the PDP. This means that one can retrofit existing authorization systems with our SDP without changing PEPs or PDPs. For this purpose, dynamic weaving [Schroder-Preikschat et al. 2006] or other existing techniques, such as meta-objects [Astley et al. 2001], for automatically generating custom RPC stubs are readily available. Those RBAC systems that already employ SDPs for *precise* recycling are even more amenable to being retrofitted with the $\text{SAAM}_{\text{RBAC}}$ *aproximate* recycling logic proposed in this article.

To evaluate properties of $\text{SAAM}_{\text{RBAC}}$ algorithms, we used an experimental testbed with 100 subjects, 3000 permissions, and 50 roles. The evaluation results demonstrate an 80% increase, compared to precise recycling, in the number of authorization requests that can be served without consulting the access control policies stored remotely at the PDP. These results suggest that deploying $\text{SAAM}_{\text{RBAC}}$ improves the availability and scalability of RBAC systems, which in turn improves the performance of the enterprise systems.

To summarize, we make four technical contributions. First, we define inference rules specific to RBAC authorization semantics and develop recycling algorithms from these rules. Second, we suggest how modification to these algorithms in order to support hierarchical RBAC. Third, we develop algorithms that update SDP cache to handle different types of policy changes. Finally, we study deployment strategies of our algorithms to achieve different performance-related goals.

The rest of this article is organized as follows. Section 2 presents background, including SAAM and RBAC. Section 3 describes $\text{SAAM}_{\text{RBAC}}$ design. Section 4 reports results of evaluating $\text{SAAM}_{\text{RBAC}}$. Section 5 discusses related work. We conclude in Section 6.

2. BACKGROUND

This section provides background on SAAM and ANSI RBAC that is necessary for understanding the rest of the article.

2.1. Secondary and Approximate Authorization Model

SAAM [Crampton et al. 2006] is a general framework for making use of cached PDP responses to compute *approximate responses* for new authorization requests. An authorization request is a tuple (s, o, a, c, i), where s is the subject, o is the object, a is the access right, c is the contextual information relevant to the request, and i is the request identifier. Two requests are *equivalent* if they only differ in their identifiers. An authorization response is a tuple (r, i, E, d), where r is the response identifier, i is the corresponding request identifier, d is the decision, and E is the evidence. The evidence can be used in some SAAM implementations to aid the response verification.

SAAM also defines primary, secondary, precise, and approximate authorization responses. A *primary* response is a response made by the PDP, whereas a *secondary* response is produced by an SDP. A response is *precise* if it is a primary response to the request in question or a (secondary) response to an *equivalent* request. Otherwise, if the SDP infers the response based on primary responses to other requests, the response is *approximate*.

In general, the SDP infers approximate responses based on cached primary responses and any information that can be deduced from the application request and system environment. The larger the number of cached responses, the more information is available to the SDP, and the SDP will become a better and better PDP simulator.

We say an SDP is *safe* if any request it allows would also be allowed by the PDP [Crampton et al. 2006]. A safe SDP returns either undecided or deny for any request for which it cannot infer an allow response. A safe SDP can be configured or designed to implement a closed world policy[1] by simply denying any request that it cannot evaluate. More generally, we allow the SDP to return an undecided response; it is then up to the PEP to decide how such a response should be handled. In most cases, the PEP will deny the request, thereby "failing safe"—one of the important principles identified by Saltzer and Schroeder [1975]. We say an SDP is *consistent* if any request it denies would also be denied by the PDP.

In general, one would wish to implement a safe and consistent SDP, which returns the same response as the PDP would have for any request that it can evaluate. Clearly, any SDP that only returns precise decisions—by only returning responses for equivalent requests for which decisions have been cached—is safe and consistent. However, such an SDP is rather limited. SAAM seeks to extend the functionality of the SDP so that it can generate approximate responses and remain safe and consistent. However, the limitations of the underlying access control policy, time or space complexity of the

[1]A closed world policy allows a request if there exists an allow response for it, and denies it otherwise.

inference algorithms, or business requirements could limit an SDP implementation to being either safe or consistent, but not both.

2.2. Role-Based Access Control

There are a number of RBAC models in the literature, including RBAC96 [Sandhu et al. 1996] and the ANSI RBAC standard [ANSI 2004]. All such models assume the existence of a set of users U, a set of roles R, and a set of permissions P. They also assume the existence of a user-to-role assignment relation $UA \subseteq U \times R$ and a permission-to-role assignment relation $PA \subseteq P \times R$. A user u is authorized for a permission $p \in P$ if there exists a role $r \in R$ such that $(u, r) \in UA$ and $(p, r) \in PA$.

Many models also assume the existence of a role hierarchy RH, which is modeled as a partial order on the set of roles. That is, $RH \subseteq R \times R$, where RH is reflexive, anti-symmetric, and transitive. It is customary to write $r \leqslant r'$ rather than $(r, r') \in RH$. In this case, u is authorized for p if there exist roles $r, r' \in R$ such that $(u, r) \in UA$, $r \geqslant r'$, and $(p, r') \in PA$.

An important innovation in RBAC96 and ANSI RBAC is the concept of *sessions*. A user initiates a session (typically when authenticating to the system) by activating some subset of the roles to which he is assigned. Access requests are evaluated in the context of the session that initiates the request. A request for permission p is granted if the user session contains a role r and there exists a role r' such that $r \geqslant r'$ and $(p, r') \in PA$.

3. $\text{SAAM}_{\text{RBAC}}$

$\text{SAAM}_{\text{RBAC}}$ applies SAAM concepts to RBAC systems. In a system using $\text{SAAM}_{\text{RBAC}}$, the SDP caches authorization requests and the corresponding authorization decisions, and computes new authorization decisions based on the cache when the PDP is unable to make a timely decision. As these decisions are not obtained from the PDP, they are by necessity *secondary*. In this section we present the algorithms that can be employed by an SDP in the context of RBAC systems. We show that an SDP that implements these algorithms will make safe and consistent secondary decisions.

3.1. Assumptions

In general, we assume that the PDP is the only component that has access to the entire authorization policy and the SDP is not aware of the policy. In particular, we assume that the SDP does not have direct access to the permission-to-role assignment relation (PA). It is the job of the SDP to try to "reconstruct" PA on the basis of information that can be inferred from primary responses to previous requests. If we relieve this assumption, for example, the PDP is able to "push" the entire policy to the SDP, then the SDP may compute a precise response using the same authorization logic as the PDP. However, pushing entire policy is rarely done in enterprise-grade deployments due to the limitations of the underlying security protocols, the scale of the authorization policies, the administrative constraints, or the cost of keeping up to date user, attribute, and permission data at multiple SDPs. Providing SDP with no direct access to PA also enables transparently interposing SDP between PEP and PDP without having to modify the protocol between the two. This is an important practical benefit when it comes to retrofitting existing authorization systems with SDP-like components.

We further assume that authorization requests made to the SDP (and the PDP) include the set of roles, this information being supplied by the PEP. This role information arrives at the PEP in a number of different ways [Lorch et al. 2003]. First, it can be "pushed" from the client's security subsystem, as in CORBA [OMG 2002], DCE [Gittler and Hopkins 1995], SESAME [Kaijser 1998], and GAA API [Ryutov and Neuman 2000], where the security attributes are a part of the security context of the

application request. Second, it can also be "pulled" by the PEP from external attribute services such as LDAP or the Shibboleth Attribute Authority [Internet2 2008].

For most of Section 3, we assume that the SDP does not have access to the role hierarchy relation and does not try to reconstruct hierarchical relationships between roles. In Section 3.8, we drop this assumption and show how our approach needs to be modified when the SDP is aware of the role hierarchy structure.

3.2. Preliminaries

We must first consider how to map SAAM notions of subject and request to appropriate RBAC concepts. The notion of session is important in RBAC96 and ANSI RBAC for implementing the principle of least privilege [Saltzer and Schroeder 1975]: by activating a strict subset of the roles to which she is authorized, a user may limit the privileges that she can exercise while interacting with a computer system. It is a session that is synonymous with a subject in identity-based access control systems, since access decisions are made on the basis of the permissions that are available to the activated roles. Accordingly, $\text{SAAM}_{\text{RBAC}}$ models a subject as a set of roles.

The Core Specification of ANSI RBAC, similarly to RBAC96, defines two functions that map sessions to users and roles: $session\_users(s : SESSIONS) \rightarrow USERS$ and $session\_roles(s : SESSIONS) \rightarrow 2^{ROLES}$. $\text{SAAM}_{\text{RBAC}}$ abstracts a subject as a set of roles activated in a given session. In the terms of above functions, a subject in $\text{SAAM}_{\text{RBAC}}$ is an output of $session\_roles(\dots)$. Therefore, if user u started two sessions s_1 and s_2, they are treated as two separate—and possibly unrelated—subjects in $\text{SAAM}_{\text{RBAC}}$, unless same roles are activated for both of these sessions. On the other hand, if another user u' started session s_3, and the sets of activated roles in s_2 and s_3 are equal, then $\text{SAAM}_{\text{RBAC}}$ algorithms do not distinguish between the corresponding subjects. Furthermore, in $\text{SAAM}_{\text{RBAC}}$, we do not consider the relationship between users and their sessions.

RBAC96 treats permissions as "uninterpreted symbols," because such entities are very likely to be application- and context-specific. However, ANSI RBAC defines permissions to be object-operation pairs. It seems appropriate to regard a SAAM request (s, o, a, c, i) and an RBAC request (s, p, c, i) as equivalent, where $p = (o, a)$.

A response (r, i, E, d) indicates the decision to a request (s, p, c, i). For simplicity, we introduce the following conventions that will be used in the remainder of the article: we omit c and i from requests; we omit r, E and i from responses; we write $+(s, p)$ to denote a grant decision for request (s, p), and $-(s, p)$ to denote a deny decision. More specifically, $+(s, p)$ means that there exists role $r \in s$ such that $(p, r) \in PA$ and $-(s, p)$ means that there does not exist such an r. We also write $X - Y$ to denote the set $\{x \in X : x \notin Y\}$.

3.3. Inference Rules

Using the notation from the previous section, we first note the following rules that can be applied to generate approximate responses.

—$\mathbf{Rule}^{+}$. If $+(s, p)$ and $s' \supseteq s$, then request (s', p) should be granted.
—$\mathbf{Rule}^{-}$. If $-(s, p)$ and $s' \subseteq s$, then request (s', p) should be denied.

$\mathbf{Rule}^{+}$ follows from the fact that if some permission p is granted for the set of roles s, then there exists $r \in s$ such that r is authorized for p, and $r \in s'$ for any $s' \supseteq s$. $\mathbf{Rule}^{-}$ follows from the fact that if p is denied for the set of roles s, then there does not exist $r \in s$ such that r is authorized for p; trivially, no subset of s will be authorized for p.

In the following subsections, we first present naive algorithms and show that they are suboptimal in terms of success rate and space. Then we define canonical form of cache and describe algorithms that work over such cache.

3.4. Naive Algorithms

We construct two relations $Cache^+ \subseteq 2^R \times P$ and $Cache^- \subseteq R \times P$ to generate approximate responses. The basic idea is to use primary deny responses to build $Cache^-$ and primary allow responses to build $Cache^+$.

Cache construction. Whenever the SDP receives a deny response $-(s, p)$, the pair (r, p) is added to $Cache^-$ for every role $r \in s$ (since we know that no role in s can be authorized for p). In contrast, whenever the SDP receives an allow response $+(s, p)$, the pair (s, p) is added to $Cache^+$.

Request evaluation. Then to evaluate a request (s, p), the SDP first checks whether s contains a role r such that $(r, p) \notin Cache^-$. (If not, no role in s is authorized for p and the SDP denies the request.) Then the SDP checks whether there exists $(s', p) \in Cache^+$ such that $s \supseteq s'$. If so, then the SDP allows the request and otherwise the SDP returns undecided. The algorithm to evaluate request (s, p) is summarized below.

(1) Let $s^+ = \{r \in s : (r, p) \notin Cache^-\}$
(2) If $s^+ = \emptyset$, then deny (every role in s was not authorized for p)
(3) Else
  (a) If there exists $(s', p) \in Cache^+$ such that $s \supseteq s'$ then allow
  (b) Else undecided

PROPOSITION 1. *An SDP that implements the above request evaluation algorithm is safe and consistent.*

PROOF. Consider the response produced by the SDP for request (s, p). If the SDP produces a deny response then, for all $r \in s$, there exists $(r, p) \in Cache^-$. This means that the PDP must have generated a number of responses of the form $-(s_1, p), \ldots, -(s_k, p)$, $k \geqslant 1$, such that for all $r \in s$, $r \in s_i$ for some i. Hence, the PDP would also deny (s, p).

If the SDP produces an allow response then there exists $(s', p) \in Cache^+$ such that $s \supseteq s'$. Hence, the PDP would allow request (s, p), since it must have allowed (s', p). □

The naive algorithms, however, may return undecided responses for some requests that would be allowed by the PDP. Suppose that $(\{r_1, r_2, r_3\}, p) \in Cache^+$ and $(r_3, p) \in Cache^-$. Now the evaluation of request $(\{r_1, r_2, r_4\}, p)$ with the above algorithm returns undecided because $\{r_1, r_2, r_4\} \not\supseteq \{r_1, r_2, r_3\}$. However, $\{r_1, r_2, r_4\} \supset \{r_1, r_2\}$ and hence request $(\{r_1, r_2, r_4\}, p)$ can safely be authorized. The optimized algorithms we present in the following sections correct this problem.

3.5. Cache Compression

We have seen that the naive method of constructing the cache may not optimize the "hit rate"—the proportion of requests for which the SDP can provide a definitive answer. We now define the *canonical form* of the cache.

Definition 1. Given a cache, $Cache = (Cache^+, Cache^-)$, we say $Cache$ is in *canonical form* if the following conditions hold

(1) if for all $(s, p) \in Cache^+$, there does not exist $r \in s$ such that $(r, p) \in Cache^-$;
(2) for all distinct $(s, p), (s', p) \in Cache^+$, $s \not\subseteq s'$ and $s \not\supseteq s'$.

The first of the two requirements above ensures that all roles of a subject s that are known *not* to be authorized for a permission are removed from s. The second requirement simply ensures that there is no redundancy in the cache: it makes no difference to the allow responses returned by the request evaluation algorithm; in other words, it minimizes the amount of storage required for $Cache^+$.

Input: response q
1C: $AddResponse(q)$
2C: **if** $q = -(s, p)$ **then**
3C: replace each $(s^+, p) \in Cache^+$ with $(s^+ - s, p)$
4C: **if** $(s^-, p) \in Cache^-$ **then**
5C: replace it with $(s \cup s^-, p)$
6C: **else**
7C: add (s, p) to $Cache^-$
8C: **end if**
9C: **else** // we know that $q = +(s, p)$
10C: **if** there exists $(s^+, p) \in Cache^+$ such that $s^+ \subseteq s$ **then**
11C: **return**
12C: **end if**
13C: find $(s^-, p) \in Cache^-$
14C: delete all $(s^+, p) \in Cache^+$ such that $s - s^- \subseteq s^+$
15C: add $(s - s^-, p)$ to $Cache^+$
16C: **end if**

(a) The cache construction algorithm

Input: request (s, p)
1D: $EvaluateRequest(s, p)$
2D: find $(s^-, p) \in Cache^-$
3D: $d \leftarrow s - s^-$
4D: **if** $d = \emptyset$ **then**
5D: **return** deny
6D: **else**
7D: **for all** $(s^+, p) \in Cache^+$ **do**
8D: **if** $s^+ \subseteq d$ **then**
9D: **return** allow
10D: **end if**
11D: **end for**
12D: **return** undecided
13D: **end if**

(b) The decision algorithm

Fig. 3. $\text{SAAM}_{\text{RBAC}}$ optimized recycling algorithms.

Cache compression improves the hit rate. In particular, we claim the following statements hold.

(1) *If* $Cache^+$ *satisfies property* (2) *but does not satisfy property* (1) *then the hit rate is not optimal*.
(2) *If* $Cache^+$ *does not satisfy property* (2) *then the size of the cache is not minimal, that is, there exists a smaller cache that provides the same hit rate.*
(3) *If the cache is in canonical form, then any smaller cache has a lower hit rate.*

PROOF OF CLAIM 1. Suppose that $Cache^+$ does not satisfy property (1). Then there exists $(s, p) \in Cache^+$ such that $r \in s$ and $(r, p) \in Cache^-$. Now request (s', p), where $s' \supseteq s - \{r\}$, is authorized since r is not authorized for p. However, $s' \not\supseteq s$ and by assumption $Cache^+$ satisfies property (2) so there does not exist $s'' \subseteq s$ such that $(s'', p) \in Cache^+$. Hence, the SDP cannot resolve request (s', p). Hence, the hit rate is not optimal. □

PROOF OF CLAIM 2. Suppose that $(s, p), (s', p) \in Cache^+$ and $s' \supseteq s$. Then (s', p) is authorized and any request (s'', p), where $s'' \supseteq s'$, is authorized because $s'' \supset s$. Hence (s', p) may be omitted from $Cache^+$. □

PROOF OF CLAIM 3 Suppose now that $(s, p) \in Cache^+$ and there does not exist $(s', p) \in Cache^+$ such that $s' \supset s$ or $s' \subset s$. Then request (s, p) is authorized when the SDP uses $Cache^+$ but is not authorized if we remove (s, p) from $Cache^+$ (since, by assumption, there is no $s' \subset s$ such that $(s', p) \in Cache^+$, we cannot find an entry $(s', p) \in Cache^+$ such that $s \supseteq s'$). In other words, omitting (s, p) from $Cache^+$ will decrease the hit rate. □

3.6. Optimized Algorithms

We now present optimized algorithms that produce a canonical form of the cache in order to improve the likelihood of the evaluation algorithm returning an allow response. Henceforth, we write $Cache^- \subseteq 2^R \times P$, making it consistent with the representation of $Cache^+$. Naturally, the meaning of $(s, p) \in Cache^-$ is that all roles in s are known not to be authorized for p. The full algorithm (**C**) for constructing compressed cache relations is shown in Figure 3(a). To satisfy property (1) of the canonical form definition, in line 3C, which handles negative primary responses, we delete any roles in s from sets of roles that had previously been authorized for p (that is, tuples in $Cache^+$).

Table I. Building $Cache^+$ and $Cache^-$ from Primary Responses

Response	$Cache^+$	$Cache^-$
$-(\{r_1, r_2\}, p)$		$(\{r_1, r_2\}, p)$
$+(\{r_2, r_3, r_4\}, p)$	$(\{r_3, r_4\}, p)$	$(\{r_1, r_2\}, p)$
$+(\{r_4, r_5, r_6\}, p)$	$(\{r_3, r_4\}, p)$, $(\{r_4, r_5, r_6\}, p)$	$(\{r_1, r_2\}, p)$
$-(\{r_4, r_7\}, p)$	$(\{r_3\}, p)$, $(\{r_5, r_6\}, p)$	$(\{r_1, r_2, r_4, r_7\}, p)$

Analogously, in line 15C, which handles positive primary responses, we delete any roles from s that are known not to be authorized for p. To satisfy property (2) of the canonical form definition, line 10C is used to prevent any superset of existing roles in $Cache^+$ from being added and line 14C is used to prune redundant tuples from $Cache^+$.

Figure 3(b) shows the decision algorithm (**D**) for generating an approximate response, which follows directly from rules $\mathbf{Rule}^+$ and $\mathbf{Rule}^-$ (Section 3.3) and the construction of the cache. Since s may include roles that are known not to be authorized for p, we remove those roles first and then see whether the remaining roles are authorized for p. In other words, given request (s, p), we first find $(s^-, p) \in Cache^-$ (line 2D) and compute those roles in s that are not in s^- (line 3D), namely, $s - s^-$. If this set is empty, then we know that all roles in s are in s^-; that is, $s \subseteq s^-$ and the request should be denied (by $\mathbf{Rule}^-$). Otherwise, we need to check whether there is a tuple $(s^+, p) \in Cache^+$ such that $s^+ \subseteq (s - s^-)$.

The following example shows how the optimized algorithms work. Suppose $Cache^-$ and $Cache^+$ are empty and the following primary responses are obtained from the PDP:

$$-(\{r_1, r_2\}, p),\ +(\{r_2, r_3, r_4\}, p),\ +(\{r_4, r_5, r_6\}, p),\ -(\{r_4, r_7\}, p).$$

Table I illustrates how $Cache^-$ and $Cache^+$ develop as these responses are processed by the SDP. Notice how r_4 is removed from both tuples in $Cache^+$ once the primary deny response $-(\{r_4, r_7\}, p)$ is processed.

Note also that the final contents of $Cache^-$ and $Cache^+$ are independent of the order in which primary responses are received. If, for example, we reverse the order of the last two responses, we find that r_4 is added to $Cache^-$ a step earlier and that r_4 does not appear with r_5 and r_6 in a tuple in $Cache^+$.

Now suppose we wish to generate secondary responses for the following requests: (1) $(\{r_3, r_4\}, p)$, (2) $(\{r_1, r_4, r_7\}, p)$, (3) $(\{r_1, r_5\}, p)$.

—The SDP returns an allow response for request (1) because $(\{r_3\}, p) \in Cache^+$.
—The SDP returns a deny response for request (2) because $(\{r_1, r_2, r_4, r_7\}, p) \in Cache^-$.
—The SDP returns an undecided response for request (3).

It is worth noting that although the SDP does not explicitly store primary responses, it will always return the same response as the PDP for any requests whose decisions have been included in the cache relations. More formally, we have the following result.

PROPOSITION 2. *Suppose the PDP has produced a response for request* (s, p)*. Then an SDP that implements the construction and decision algorithms in Figure 3 will produce the same response as the PDP for request* (s, p)*.*

PROOF. First note that lines 3C and 15C imply that, if $(t^-, p) \in Cache^-$ and $(t^+, p) \in Cache^+$, then $t^- \cap t^+ = \emptyset$.

Given that the PDP has produced a response, there are two possibilities to consider. If the PDP produced an allow response for (s, p), then $(s^+, p) \in Cache^+$ for some $s^+ \subseteq s$, by construction of $Cache^+$. If there does not exist $(s^-, p) \in Cache^-$ then we are done. Otherwise, consider $d = s - s^-$ (as computed in line 3D). We claim that $d \supseteq s^+$ and hence

the SDP will return an allow response. To establish the above claim, consider $r \in s^+$. Then $r \in s$ since $s^+ \subseteq s$. Now $s^+ \cap s^- = \emptyset$ and $r \in s^+$. Hence, $r \notin s^-$ and $r \in s - s^- = d$.

Conversely, if there exists a primary deny response for (s, p), then $(s^-, p) \in Cache^-$ for some $s^- \supseteq s$, by construction of $Cache^-$. Hence $s - s^- = \emptyset$ and the SDP will return a deny response (line 5D). □

LEMMA 1. *An SDP that implements the construction and decision algorithms is safe and consistent.*

PROOF. We need to show that if the SDP produces a conclusive (i.e., not undecided) secondary response for request (s, p), then that response is the one that would be produced by the PDP.

Suppose that the SDP produces the response $-(s, p)$. Then there exists $(s^-, p) \in Cache^-$ such that $s \subseteq s^-$ (by line 5D). Moreover, for each $r \in s^-$, r is not authorized for p, by construction of $Cache^-$. Hence, the PDP would return $-(s, p)$.

Suppose that the SDP produces the response $+(s, p)$. Then there exists $(s_1^+, p) \in Cache^+$ such that $s \supseteq s_1^+$, which implies the existence of a primary response $+(s_2^+, p)$ with $s_2^+ \supseteq s_1^+$. This implies the existence of $r \in s_2^+$ such that r is authorized for p. Moreover, the construction of $Cache^-$ and $Cache^+$ implies that $r \in s_1^+$. Hence $r \in s$, since $s \supseteq s_1^+$ and $r \in s_1^+$, and the PDP would return $+(s, p)$. □

3.7. Discussion

We now briefly and informally discuss the expected behavior of the SDP algorithms. In Section 4, we describe the experimental work we undertook to evaluate the actual behavior.

Suppose p is assigned to roles $r_1, \ldots, r_k$, and that there are n users $u_1, \ldots, u_n$ with u_i assigned to roles $s_i \subseteq R$. Now a user u_i may request p using a subject comprising any subset of s_i. In principle, therefore, $Cache^-$ may contain (s^-, p), where $s^- \subseteq R - \{r_1, \ldots, r_k\}$, and $Cache^+$ may contain (s^+, p), where $s^+ \subseteq s_i$ for some i.

3.7.1. Secondary Response Rate. Let us suppose that $(s_1^+, p), \ldots, (s_m^+, p) \in Cache^+$ and $(s^-, p) \in Cache^-$. Then the probability that our SDP can produce an approximate response (a *hit*) is the probability of it returning either allow or deny. Clearly, the smaller $s_1^+, \ldots, s_m^+$ are, the greater the chance of an allow response, because allow responses require the subject to be a superset of an element in $Cache^+$. Conversely, the larger s^- is, the greater the chance of a deny response are, because allow responses require the subject to be a subset of an element in $Cache^-$.

In short, the probability of a hit increases as the size of s^- increases and the sizes of s_i^+ decrease. It can be seen from the construction algorithm that the effect of processing a primary response (whether it is an allow or deny response) is to either increase the size of s^- or decrease the size of s_i^+ (or both). In other words, increasing the cache size will increase the hit rate.

It is worth noting that it is advantageous to have negative primary responses in the cache, because these affect both $Cache^+$ and $Cache^-$. If there have only been allow primary responses, then $Cache^- = \emptyset$ and hits can only be obtained from secondary allow responses.

For a cache of fixed size, it is advantageous to have s^- large and s_i^+ small. It is easy to see that s^- will be large if the number of roles to which p is assigned is small and there have been a large number of requests for p that have been denied (by the PDP). One way to ensure small s_i^+ is by assigning each user to a small number of roles.

Alternatively, we are likely to get a hit if there is a significant amount of overlap between the sets of roles assigned to different users. This situation arises when each

user is assigned to a significant fraction of the available roles or when some roles are more popular than others so that many users are assigned to those roles. In summary, we would expect the probability of a hit (the *hit rate*) to increase when users are assigned to a small number of roles, or to a significant proportion of the roles available, or to a similar set of roles due to the uneven role assignment. We sought to confirm these expectations by experiment, the results of which are reported in Section 4.1.2.

3.7.2. Performance Considerations. Clearly, the number of tuples in $Cache^-$ is bounded by $|P|$, while the number of tuples in $Cache^+$ is bounded by $|P|\,2^{|R|}$. For a request (s, p), a secondary deny response can be computed in time proportional to $|s| \log |R|$, as we simply need to determine whether s is a subset of the roles contained in s^-. Therefore, the number of primary deny responses is unlikely to have a significant effect on performance. However, the time taken to compute a secondary allow response grows with the number of primary allow responses.

The time taken by the construction algorithm to process a primary response is proportional to the size of $Cache^+$. In the case of a deny response, it is necessary to check each tuple in $Cache^+$ and remove any roles that formed part of the denied request (line 3C, Figure 3). In the case of an allow response, we check to see whether each tuple has been made redundant by the new information (line 14C, Figure 3).

However, we note that the existence of redundant tuples in $Cache^+$ does not compromise the ability of the SDP to compute correct secondary responses, although it may degrade the response time. Therefore, we could periodically purge $Cache^+$ of redundant tuples, rather than delete them as new primary responses are added, thereby improving the processing time for primary allow responses.

In summary, it is easier to incorporate new primary allow responses into the cache rather than deny responses, but it is harder to produce secondary allow responses than deny responses. We investigate these aspects in Section 4.1.4.

3.8. Using the Role Hierarchy in $SAAM_{RBAC}$

When flat RBAC is employed, the binding of a session to a set of roles is trivial: the session is associated with the roles activated by the user. However, in hierarchical RBAC, there are two possibilities, which we call *prerequest* and *postrequest* session-to-role binding. We assume that a user initiates a session $s \subseteq R$ by selecting some subset of the roles to which she is assigned. The set of permissions for which the session is authorized is determined by the permissions assigned to the roles in s and to any roles in R that are junior to at least one role in s. It is this set of roles, therefore, that should be used to evaluate requests, not simply s. More formally, let $\downarrow s$ denote $\{r' \in R : \exists r \in s, r' \leqslant r\}$. Then the evaluation of a request originating from session s requires the computation of the permissions for which the roles in $\downarrow s$ are authorized.

Prerequest binding occurs when the user authenticates. The user u first activates a set of roles s for which she is authorized: that is, for all $r \in s$, there exists $r' \geqslant r$ and $(u, r') \in UA$. The authentication service uses the role hierarchy to compute $\downarrow s$, which is then bound to each process associated with session s. This set of roles forms part of the application request that is passed to the PEP. Clearly, prerequest binding means that the computation of all roles associated with a session is performed once, which means that request evaluation should be quicker. Postrequest binding occurs when the PDP evaluates an access request. In this case, the PDP has to compute $\downarrow s$ before querying the PA relation.

In the case of prerequest binding, neither the PDP nor the SDP need be aware of the role hierarchy. Hence, we only need to consider what we should do if postrequest binding is employed. So far, we have assumed that the SDP is unaware of the role hierarchy.

```
Input: request (s, p)
1D′: EvaluateRequest(s, p)
2D′: find (s⁻, p) ∈ Cache⁻
3D′: d ← s − s⁻
4D′: if d = ∅ then
5D′:     return deny
6D′: else
7D′:     for all (s⁺, p) ∈ Cache⁺ do
8D′:         if s⁺ ⊆ ↓d then
9D′:             return allow
10D′:        end if
11D′:    end for
12D′:    return undecided
13D′: end if
```

Fig. 4. The decision algorithm in a hierarchical setting.

As an optimization, let us now assume that the SDP is aware of the structure of the role hierarchy, and examine how the $\text{SAAM}_{\text{RBAC}}$ algorithms need to be modified.

We first note that the SDP could perform postrequest binding in exactly the same way as the PDP would. However, we observe that it is not necessary to do this when checking $Cache^-$. To see this, suppose that $(s^-, p) \in Cache^-$ and request (s, p) is received by the SDP. We can check whether $s \subseteq s^-$, as before. Moreover, if $s \subseteq s^-$, then no role belonging to $\downarrow s$ can be authorized for p either (otherwise, some role in s, and hence s^-, would be authorized for p). Hence, it suffices to compute $\downarrow s$ only if the request is not denied. The revised decision algorithm is shown in Figure 4. Notice the use of $\downarrow d$, where $d = s - s^-$, in line 8D′ of Figure 4. Also note that $\downarrow d$ can be computed in polynomial time.[2]

3.9. Handling Policy Changes

An enterprise authorization system must support changes to security policies. If the access control policy changes and the SDP is not updated accordingly, the SDP may make incorrect decisions. Policy changes in an RBAC system occur as a result of changes to one of U, P, UA, PA, R, or RH. Changes to U, P, or UA do not affect the cache construction and decision-making algorithms. Hence, we only consider changes to PA, R, and RH.

The first type of change involves modification of PA. In particular, we considered the following two basic cases.

—A permission p is assigned to a role r, that is, (p, r) is added to PA. If the cache is not updated, the SDP may return incorrect negative decisions for some requests for p. Specifically, if $(s, p) \in Cache^-$ and $r \in s$, then any request (s', p) such that $s' \subseteq s$ and $r \in s'$ will be denied despite the fact that r is now authorized for p. To avoid this situation, r needs to be removed from $(s, p) \in Cache^-$. Moreover, $(\{r\}, p)$ should be added to $Cache^+$.

—A permission p is revoked from a role r, that is, (p, r) is removed from PA. If the cache is not updated, the SDP may make false positive decisions, because it may compute allow decisions to those requests that are denied by the PDP. To avoid this, we replace $(s, p) \in Cache^-$ (if it exists) with $(s \cup \{r\}, p)$, or add $(\{r\}, p)$ to $Cache^-$ otherwise. Moreover, we delete every $(s, p) \in Cache^+$ such that $r \in s$. This is because we cannot assume that any of the remaining roles in s are authorized for p.

[2]More specifically, it can be computed in time proportional to the total number of edges and vertices in the role hierarchy using a simple modification to a standard graph traversal algorithm.

Input: policy update response q

1U$_{\text{PA}}$: $UpdateCache(q)$
2U$_{\text{PA}}$: **if** $q = -(\{r\}, p)$ **then**
3U$_{\text{PA}}$: remove those $(s^+, p) \in Cache^+$ for which $r \in s^+$
4U$_{\text{PA}}$: **if** $(s^-, p) \in Cache^-$ **then**
5U$_{\text{PA}}$: replace it with $(s^- \cup \{r\}, p)$
6U$_{\text{PA}}$: **else**
7U$_{\text{PA}}$: add $(\{r\}, p)$ to $Cache^-$
8U$_{\text{PA}}$: **end if**
9U$_{\text{PA}}$: **end if**
10U$_{\text{PA}}$: **if** $q = +(\{r\}, p)$ **then**
11U$_{\text{PA}}$: find $(s^-, p) \in Cache^-$
12U$_{\text{PA}}$: **if** $r \in s^-$ **then**
13U$_{\text{PA}}$: replace it with $(s^- - \{r\}, p)$
14U$_{\text{PA}}$: **end if**
15U$_{\text{PA}}$: delete all $(s^+, p) \in Cache^+$ such that $r \in s^+$
16U$_{\text{PA}}$: add $(\{r\}, p)$ to $Cache^+$
17U$_{\text{PA}}$: **end if**

(a) The cache update algorithm when PA is changed

Input: the role r which is to be removed

1U$_{\text{R}}$: $UpdateCache(r)$
2U$_{\text{R}}$: **for all** p in $Cache^-$ **do**
3U$_{\text{R}}$: find $(s^-, p) \in Cache^-$
4U$_{\text{R}}$: **if** $r \in s^-$ **then**
5U$_{\text{R}}$: replace it with $(s^- - \{r\}, p)$
6U$_{\text{R}}$: **end if**
7U$_{\text{R}}$: **end for**
8U$_{\text{R}}$: **for all** p in $Cache^+$ **do**
9U$_{\text{R}}$: delete all $(s^+, p) \in Cache^+$ such that $r \in s^+$
10U$_{\text{R}}$: **end for**

(b) The cache update algorithm when a role is removed from R

Fig. 5. Cache update algorithms.

The full algorithm for updating the cache relations to deal with updates to PA is shown in Figure 5(a). Comparing it with the cache construction algorithm (Figure 3(a)), we note that there are two main differences. First, if p is revoked from r, it is not sufficient to remove r from each tuple in the $Cache^+$; instead, all tuples in $Cache^+$ that contain r need to be removed (Figure 5, line 3U$_{\text{PA}}$). Second, if p is assigned to r, we add $(\{r\}, p)$ to $Cache^+$ (Figure 5, line 16U$_{\text{PA}}$) and also delete r from the set of roles in $Cache^-$ (Figure 5, line 13U$_{\text{PA}}$), since we know that r is authorized for p.

PA changes can be signaled to the SDP by passing "artificial" responses to it. For example, when (p, r) is added to PA, the SDP can be sent response $+(\{r\}, p)$. These responses are "artificial" in the sense that they are not generated as a result of a genuine request. In order to distinguish them from normal primary responses, we call them *policy update responses*. When the SDP receives a policy update response, it will invoke the cache update algorithm (shown in Figure 5(a)), rather than the cache construction algorithm.

We note that adding $(\{r\}, p)$ to $Cache^+$ may not be necessary but it is a desirable optimization step for two reasons. First, having many tuples of the form $(\{r\}, p)$ in $Cache^+$ will lead to a higher hit rate since more sessions will be a strict superset of an entry in $Cache^+$. Second, it helps remove redundancy from $Cache^+$ as shown in line 15U$_{\text{PA}}$ of Figure 5.[3] In an extreme case, while all permissions are being added to PA from scratch and the cache is updated using the cache update algorithm, $Cache^+$ will increasingly resemble PA. However, due to the limited size of cache storage and the large size of PA, it is unlikely that the SDP will eventually store the whole PA in the cache. By using some cache replacement algorithm, for example, the least-frequently used (LRU) algorithm, SDP is able to keep a small but most-used portion of PA in the cache.

The second type of changes that we considered involves modification of R, in particular, when a role r is removed from R. Assuming that users cannot start a session that includes deleted role(s), keeping r in the cache will not affect the correctness of the

[3]Note line 15U$_{\text{PA}}$ is used to remove redundancy from $Cache^+$: as for the construction algorithm, this step may be omitted and $Cache^+$ periodically purged of redundant tuples instead.

responses that the SDP makes, but will degrade the performance of the SDP. Therefore, it is still desirable to purge the cache of those roles.

The full algorithm for updating the cache relations to deal with updates to R is shown in Figure 5(b). Unlike the previous cache update algorithm, this algorithm must consider all tuples containing r in both $Cache^-$ and $Cache^+$. Therefore, this change may result in a large number of tuples being removed from the cache. Like the previous algorithm, all tuples in $Cache^+$ that contain r need to be removed, because we can not assume that any of the remaining roles in the tuple are authorized for p.

Third, we consider those changes that involve modification of RH. No support for changes in RH is needed if *prerequest* binding is used. When *postrequest* binding is used, the SDP needs to be updated with the new RH so that the computation of $\downarrow d$ is correct when a request is evaluated (line 8D′ in Figure 4).

PROPOSITION 3. *A safe and consistent SDP that implements the cache update algorithm is still safe and consistent after a policy change.*

PROOF. We need to show that, after a policy change, if the SDP produces a secondary response for request (s, p), then that response is the one that would be produced by the PDP after the same policy change.

First, we consider the case when (p, r) is added to PA. For any request (s, p) such that $r \notin s$, the policy change has no effect on the decisions returned by the PDP and SDP. We now consider the case when $r \in s$. Clearly, the SDP will return allow for all such requests, since $(\{r\}, p)$ was added to $Cache^+$ as a result of the cache update. Equally, the PDP will return allow for such requests since $(p, r) \in PA$ as a result of the policy change.

Second, we consider the the case when (p, r) is removed from PA. As above, we need only consider the case when $r \in s$. If the SDP returns an allow decision then there exists $(s^+, p) \in Cache^+$ such that $s^+ \subseteq s$ and $r \notin s^+$. Hence, there exists some role $r' \in s^+$ that is authorized for p. Since the only change to PA was to remove the authorization for role r, we may infer that the PDP would also allow request (s, p), since $r' \in s$. If the SDP returns a deny decision then $s \subseteq s^- \cup \{r\}$. In other words, no role in $s^- \cup \{r\}$ is authorized for p, and now that (p, r) has been removed from PA, the PDP will also return deny.

Third, we consider the case when role r is removed from R. No new session will contain role r and $Cache^-$ and $Cache^+$ are able to decide fewer requests.

—Suppose first that (s, p) was allowed by the SDP and the PDP before the removal of r.
Now if $(s-\{r\}, p)$ is denied by the SDP after the removal of r, then $(s^-, p) \in Cache^-$ and $r' \in s^-$ for all $r' \in s - \{r\}$, which in turn implies that no role in $s - \{r\}$ is authorized for p and the request would also be denied by the PDP. If, however, $(s - \{r\}, p)$ is allowed by the SDP after the removal of r, then there exists $(s^+, p) \in Cache^+$ such that $s - \{r\} \supseteq s^+$ and hence it would be allowed by the PDP.
(There are also requests that may be allowed by the SDP before the removal of r, but cannot be decided after. However, these requests are irrelevant to the definitions of safety and consistency.)
—Suppose now that (s, p) was denied by the SDP and the PDP before the removal of r. Then $(s - \{r\}, p)$ will be denied after r's removal. Hence, any request that is denied by the SDP after r's removal will be denied by the PDP. □

3.9.1. Propagating Policy Changes. An important question to answer is how to propagate update messages to SDPs. We provided in Wei et al. [2007] a detailed discussion on the alternatives for propagating update messages and a solution for implementing well-defined semantics for policy updates. In what follows, we briefly describe our solution.

We first state our assumptions relevant to the access control systems. We assume that the PDP makes decisions using an access control policy stored persistently in a policy store of the authorization server. In practice, the policy store can be a policy database or a collection of policy files. We further assume that security administrators deploy and update policies through the policy administration point (PAP), which is consistent with the XACML architecture [Committee 2005]. To avoid modifying existing authorization servers and maintain backward compatibility, we further add a policy change manager (PCM), collocated with the policy store. The PCM monitors the policy store, detects policy changes, and informs the SDPs about the changes.

Based on the fact that not all policy changes are at the same level of criticality, we divide policy changes into three types: critical, time-sensitive, and time-insensitive changes. By discriminating policy changes according to these types, system administrators can choose to achieve different consistency levels. In addition, system designers are able to provide different consistency techniques to achieve efficiency for each type. Our design allows a $\text{SAAM}_{\text{RBAC}}$ deployment to support any combination of the three types. In the rest of this section, we define each type of policy change and discuss the consistency properties.

Critical changes of authorization policies are those changes that need to be propagated urgently throughout the enterprise applications, requiring immediate updates on all SDPs. When an administrator makes a critical change, our approach requires that he also specifies a time period t for the change. PCM will attempt to propagate the policy change by contacting all SDPs involved, and must within period t either inform the administrator that the change has been successfully performed or provide a list of SDPs that have not confirmed the change. In the latter case, administrators might want to resort to out-of-band means of flushing caches of the unconfirmed SDPs by, for example, restarting them. To support critical changes, SDPs would have to implement algorithms in Figure 5 and PCM would have to "push" changes to SDPs, which requires adding SDP-PCM communication channel. Support for two other types of policy changes is less intrusive, however.

Time-sensitive changes in authorization policies are less urgent than critical ones but still need to be propagated within a known period of time. When an administrator makes a time-sensitive change, it is the PCM that computes the time period t during which caches of all SDPs are guaranteed to become consistent with the change. As a result, even though the PDP starts making authorization decisions using the modified policy, the change becomes in effect throughout the $\text{SAAM}_{\text{RBAC}}$ deployment only after time period t. Notice that this does not necessarily mean that the change itself will be reflected in the SDPs' caches by then, only that the caches will not use responses invalidated by the change.

We suggest using time-to-live (TTL) approach for processing time-sensitive changes. Every primary response is assigned a TTL that determines how long the response should remain valid in the cache, for example, 1 day, 1 h, or 1 min. The assignment can be performed by either the SDP, the PDP itself, or a proxy, through which all responses from the PDP pass before arriving to the SDPs. The choice depends on the deployment environment and backward compatibility requirements. Every SDP periodically purges from its cache those responses whose TTL elapses.

The TTL value can also vary from response to response. Some responses (say, authorizing access to more valuable resources) can be assigned a smaller TTL than others.

When the administrator makes a *time-insensitive change*, the system guarantees that all SDPs will *eventually* become consistent with the change. No promises are given, however, about how long it will take. Support for time-insensitive changes is necessary because some systems may not be able to afford the cost of, or are just not

willing to support, critical or time-sensitive changes. A simple approach for supporting time-insensitive change is for system administrators to periodically flush SDPs caches.

3.10. Implementation Considerations

To facilitate the integration with existing access control systems, the SDP should provide the same policy evaluation interface to its PEP as the PDP, thus enabling SAAM incremental deployment without any change to existing PEP or PDP components. Similarly, in systems that already employ authorization caching but do not use SAAM, the SDP can offer the same interface and protocol as the existing cache component.

SAAM may be deployed for a variety of performance-related reasons, depending on the specific application, geographic distribution, and network characteristics. These reasons will typically include one or more of the following: to reduce the overall load on the PDP; to minimize the delay in responding to the client; and to minimize the network traffic generated by the authorization service. We now discuss two alternative ways for managing the interactions between the PEP, the SDP, and the PDP. These strategies lead to different performance characteristics. Hence, different performance-related priorities can be realized by choosing different deployment strategies.

The first strategy is *concurrent authorization* by the SDP and the PDP. When the SDP receives an authorization request from the PEP, it forwards the request to the PDP. While waiting for a decision from the PDP, it also computes a decision locally. The SDP then returns to the PEP the first conclusive decision it receives or computes. The use of concurrent authorization reduces system response time but increases load on the PDP. Alternatively, we may use *sequential authorization*. The SDP only forwards the request to the PDP if it cannot decide the request. The use of sequential authorization reduces network traffic and load on the PDP, at the cost of increased response time as observed by the PEP. The evaluation of these two strategies is presented in Section 4.2.

4. EXPERIMENTAL EVALUATION

While the previous section describes $\text{SAAM}_{\text{RBAC}}$ algorithms and estimates their complexity, this section presents an experimental evaluation of those algorithms. We used both simulation and a prototype for evaluation. The simulation enabled us to study the algorithms by hiding the complexity of underlying communication, while the prototype enabled us to study the system performance in a more dynamic and realistic environment.

4.1. Simulation-Based Evaluation

In the simulation-based evaluation, we studied three performance aspects of our algorithms: the achieved hit rate, the impact of policy changes on the hit rate, and the computational cost.

First, we studied the *hit rate*, which we define to be the ratio between the number of requests resolved by the SDP (regardless of the specific allow/deny decision) and the total number of requests received. A high hit rate has the effect of masking transient PDP failures, thus improving the overall authorization system's availability. It also reduces the load on the PDP, thus improving the system's scalability, and the authorization system response time.

Our informal analysis in Section 3.7 suggested that the hit rate is influenced by the following factors: (1) the *cache warmness* (the ratio between the number of authorization responses cached at the SDP and the number of possible requests); (2) the percentage of deny responses in the cache at a fixed cache warmness; (3) the characteristics of the RBAC policy, including the ratios between the numbers of users, permissions, and roles in the system; and (4) the popularity distribution of roles. Section 4.1.2 presents results of our experiments investigating the impact of these factors on the hit rate.

The second performance aspect we studied was the impact of *policy changes* on the hit rate. We wanted to understand how the algorithms for handling policy changes (Figure 3) affected the hit rate. Section 4.1.3 presents the experiment results.

The third performance aspect we investigated was the *computational cost* of the SDP algorithms. We measured two types of computational cost: the *inference time*—the time that the SDP takes to infer an approximate response (allow or deny) using its cache; and the *update time*—the time that the SDP takes to incorporate a new primary response in its cache. In particular, the lower the inference time, the more efficient the SDP is in accelerating the access control system. As cache warmness appears to be the main factor influencing performance, Section 4.1.4 presents the influence of cache warmness on the inference and update time.

4.1.1. Experimental Setup. To conduct the experiments, we implemented $\text{SAAM}_{\text{RBAC}}$ recycling algorithms and integrated the implementation with the SAAM evaluation engine used in Crampton et al. [2006]. Each run of the evaluation involved two stages.

The first stage was to create the data input files that were required for the simulation. The engine first created an RBAC policy and assigned roles to both users (UA) and permissions (PA). Second, the engine created the warming set and testing set, which were simply lists of requests. Each request was made up of a subject and a permission. The *warming set* was a pseudorandom permutation of all possible requests, while the *testing set* was a random sampling of requests.

In the second stage, the simulation engine started operating by alternating between *warming* and *testing* modes. In the warming mode, the engine used a subset of the requests from the warming set, evaluated them using a simulated PDP, and sent the responses to the $\text{SAAM}_{\text{RBAC}}$ SDP to build up the cache. During this phase, the evaluation engine also recorded the time required to add primary responses to the cache. Once the desired cache warmness was achieved, the engine calculated the average update time and then switched into the testing mode during which the SDP cache remained constant. We used this mode to evaluate the hit rate and the inference time at controlled, fixed levels of cache warmness. The engine submitted requests from the testing set, and recorded the inference time. Once all the requests in the testing set had been submitted, the engine calculated the hit rate as the ratio of the testing requests resolved by the SDP to all test requests and the average inference time, and then switched back to the warming mode. These two modes were then repeated for different levels of cache warmness, from 0% to 100% in increments of 5%.

For all experiments we used a Linux machine with two Intel Xeon 2.33-GHz processors and 4 GB of memory. The evaluation framework ran on Sun's 1.5.0 Java Runtime Environment (JRE). Each experiment was run ten times and the average results are reported.

We assumed for simplicity that a user always activated all her roles. This assumption allowed us to describe the entire request space more easily because we could assume then that the request space was defined by the set of users and the set of permissions rather than the set of permissions and the set of all subsets of any set of roles for which some user was authorized. However, we do not believe that this assumption had a detrimental impact on our results. Indeed, our choice was likely to mean that the hit rate was lower than might be expected if users were to use subsets of their authorized roles. The reason for this is due to the fact that smaller role sets in subjects mean that (1) the likelihood of a negative response is increased, which increases the hit rate, and (2) the size of role sets in $Cache^{+}$ may be reduced, which means that the chance of a hit is also increased.

The reference RBAC policy used in our experiments contained 100 users, 3000 permissions, and 50 roles. Thus the overall size of the request space and the warming set

was 300,000. The testing set contained 20,000 unique requests which were randomly selected from the request space. For simplicity, we only considered the flat RBAC model. Each assigned role was randomly selected from R. The probability of a given user being assigned to a given role was 0.1. Hence the number of roles assigned to a user was binomially distributed with mean 5 and variance 4.5, and the number of users to which a role was assigned was binomially distributed with mean 10 and variance 9. Similarly, the probability of a given permission being assigned to a given role was 0.04.

While the scale of the system we studied was limited by the computational resources available, we believe that the values of these parameters are not important in themselves. We were interested in configuring a reasonably large system that would manifest a behavior asymptotically similar to possible real-world deployments. Additionally, we studied the impact of varying the number of users, roles per user, roles, and roles per permission as well as the popularity distribution of roles on system's performance. We note that, while the overall number of permissions in the system may influence the response time as a large number of permissions leads to less efficient memory use by the SDP, it will not influence the achieved hit rate.

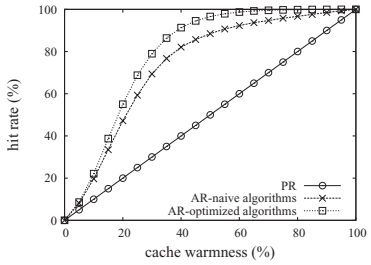
4.1.2. Evaluating Hit Rate. We first studied the hit rate for the reference RBAC configuration. Figure 6(a) presents the hit rate as a function of *cache warmness* for both approximate recycling and precise recycling with the reference policy. As expected, the hit rate of approximate recycling (AR in the figure) increased with cache warmness and was always higher than that of precise recycling (PR in the figure). In addition, the results demonstrate that optimized recycling algorithms achieved a better hit rate than naive recycling algorithms. The improvement was relatively small because it was only due to the increase in secondary allow responses.

Figure 6(b) compares the cache size of the naive and optimized approximate recycling algorithms. The results demonstrate that the optimized algorithms help reduce the cache size significantly. Specifically, using the optimized algorithms, the cache size stabilized at about 600 kB after cache warmness reached about 20%. Using the naive algorithms, however, the cache size kept increasing with the cache warmness, and eventually reached about 1700 kB. The reason is that optimized algorithms maintain the cache in canonical form. In the rest of our evaluation, we used the optimized algorithms for all the experiments.

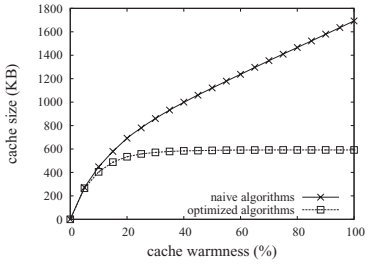
We then studied the impact of varying *the number of users* while the other configuration parameters were fixed. Figure 6(c) shows the *percentage increase* for the hit rate compared with precise recycling for an RBAC system that had 50, 100, and 200 users, respectively. As expected, an increase in the number of users increased the chance that a role-permission pair was already cached thus leading to a higher hit rate. When averaged over the full range of cache warmness, the percentage increase was 36%, 80%, and 132% for 50, 100, and 200 users, respectively.

For the experiments described in the rest of this section, we fixed the cache warmness and studied the impact of other system characteristics on the achieved hit rate. We chose to explore hit rate for relatively low cache warmness values as this is the region where we estimated the system would be most likely to operate due to workload characteristics, limited storage space, or frequently changing access control policies.

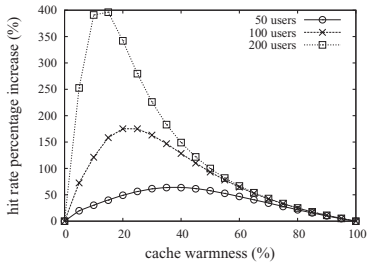
First, we studied the impact of the percentage of deny responses in the cache. In some systems, users may know what they are allowed to do, or the user interface may even hide unauthorized actions from users. Hence, the cache may contain more primary allow responses than primary deny responses. To study this effect in the experiment, we engineered the warming set so that the PDP could generate a specified proportion of deny responses. Figure 6(d) confirms our prediction that a higher proportion of deny responses leads to a higher hit rate. The intuition behind this result is that a negative



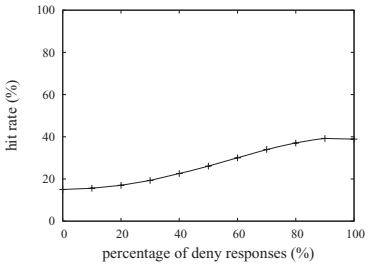
(a) Hit rate as a function of cache warmness



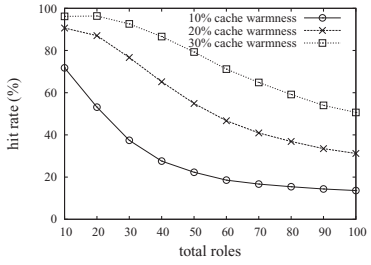
(b) Cache size as a function of cache warmness



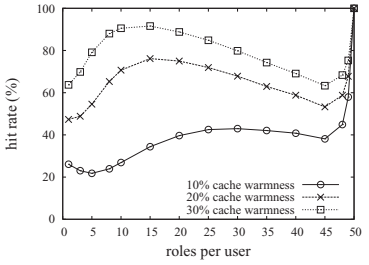
(c) Hit rate percentage increase as the SDP cache warmness varies, for 50, 100, and 200 users in the RBAC system.



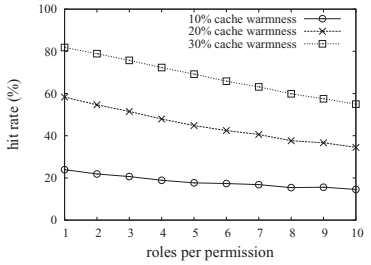
(d) Hit rate variation with the percentage of primary deny responses in the SDP cache for 15% cache warmness.



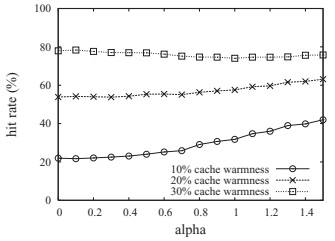
(e) Hit rate variation with the total number of roles in the RBAC system



(f) Hit rate variation with the mean number of roles per user



(g) Hit rate variation with the mean number of roles per permission



(h) Hit rate variation with the coefficient α in Zipf distribution.

Fig. 6. The impact of various system characteristics on the hit rate.

primary response for a permission and a user means that the permission is not assigned to *any* of the user's roles. In contrast, a positive primary response only allows us to infer that the permission is assigned to at least one of the roles, but without the ability to infer exactly which role. Note that we only show the results for 15% cache warmness because the maximum cache warmness we could reach by using only allow responses was less than 20%.

Second, we studied the impact of the total number of roles on the hit rate by varying it from 10 to 100 (Figure 6(e)) and keeping constant the number of users and the mean number of roles a user/permission is assigned to. The results indicate that, as the number of roles increases, the hit rate decreases. This confirms our intuition that, as the number of roles increases, the overlap between the sets of roles each user is assigned to also decreases thus reducing the likelihood of a successful inference.

Third, we studied the impact of the mean number of roles to which each user is assigned by varying it from one to all the roles the system (50 roles) while keeping all other parameters constant. The results in Figure 6(f) suggest that the influence of this parameter on the hit rate is more complex. We now describe our understanding of these curves. The hit rate was low when each user was assigned to few (fewer than five) roles because there were few roles in each entry of $Cache^+$ and $Cache^-$ and hence the chances of making an approximate response were limited. As the number of roles per user increased, the size of the entries of the role sets in the cache increased and the chance of two users' role sets overlapping increased. While the overlap was still relatively low (when each user was assigned to fewer than 10 roles), the deny responses dominated the content of the SDP cache. However, when the number of roles per user increased further, $Cache^+$ started increasing at the expense of $Cache^-$, leading to the decrease in the hit rate (as we predicted in Section 3.5). Moreover, for entries of the form $(s, p) \in Cache^+$, s was likely to be large (since there were few deny responses to reduce their size). Since subjects contained all roles assigned to a user and users were assigned to a large number of roles, it became difficult to generate an allow secondary response for (s, p), because s was large and our approach requires a tuple $(s', p) \in Cache^+$ such that $s' \subseteq s$, and in such tuples s' was also likely to be large. Less intuitive is the sharp increase to 100% in the hit rate on the right side of the graph. This increase was likely due to the fact that each user was assigned to (almost) all the roles in the system and, as a result, (almost) every user had the same set of roles. In practice, we would expect the number of roles to be a relatively small compared to the number of users (e.g., Schaad et al. [2001] found it to be around 3–4%) and that users would be allocated to a small fraction of those roles. Our experimental results suggest that the characteristics of real RBAC systems will not compromise the efficacy of our algorithms.

Fourth, we studied the impact of the mean number of roles to which each permission was assigned. Figure 6(g) confirms the results of our analytical analysis, which predicted that a larger number of roles per permission leads to a lower hit rate. This effect can be also attributed to the decrease of $Cache^-$.

Finally, we studied the impact of role popularity distribution. In all our previous experiments, roles were uniformly assigned to users and permissions so that all roles were equally "popular" in UA and PA relations. However, in reality some roles may be assigned to users or permissions more frequently than other roles. For example, in an enterprise most users are assigned an "employee" role while only a few are assigned a "manager" role. To model this type of highly uneven popularity, we used a Zipf distribution.

Zipf distributions have been widely used to model heterogeneous popularity distributions (e.g., Web page popularity [Breslau et al. 1999], Web site popularity [Adamic and Huberman 2002], and query term popularity [Klemm et al. 2004]). A set of data obeys Zipf's law if the frequency of an item is inversely proportional to (some nonnegative)

power of its rank (determined by frequency of occurrence). More formally, suppose we have a frequency distribution $(x_1, f_1), \ldots, (x_n, f_n)$, where data item x_i occurs f_i times and $f_1 \geqslant f_2 \geqslant \cdots \geqslant f_n$. Then the distribution obeys Zipf's law if $f_i \propto \frac{1}{i^\alpha}$ for some $\alpha \geqslant 0$. Using English language as an example, the relative frequency of the most popular word "the" is 7%, and the relative frequencies of the next most popular words ("of" and "and") are 3.5% and 2.7%, respectively [Francis and Kucera 1967]. In other words, the most popular word occurs twice as often as the next most popular word, and approximately three times as often as the third most popular word. The frequency of words approximately follows Zipf's law with $\alpha = 1$.

In our experiment, roles selected from the role set R and assigned to users and permissions followed Zipf distribution. In particular, the more popular roles were assigned to more users in UA than the less popular roles. A role that appeared more frequently in UA, however, was assigned to fewer permissions in PA. This simulated a scenario where, for example, the "employee" role is usually assigned to more users than the "manager" role but the "employee" role usually has fewer permissions than the "manager" role.

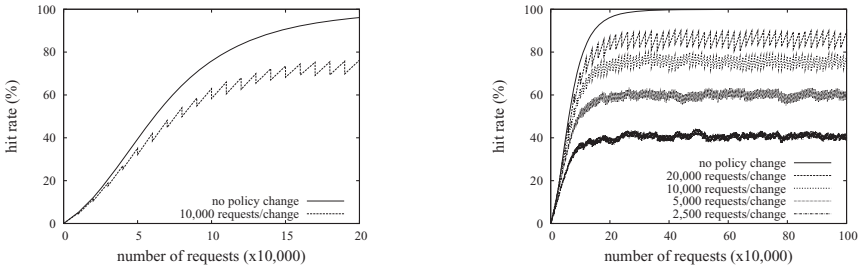
By using a Zipf distribution and varying α for role assignment, we implicitly simulated the existence of a role hierarchy RH. A popular role in UA simulated a junior role in RH that had fewer permissions but was assigned to more users. In contrast, a less popular role in UA simulated a senior role in RH that had more permissions (as it inherited permissions from all its junior roles) but was assigned to fewer users. In addition, by varying α, we implicitly varied the shape of the RH graph. When α is small, the corresponding RH graph has a wide and shallow shape. A large α makes the RH graph narrow and deep.

Since the popularity distribution becomes less and less skewed with the decrease of α, collapsing to a uniform distribution when $\alpha = 0$, we varied α between 0 and 1.5 in steps of 0.1. The results in Figure 6(h) show that, when α was lower than 1, the hit rate was almost the same as in the uniform distribution. When α was larger than 1, the hit rate began to increase along with α. This is expected because the number of "overlapping" roles between users increased. This was also due to the increase of negative responses in the cache because more users were assigned fewer permissions. However, when the cache warmness increased, this improvement was less significant due to the already high hit rate.

4.1.3. Evaluating the Impact of Policy Changes. We also studied the impact of policy changes on the hit rate. Since the hit rate depends on the cache warmness, and a policy change may result in removing one or more responses from SDP caches, we expected that frequent policy changes at a constant rate would unavoidably result in a reduced hit rate. This section quantifies this effect.

In the experiments, the simulation engine was responsible for firing a random policy change and sending the policy change message to both the PDP and SDP at predefined intervals, for example, after every 10,000 requests. The experiment switched from the warming mode to the testing mode once a policy change message was received. After measuring the hit rate right before and after each policy change, the experiment switched back to the warming mode.

We studied three types of policy change operations: adding a tuple to the PA relation; deleting a tuple from the PA relation; and deleting a role from R. When adding a tuple, the cache warmness increases slightly (as a tuple is added to $Cache^+$), so we would expect to see a slight increase in the hit rate. Our experiments confirmed this, although the hit rate never increased by more than 0.1%. Conversely, deleting a tuple from PA causes a reduction in the cache warmness, and is expected to result in a



(a) Hit rate as a function of number of requests with and without changes to R.

(b) Hit rate as a function of number of requests at various frequencies of R changes.

Fig. 7. The impact of removing a role from R on hit rate. In both parts of the figure, the order of the curves (from top to bottom) matches that of the legends.

decrease in the hit rate. Again, our experiments confirmed this expectation, and the decrease in hit rate was negligible.

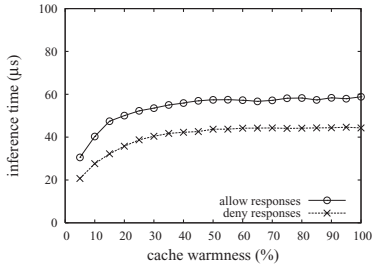
We now focus on the impact of deleting a role, as it is expected to have a more significant impact on the hit rate. We first studied how the hit rate was affected by an individual policy change, that is, the removal of a single role from R. We expected that $\text{SAAM}_{\text{RBAC}}$ inference algorithms were sufficiently robust so that an individual change would result in only minor degradation of the hit rate. In the experiment, the warming set contained 200,000 requests which were selected from the total request space with equal probability (with replacement). A randomly selected role was removed from R every 10,000 requests and tuples containing that role in UA and PA were also deleted. Then the cache was updated accordingly. After the experiment switched back to the warming mode from the testing mode, the removed role was returned to R; UA and PA were also restored. Thus the simulated system kept its policy characteristics. Any change in the hit rate was attributed to the reduced cache size.

Figure 7(a) shows the hit rate as a function of the number of observed requests, with policy changes (lower curve) or without policy changes (upper curve). Because the hit rate was measured just before and after each policy change, every kink in the curve indicates a hit rate drop caused by a policy change. The results suggest that the hit rate drops were relatively small; the maximum hit rate drop was 6.2%, and the average was 4.0%. After each drop, the curve climbed again because the cache warmness increased with new requests.

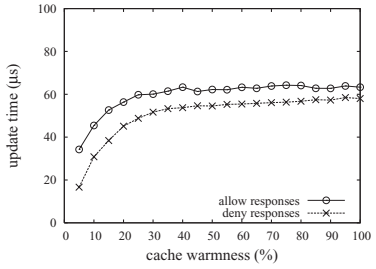
Although the hit rate drop for each policy change was small, one can see that the cumulative effect of policy changes could be large. As Figure 7(a) shows, the hit rate decreased about 20% in total when the request number reached 200,000. This result led us to another question: would the hit rate finally stabilize at some point?

To answer this question, we ran another experiment to study how the hit rate varied with continuous policy changes over a longer term. We used a larger number of requests (i.e., 1,000,000), and varied the frequency of policy changes from 2,500 to 20,000 requests per change.

Figure 7(b) shows the hit rate as a function of the number of requests, with each curve corresponding to a different frequency of random policy changes. Because of the continuous policy change, one cannot see a perfect asymptote of curves. However, the curves indicate that the hit rate stabilized after 200,000 requests. As we expected, the more frequent the policy changes were, the lower the stabilized hit rates were, since the responses were removed from the SDP caches more frequently. This result suggests



(a) Inference time (the time to generate approximate responses) variation with cache warmness



(b) Update time (the time to add a primary response to the cache) variation with cache warmness

Fig. 8. The impact of cache warmness on the inference and update time.

that, if R is changed frequently, it is preferable to purge the cache periodically instead of immediately.

Figure 7(b) also shows that each curve has a knee. The steep increase in the hit rate before the knee implies that caching new responses improves the hit rate dramatically in this interval. Once the number of responses passes the knee, the benefit brought by caching further responses becomes negligible.

4.1.4. Evaluating Inference and Update Time. Figure 8(a) shows the inference time for allow and deny approximate responses as a function of cache warmness for our reference configuration. As expected, the computational overhead to infer allow responses was larger than that for deny responses. The inference time increased with cache warmness for two reasons: first, when more responses were cached, the SDP used more responses for inference leading to higher computational overheads. Second, larger cache sizes led to less efficient memory usage (because the cache could not accommodate the SDP data).

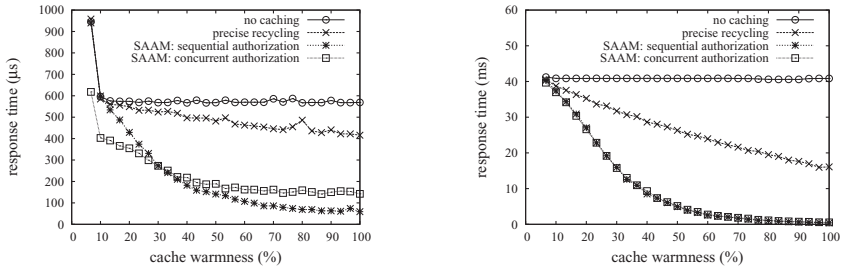
Figure 8(b) shows the time for updating the SDP cache using both allow and deny primary responses as a function of cache warmness. As expected, the update time also increased with cache warmness. Additionally, the SDP used more time to process allow than deny responses. The reason is that, in the case of processing each allow response $+(s, p)$, the SDP had to purge redundant tuples, that is, delete all $(s^+, p) \in Cache^+$ such that $s - s^- \subseteq s^+$, which involved an extra subset computation. This result suggests that, to improve the update time, the purge operation should be done in a periodical manner.

Note that both the inference time and update time stabilized when cache warmness reached about 40%. This was because at about 40% warmness the SDP was able to resolve all possible requests so new responses provided no new information to the cache.

4.2. Prototype-Based Evaluation

We have also implemented a simplified $\text{SAAM}_{\text{RBAC}}$ prototype system to evaluate the performance of overall authorization system. In particular, we studied the response time for two SAAM schemes (described in Section 3.10): sequential and concurrent authorization.

4.2.1. Experimental Setup. The prototype system consisted of the implementations of PEP, SDP, and PDP. The PEP was process collocated with the SDP, while the SDP communicated with the PDP using Java Remote Method Invocation (RMI). The PEP/SDP



(a) Scenario I: the SDP and the PDP were collocated on the same LAN and that the authorization policy of the PDP was relatively simple.

(b) Scenario II: the SDP was separated from the PDP by a WAN or/and the PDP had a complex authorization policy.

Fig. 9. Response time variation with cache warmness.

and PDP were located in two separate cluster nodes connected by a 1-Gb/s network. Each node was equipped with two Intel Xeon 2.33-GHz processors and 4 GB of memory, running Fedora Linux 2.6.24.3. Upon generating a random request at the PEP, the system attempted to resolve the request using one of the following two authorization schemes: *sequential authorization*, where a request was resolved first by the SDP and then by the PDP, or *concurrent authorization*, where a request was resolved by the SDP and the PDP concurrently.

For each authorization scheme, we ran experiments in the following two scenarios: (1) *Scenario I*, where the SDP and the PDP were collocated on the same local area network (LAN) and that the authorization policy of the PDP was relatively simple, thus allowing the PDP to make authorization decisions swiftly; and (2) *Scenario II*, where the SDP was separated from the PDP by a wide area network (WAN) or/and the PDP had a complex authorization policy. To model this scenario, we simulated additional 40-ms delay added to each authorization request sent to the PDP.

4.2.2. Evaluating Response Time. In our experiments, response time was measured as the time elapsed after the PEP generated a request until it received the response for that request. At the start of each experiment, the SDP caches were empty. The PEP uniformly selected a request from the request space, sent it to the SDP, and then recorded the response time for each request. After every 10,000 requests, the PEP calculated the mean response time and used it as an indicator of the response time for that period.

For both scenarios, we also ran experiments for the authorization system without SAAM, including authorization without caching or only using precise recycling. Our purpose was to evaluate the gains in terms of response time. The results shown in Figure 9 suggest that using SAAM helped to reduce the system response time in both scenarios and this reduction increased with cache warmness. Additionally, as we expected, the two SAAM authorization schemes showed different patterns in the two scenarios, which we explain below.

Figure 9(a) shows the result for Scenario I. The figure demonstrates that the response time for both schemes decreased with cache warmness, while sequential authorization decreased more quickly. The reason was likely due to the lower cost of resolving requests at the SDP. When cache warmness increased, more requests were resolved by the SDP. Since the SDP was process collocated with the PEP, getting responses through an interprocess call to the SDP was faster than getting responses through a network RMI call to the PDP.

More specifically, when cache warmness was small, that is, less than 30%, concurrent authorization achieved a shorter response time than sequential authorization. This was due to the extra time incurred by cache misses in sequential authorization. One unusual pattern in our result was that sequential authorization achieved a lower response time as cache warmness exceeded 30%. This was possibly caused by the thread management overhead in our concurrent authorization implementation. We should point out that, in an optimized implementation, concurrent authorization should at least achieve the same response time as the sequential authorization since concurrent authorization always uses first returned response.

Figure 9(b) shows the results for Scenario II. As expected, both response times decreased with cache warmness. More interestingly, the curves for concurrent and sequential $SAAM_{RBAC}$ authorizations almost overlapped each other. The reason is that in this scenario the extra time incurred by cache misses and thread management were small compared to the 40-ms delay at the PDP. Therefore, their impact on the response time was trivial.

4.3. Discussion

The results of our experiments indicate that approximate recycling leads to higher SDP hit rates than precise recycling alone, thus improving the availability and scalability of the access control system. Compared with the naive algorithms, the optimized algorithms achieve a higher hit rate using a smaller cache. These results extend our understanding of the factors that influence the hit rate as follows.

—For cache warmness between 5% and 50%, the hit rate for approximate recycling is notably better than that of precise recycling.
—Larger numbers of users in the system having similar role memberships substantially improve the hit rate.
—A higher proportion of deny responses in the cache leads to a higher hit rate.
—As the number of roles increases, the overlap between the sets of roles each user is assigned to decreases, thus reducing the likelihood of a successful inference based on cached responses.
—The hit rate is low when each user is assigned to few roles because the SDP cache has little relevant information. With the increase of overlap in users' roles, the number of relevant entries increases, resulting in the increase of the hit rate. While the overlap is still relatively low, the deny responses dominate the content of the SDP cache, resulting in a higher hit rate. However, when the number of roles per user increases further, $Cache^+$ starts increasing at the expense of $Cache^-$, leading to the decrease in the hit rate. When each user is assigned to (almost) all the roles in the system (almost) every user has the same set of roles, and the hit rate increases sharply to 100%.
—A larger number of roles per permission leads to a lower hit rate.
—Zipf's popularity distribution of roles leads to a higher hit rate when α is larger than 1, due to the increased overlap of roles assigned to users and permissions.

The volume of information available for inference, the percentage of deny responses, and the distribution of role assignment are the factors that are not controlled by the administrators of RBAC systems. Other factors that impact performance, however, for example, the total number of roles, the number of roles per user, and the roles per permission, might be engineered (e.g., by role engineering [Vaidya et al. 2007]) by the designers of access control policies who might be able to tune these factors to achieve higher hit rates using the trends our experiments and evaluation revealed. Thus, we believe our evaluation results can be used to inform efficient $\text{SAAM}_{\text{RBAC}}$ deployment in real enterprise systems, even though our experimental testbed was relatively small compared to large-scale systems deployed in organizations (e.g., Schaad et al. [2001]).

Our results indicate that the impact of the update to PA is trivial, as only a single permission is affected. In contrast, frequent policy changes to R may have a large impact on the hit rate. Since the correctness of the response is not affected if the cache is not updated immediately, it is preferable to purge the cache periodically instead of immediately.

Our experiments also demonstrate inference and update time well under 1 ms, and we believe that response times can be further reduced by optimizing the implementation. We note that a low inference time is a key attribute for a real-world deployment as it directly affects the perceived performance of the access control system: an application request cannot be processed until the PEP obtains a response, either primary or secondary. Cache changes triggered by adding primary responses or policy changes, on the other hand, can be implemented in the background to hide their impact on perceived performance.

The evaluation results on response time further suggest that the usefulness of SAAM techniques for reducing the response time of the overall access control system, especially in network-based deployments where network latencies are much larger or the PDP authorization logic, is complex. The results with two authorization schemes indicate that concurrent authorization is only helpful when the PDP can make authorization decisions quickly. In other cases, sequential authorization is preferable because it can achieve both reduced response time and reduced load at the PDP.

An alternative to approximate recycling for RBAC systems is to replicate RBAC policy at each SDP. Runtime benefits of the proposed approach—compared to just replicating PA and RH relations at each SDP—depend on a number of factors. The first factor is the size of the policy (mainly the PA, since this is likely to be the largest) relative to the size of a PEP working set (the set of all requests that come through the PEP). For a workload with good locality and a large PA, the proposed approach will require less space and may well be faster. Furthermore, if the PA is very large (say, larger than 10^9 elements) then it may be too expensive to duplicate the hardware that supports the PDP to additionally support each SDP. The second factor is the ability of a PDP to predict the working set of a PEP. If the PDP is able to predict a PEP's future working set, then providing the SDP with corresponding subsets of PA and RH will work better than authorization recycling (regardless of the relative sizes of the policy and the PEP working set). The third factor is the frequency of policy changes and the scope of these changes, that is, how many elements in the PA they affect. The fourth factor is the relative benefits brought by one-time replication of the PA (or some subset of it)—as proposed by Tripunitara and Carbunar [2009], for example—to the SDPs, as opposed to item-by-item caching of the responses.

Depending on the workload and policy characteristics, the most efficient solution may combine the proposed approach with the replication of some policy elements. For example, RH can be replicated to the SDPs, as suggested in Section 3.9. As a case in point, PEPs in the IBM Tivoli Access Manager [Karjoth 2003], which encodes PA in the form of access control lists, can operate in two modes. In "remote mode," a PEP sends authorization requests to the PDP. In "local mode," the PEP maintains a local replica of the authorization policy and performs all authorization decisions locally. Depending on the configuration, the policy local replica can be "pulled from" and/or "pushed" by the master authorization service database. "Overhead of policy replication" is mentioned in the technical documentation of the Access Manager [Bücker et al. 2003], but no evaluation is reported.

5. RELATED WORK

To improve the performance and availability of access control systems, caching of authorization decisions has been employed in a number of commercial systems

[Entrust 1999; Oracle 2008; Netegrity 2000], as well as several academic access control systems [Borders et al. 2005; Spencer et al. 1999]. However, all these systems only compute precise authorizations and therefore are only effective for resolving repeated requests. Beznosov [2005] introduced the concept of recycling approximate authorizations, and later Crampton et al. [2006] formally defined SAAM and introduced the concept of SDP. The SDP can resolve new requests by extending the space of supported responses to approximate ones. In other words, SAAM provides a richer alternative source for authorization responses than the existing approaches do. Additionally, to further improve the performance and availability of access control systems, Wei et al. [2007] explored the cooperation between multiple SDPS and combined SDP cooperation and approximate authorizations.

The inference of approximate responses usually depends on the underlying access control policy. For access control systems based on the Bell-LaPadula (BLP) model [Bell and LaPadula 1973a, 1973b], SAAM_{BLP} [Crampton et al. 2006] uses the relationships between subjects and objects of previous responses to infer approximate responses. In comparison with SAAM_{BLP}, $\text{SAAM}_{\text{RBAC}}$ infers relationships between sets of roles and the permissions assigned to those roles, thereby enabling the computation of approximate responses. Other work [Motro 1989; Rizvi et al. 2004; Rosenthal and Sciore 2001] used the relationships between (database) objects to infer new authorizations.

In general, SAAM is a domain-specific approach to improving performance and fault tolerance of access control mechanisms that employ remote authorization servers. The general classes of fault tolerance solutions are failure masking through information redundancy (e.g., error correction checksums), time redundancy (e.g., repetitive invocations), or physical redundancy (e.g., data replication). SAAM employs physical redundancy [Johnson 1996]: when the PDP is unavailable, the SDP is able to mask the fault by providing the requested access control decision if relevant authorization responses are cached. The SAAM approach requires no specialized operating system or communication software except modifications to the logic of the PEP cache. No distributed state, election, or synchronization algorithms are necessary either. With SAAM, only authorization responses are cached, and no dynamic authorization data are replicated, enabling linear scalability with the number of PEPs and PDPs.

6. CONCLUSION

As distributed systems become increasingly large and complex, their access control infrastructures face new challenges. Conventional request-response authorization architectures become fragile and scale poorly to large systems. Caching authorization decisions have long been used to improve access control infrastructure availability and performance. $\text{SAAM}_{\text{RBAC}}$ extends this approach by enabling the inference of approximate authorizations for RBAC systems. We propose algorithms to compactly cache authorization decisions and to efficiently infer approximate decisions from cached data. Our evaluation results demonstrate an average percentage increase of 36–132% in the number of authorization requests that can be served without consulting the original decision point, compared to precise recycling. These results suggest that deploying $\text{SAAM}_{\text{RBAC}}$ improves the availability and scalability of RBAC systems, and in turn the performance of entire enterprise systems.

ACKNOWLEDGMENTS

The initial ideas of authorization recycling and approximate authorizations have benefited significantly from the presentation and discussion of Beznosov [2005] at the New Security Paradigms Workshop 2005. Members of the Laboratory for Education and Research in Secure Systems Engineering gave valuable feedback on the earlier drafts of this article. The authors are grateful to the SACMAT and TISSec anonymous reviewers for their helpful comments.

REFERENCES

ADAMIC, L. AND HUBERMAN, B. 2002. Zipf's law and the Internet. *Glottometrics 3,* 1, 143–50.

ANSI. 2004. ANSI INCITS 359-2004 for role based access control. American National Standards Institute, New York, NY.

ASTLEY, M., STURMAN, D. C., AND AGHA, G. A. 2001. Customizable middleware for modular distributed software. *Comm. ACM. 44,* 5, 99–107.

BELL, D. AND LAPADULA, L. 1973a. Secure computer systems: A mathematical model. Tech. rep. MTR-2547, Volume II. Mitre Corporation, Bedford, MA.

BELL, D. AND LAPADULA, L. 1973b. Secure computer systems: Mathematical foundations. Tech. rep. MTR-2547, Volume I. Mitre Corporation, Bedford, MA.

BEZNOSOV, K. 2005. Flooding and recycling authorizations. In *Proceedings of the New Security Paradigms Workshop (NSPW'05)*. ACM Press, New York, NY, 67–72.

BORDERS, K., ZHAO, X., AND PRAKASH, A. 2005. CPOL: High-performance policy evaluation. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*. ACM Press, New York, NY, 147–157.

BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies*. IEEE Computer Society Press, Los Alamitos, CA, 126–134.

BÜCKER, A., ANTONIUS, J., RIEXINGER, D., SOMMER, F., AND SUMIDA, A. 2003. *Enterprise Business Portals II with IBM Tivoli Access Manager*. IBM Redbooks, Armonk, NY, ibm.com/redbooks.

COMMITTEE, X. T. 2005. OASIS eXtensible Access Control Markup Language (XACML) v. 2.0. OASIS, Burlington, VT.

CRAMPTON, J., LEUNG, W., AND BEZNOSOV, K. 2006. Secondary and approximate authorizations model and its application to Bell-LaPadula policies. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*. ACM Press, New York, NY, 111–120.

DEMICHIEL, L. G., YALÇINALP, L. Ü., AND KRISHNAN, S. 2001. *Enterprise JavaBeans, v. 2.0*. Sun. Oracle, Redwood Shores, CA.

ENTRUST. 1999. GetAccess design and administration guide. Entrust, Dallas, TX.

FERRAIOLO, D. AND KUHN, R. 1992. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference*. Gaithersburg, MD, 554–563.

FRANCIS, W. AND KUCERA, H. 1967. *Computational Analysis of Present-Day American English*. Brown University Press, Providence, RI.

GITTLER, F. AND HOPKINS, A. C. 1995. The DCE security service. *Hewlett-Packard J. 46,* 6, 41–48.

INTERNET2. 2008. Shibboleth system. `http://shibboleth.internet2.edu.`

JOHNSON, B. 1996. *Fault-Tolerant Computer System Design*. Prentice-Hall, Upper Saddle River, NJ, 1–87.

KAIJSER, P. 1998. A review of the SESAME development. In *Information Security and Privacy,* C. Boyd and E. Dawson, Eds. Lecture Notes in Computer Science, vol. 1438. Springer, Berlin, Germany, *1438*, 1–8.

KALBARCZYK, Z., LYER, R. K., AND WANG, L. 2005. Application fault tolerance with Armor middleware. *IEEE Internet Comput. 9,* 2, 28–38.

KARJOTH, G. 2003. Access control with IBM Tivoli Access Manager. *ACM Trans. Info. Syst. Sec. 6,* 2, 232–57.

KLEMM, A., LINDEMANN, C., VERNON, M. K., AND WALDHORST, O. P. 2004. Characterizing the query behavior in peer-to-peer file sharing systems. In *Proceedings of the SIGCOMM Internet Measurement Conference*. New York, NY, 55–67.

LORCH, M., PROCTOR, S., LEPRO, R., KAFURA, D., AND SHAH, S. 2003. First experiences using XACML for access control in distributed systems. In *Proceedings of XMLSec*. ACM, Press, New York, NY, 25–37.

MARKOFF, J. AND HANSELL, S. 2006. Google's not-so-very-secret weapon. *International Herald Tribune.* June 13.

MOTRO, R. 1989. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *Proceedings of the International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, CA, 339–347.

NETEGRITY. 2000. Siteminder concepts guide. Tech. rep. Netegrity, Waltham, MA.

NICOMETTE, V. AND DESWARTE, Y. 1997. An authorization scheme for distributed object systems. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*. IEEE Computer Society Press, Los Alamitos, CA, 21–30.

OMG. 2002. Common object services specification, security service specification v1.8. OMG, Needham, MA.

ORACLE. 2008. Oracle entitlements server: Programming security for web services. Tech. rep. Oracle. Redwood Shores, CA.

RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. 2004. Extending query rewriting techniques for fine-grained access control. In *Proceedings of the SIGMOD International Conference on Management of Data*. ACM, Press, New York, NY.

ROSENTHAL, A. AND SCIORE, E. 2001. Administering permissions for distributed data: Factoring and automated inference. In *Proceedings of AWCDAS*. Kluwer, Norwell, MA, 91–104.

RYUTOV, T. AND NEUMAN, C. 2000. Generic authorization and access control application program interface: C-bindings. Internet Draft draft-ietf-cat-gaa-bind-03, Internet Engineering Task Force. www.ietf.orgo.

SALTZER, J. AND SCHROEDER, M. 1975. The protection of information in computer systems. *Proc. IEEE 63,* 6, 1278–1308.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-based access control models. *IEEE Comput. 29,* 2, 38–47.

SCHAAD, A., MOFFETT, J., AND JACOB, J. 2001. The role-based access control system of a European bank: A case study and discussion. In *Proceedings of the ACM Symposium on Access Control Models and Technologies*. ACM Press, New York, NY, 3–9.

SCHRODER-PREIKSCHAT, W., LOHMANN, D., SCHELER, F., GILANI, W., AND SPINCZYK, O. 2006. Static and dynamic weaving in system software with AspectC++. In *Proceedings of the Hawaii International Conference on System Sciences*. 214.1.

SECURANT. 1999. Unified access management: A model for integrated Web security. Tech. rep. Securant Technologies. Belford, MA.

SPENCER, R., SMALLEY, S., LOSCOCCO, P., HIBLER, M., ANDERSEN, D., AND LEPREAU, J. 1999. The Flask security architecture: System support for diverse security policies. In *Proceedings of the 8th USENIX Security Symposium*. USENIX Berkeley, CA, 123–140.

STRONG, P. 2007. How Ebay scales with networks and the challenges. In *Proceedings of the 16th ACM/IEEE International Symposium on High-Performance Distributed Computing*. ACM Press, New York, NY. Invited talk.

TRIPUNITARA, M. V. AND CARBUNAR, B. 2009. Efficient access enforcement in distributed role-based access control (RBAC) deployments. In *Proceedings of the ACM Symposium on Access Control Models and Technologies ACM Press,* Press, New York, NY, 155–164.

VAIDYA, J., ATLURI, V., AND GUO, Q. 2007. The role mining problem: Finding a minimal descriptive set of roles. In *Proceedings of the ACM Symposium on Access Control Models and Technologies ACM Press*. New York, NY, 175–184.

VOGELS, W. 2004. How wrong can you be? Getting lost on the road to massive scalability. In *Proceedings of the 5th International Middleware Conference*. ACM Press, New York, NY. Keynote address.

WEI, Q., RIPEANU, M., AND BEZNOSOV, K. 2007. Cooperative secondary authorization recycling. In *Proceedings of the IEEE International on High-Performance Distributed Computing*. IEEE, Coputer Society Press, Los Alamitos, CA, 65–74.

Received October 2008; revised December 2009; accepted April 2010