Authorization Recycling in RBAC Systems

Qiang Wei
LERSSE*
University of British Columbia
qiangw@ece.ubc.ca

Jason Crampton
Information Security Group
Royal Holloway,
University of London
jason.crampton@rhul.ac.uk

Konstantin Beznosov
LERSSE*
University of British Columbia
beznosov@ece.ubc.ca

Matei Ripeanu
LERSSE*
University of British Columbia
matei@ece.ubc.ca

ABSTRACT

As distributed applications increase in size and complexity, traditional authorization mechanisms based on a single policy decision point are increasingly fragile because this decision point represents a single point of failure and a performance bottleneck. Authorization recycling is one technique that has been used to address these challenges.

This paper introduces and evaluates the mechanisms for authorization recycling in RBAC enterprise systems. The algorithms that support these mechanisms allow precise and approximate authorization decisions to be made, thereby masking possible failures of the policy decision point and reducing its load. We evaluate these algorithms analytically and using 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 [**Computer-Communication Networks**]: Performance of Systems—*Reliability, availability, and serviceability*

General Terms

Security, Design, Performance, Reliability

Keywords

SAAM, RBAC, access control, authorization recycling

---

*Laboratory for Education and Research in Secure Systems Engineering (lersse.ece.ubc.ca)

1. INTRODUCTION

Modern access control solutions [2, 6, 9, 10, 14, 17, 19, 25, 26] are based on the request-response paradigm illustrated in Figure 1. In this paradigm, a policy enforcement point (PEP) intercepts application requests, obtains access control decisions (a.k.a. authorizations) from a policy decision point (PDP), and enforces these decisions.
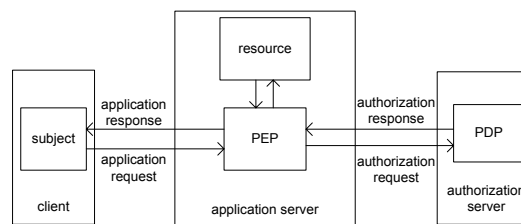

Figure 1: Access control based on request-response paradigm.

In the large enterprise systems currently deployed, PDPs are commonly implemented as logically centralized authorization servers. This design provides important benefits: consistent policy enforcement across multiple PEPs and reduced administration cost for authorization policies. Like all centralized architectures, 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 due to or 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 [12]). 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 [13, 29]. At the same time, large-scale commodity computing is becoming a reality, with eBay having 12,000 servers and 15,000 application server instances [27], and Google estimated to have "more than 450,000 servers spread in at least 25 locations around the world" [15].

In a massive-scale enterprise system with non-trivial authorization policies, making authorization decisions is often computation-
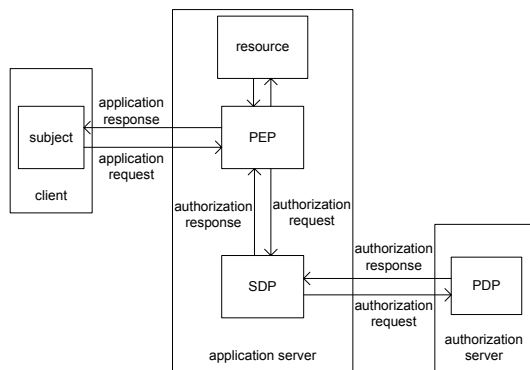
Figure 2: SAAM adds SDP to the request-response paradigm.

ally 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 [18]. Additionally, the communication delay between the PEP and the PDP can make the authorization overhead prohibitively high.

To address the aforementioned drawbacks, one possible approach could be for the PDP to push policies to each PEP so that they can make authorizations locally. However, this is rarely done in enterprise-grade deployments. We speculate that the tendency towards the centralization of authorization decisions is due to the complexity of authorization logic and the cost of keeping up to date user, attribute, and permission data in multiple PEPs. 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 [2, 6, 10, 14, 17, 26]. 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, Crampton et al. [8] propose the Secondary and Approximate Authorization Model (SAAM). SAAM adds a secondary decision point (SDP) to the request-response paradigm, 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 and access control policy. For each class of access control policies, however, specific algorithms for inferring approximate responses—generated according to a particular access control policy—need to be designed.

In this paper, we propose SAAM$_{RBAC}$—the SAAM authorization recycling algorithm for role-based access control (RBAC) model. Introduced more than a decade ago, RBAC [11, 23] has been deployed in many organizations for access control enforcement and eventually matured into the ANSI RBAC standard [1]. In RBAC, instead of directly assigning permissions to users, the users are assigned to roles and the roles are mapped to permissions. Roles normally represent the organizational position that is responsible for certain job functions. Users are assigned appropriate roles according to their qualifications. Permissions are a set of many-to-many relations between objects and operations. Roles describe the rela-

tionship between users and permissions. Our inference algorithm makes use of this structure to infer approximate authorizations for new requests.

This paper makes the following two contributions. First, we develop SAAM$_{RBAC}$, an application of SAAM to RBAC systems. We define inference rules specific to RBAC authorization semantics and develop the recycling algorithms based on these rules. We construct three algorithms: the first caches authorization responses from the PDP and represents them as a compact data structure; the second uses this data structure to generate secondary (precise or approximate) responses; the third handles policy changes by updating the data structure. We show that the computational complexity of the algorithms is bound by the cache size and the number of roles in the system.

Second, we implement SAAM$_{RBAC}$ algorithms and evaluate their properties using an experimental testbed with 100 subjects, 3,000 permissions and 50 roles. Evaluation results demonstrate a 74% increase, compared to precise recycling, in the number of authorization requests that can be served without consulting access control policies. These results suggest that deploying SAAM$_{RBAC}$ improves the availability and scalability of RBAC systems, which in turn improves the performance of the enterprise systems.

The rest of this paper is organized as follows. Section 2 presents background, including SAAM and RBAC. Section 3 describes SAAM$_{RBAC}$ design. Section 4 reports results of evaluating SAAM$_{RBAC}$. 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 paper.

2.1 Secondary and Approximate Authorization Model

SAAM [8] 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 request contextual information, 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.

In addition, SAAM 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. As more and more PDP responses are cached, 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 [8]. 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 [22]. 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 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 [23] and the ANSI RBAC standard [1]. 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-role assignment relation $UA \subseteq U \times R$ and a permission-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 that $(r, r') \in RH$. 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 $(p, r') \in PA$.

The other 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. SAAM_{RBAC}

SAAM_{RBAC} applies SAAM concepts to RBAC systems. In a system using SAAM_{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 should 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 Preliminaries

We must first consider how to map the notions of subject and request in SAAM to appropriate concepts in RBAC. The notion of session is important in RBAC96 and ANSI RBAC: 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, we model a subject as a set of roles.

We assume that access requests made to the SDP (and the PDP) include the set of roles, this information being supplied by the PEP. We also assume that the SDP does not have access to the permission-role assignment relation or the role hierarchy relation. It is the job of the SDP to try to reconstruct relationships between permissions and roles on the basis of information that can inferred from primary responses to previous requests. The SDP does not try to reconstruct hierarchical relationships between roles.

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 $p = (o, a)$. For simplicity, we omit c and i in the remainder of the paper.

A response (r, i, E, d) indicates the decision to a request (s, p). For simplicity, we omit r, E and i in the remainder of the paper, and use $\pm$ to denote d. That is, we write $+q$ to denote a response that allows request q, and $-q$ to denote a response that denies q. More specifically, $+(s, p)$ means that there exists role $r \in s$ such that $(r, p) \in PA$ and $-(s, p)$ means that there does not exist such an r.

3.2 Building the cache

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

Rule$^+$ if $+(s, p)$ and $s' \supseteq s$, then request (s', p) should be granted;

Rule$^-$ if $-(s, p)$ and $s' \subseteq s$, then request (s', p) should be denied.

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 $s' \supseteq s$. **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.

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

A primary response $-(s, p)$ means that we can deduce that certain roles are definitely *not* authorized for permission p. Hence, $(s, p) \in Cache^-$ is used to record the fact that no element of s is authorized for p. A subsequent (primary) response $-(s', p)$ means that no role in s' is authorized for p. Hence, we can simply update $Cache^-$ by replacing (s, p) with $(s \cup s', p)$. In other words, we may assume that there exists at most one entry containing p in $Cache^-$.

In contrast, a primary response $+(s, p)$ can only be used to infer that at least one role in s is authorized for p. A subsequent response $+(s', p)$ can not be "merged" with the information from $+(s, p)$. Therefore, we simply add $+(s', p)$ to $Cache^+$.

The full algorithm (**C**) for constructing the cache relations is shown in Figure 3(a). Note that in line 3C, which handles negative primary responses, we can delete any roles in s from sets of roles that had previously been authorized for p (that is, tuples in $Cache^+$). Analogously, in line 12C we can delete any roles from s that are known not to be authorized for p. Note that if we know s is

---

Input: response q

1C: $AddResponse(q)$
2C: **if** $q = -(s,p)$ **then**
3C: replace each $(s^+,p) \in Cache^+$ with $(s^+ - s, p)$[a]
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: find $(s^-,p) \in Cache^-$
11C: delete all $(s^+,p) \in Cache^+$ s.t. $s - s^- \subseteq s^+$
12C: add $(s - s^-, p)$ to $Cache^+$
13C: **end if**

---

[a] "$s^+ - s$" denotes the set of roles in s^+ that are not in s.

(a) **C**: The cache construction algorithm

Input: request (s,p)

1D: $EvaluateRequest(s,p)$
2D: **if** $s \subseteq s^-$, where $(s^-,p) \in Cache^-$ **then**
3D: **return** deny
4D: **else if** there exists $(s^+,p) \in Cache^+$ s.t. $s^+ \subseteq s$ **then**
5D: **return** allow
6D: **else**
7D: **return** undecided
8D: **end if**

(b) **D**: The decision algorithm

Figure 3: SAAM$_{\text{RBAC}}$ algorithms

authorized for p, then any superset of s is also authorized. Accordingly, line 11C is used to prune redundant tuples from $Cache^+$.

3.3 Generating approximate responses

Figure 3(b) shows the decision algorithm (**D**) for generating an approximate response, which follows directly from rules **Rule$^+$** and **Rule$^-$**. 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 1. *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 12C imply that if $(t^-,p) \in Cache^-$ and $(t^+,p) \in Cache^+$, then $t^- \cap t^+ = \emptyset$.

Suppose there exists a primary allow response for (s,p). Then $(s^+,p) \in Cache^+$ for some $s^+ \subseteq s$ (by lines 10C–12C). Therefore, using lines 4D–5D, the SDP will return an allow response for the request (s,p). (Note that the algorithm cannot return deny as this would imply that $(s^-,p) \in Cache^-$ for some $s^- \subseteq R$ and $s \subseteq s^-$. Hence, we would have $s^+ \subseteq s \subseteq s^-$, but we know by the observation in the previous paragraph that $s^+ \cap s^- = \emptyset$.)

Conversely, if there exists a primary deny response for (s,p), then $(s^-,p) \in Cache^-$ for some $s^- \supseteq s$. Hence, by lines 2D–3D, the SDP will return a deny response for request (s,p). □

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 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 $(s^-,p) \in Cache^-$ and $s \subseteq s^-$ (by lines 2D–3D). By construction of $Cache^-$, for each $r \in s^-$, r is not authorized for p. Hence, the PDP would return $-(s,p)$.

Suppose that the SDP produces the response $+(s,p)$. Then (by lines 4D–5D) there exists $(s_1^+,p) \in Cache^+$ such that $s \supseteq s_1^+$. Moreover, there exists at least one $r \in s_1^+$ such that r is authorized for p, since the existence of (s_1^+,p) in $Cache^+$ implies the existence of a primary response $+(s_2^+,p)$, where $s_2^+ \supseteq s_1^+$. Hence, the PDP would return $+(s,p)$. □

3.4 Example

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 1 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.

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

Table 1: Building $Cache^+$ and $Cache^-$ from primary responses

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).

3.5 Discussion

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 session 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.

3.5.1 Secondary response rate

Let us suppose, then, that $(s_1^+, p), \ldots, (s_m^+, p) \in Cache^+$ and $(s^-, p) \in Cache^-$. Then the probability that we can produce an approximate response (a "hit") for request (s, p) is the probability that the SDP either returns deny or returns allow. The SDP returns allow if $s \supseteq s_i^+$ for some i, and the probability that $s \supseteq s_i^+$ is

$$\frac{2^{|R|-|s_i^+|}}{2^{|R|}} = \frac{1}{2^{|s_i^+|}}$$

The SDP returns deny if $s \subseteq s^-$, and the probability that $s \subseteq s^-$ is

$$\frac{2^{|s^-|}}{2^{|R|}}$$

Hence, the probability that we have a hit is

$$\frac{2^{|s^-|}}{2^{|R|}} + \sum_{i=1}^{m} \frac{1}{2^{|s_i^+|}}$$

As one would expect, this value depends on the sizes of s^- and $s_1^+, \ldots s_m^+$. In particular, as the size of s^- increases and the sizes of s_i^+ decrease, the probability of a hit is increased. It can be seen from the construction algorithm that the effect of receiving 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). We can ensure that s_i^+ is small 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. In summary, we would expect probability of a hit (the "hit rate") to increase when users are assigned either to a small number of roles or to a significant proportion of the roles available. We sought to confirm these expectations by experiment, the results of which are reported in Section 4.

3.5.2 Performance considerations

Clearly, the number of tuples in $Cache^-$ is bounded by $|P|$, while the number of tuples in $Cache^+$ is bounded by $|P| 2^{|R|}$. A secondary deny response can be computed in time proportional to $|R|^2$, as we simply need to determine whether the requesting set of roles 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). In the case of an allow response, we check to see whether each tuple has been made redundant by the new information (line 11C).

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. Again, we investigate some of these aspects in Section 4.

3.6 Handling policy changes

A real 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. In this paper, we only consider those changes that involve modification of PA and leave changes that also involve R or RH for future work. In regards to PA, 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 make false negative decisions for some requests for p, because it may compute deny decisions for those requests that are allowed by the PDP. To avoid this, r needs to be removed from $Cache^-$ as well as 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, r needs to be added to $Cache^-$ as well as removed from $Cache^+$.

We signal policy updates to the SDP by passing "artificial" responses from the PDP to the SDP. Specifically: when (p, r) is added to PA, the SDP need only process a (primary) response $+(\{r\}, p)$; and when (p, r) is removed from PA, the SDP need only process a (primary) 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 4), rather the cache construction algorithm.

The full algorithm for updating the cache relations to deal with updates to PA is shown in Figure 4. 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 (line 3U). This is because we can not assume that any of the remaining roles in the tuple are authorized for p. Second, if p is assigned to r, we must delete r from the set of roles in $Cache^-$ (line 13U), since we know that r is authorized for p.[2]

The similarity between the construction and update algorithms means that they can easily be merged to form a single algorithm, in which different branches of the program are selected according to the type of response (normal or policy update) that is being processed. We have presented the algorithms separately for ease of exposition.

---

[2]Note line 15U 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.

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

Figure 4: U: cache update algorithm

There are two ways in which policy update responses can be propagated to SDPs: they can be sent by the PDP to each SDP immediately after a policy update or they can be sent in addition to a particular PEP (and associated SDP) in response to a request from that PDP. The former method requires the PDP to initiate communication between itself and the PEP, something that may not be compatible with the existing protocols within the authorization infrastructure. Hence, it may well be appropriate for the policy update responses to "piggy-back" on primary responses from the PDP. Of course, this raises the possibility that the SDP may make inappropriate responses for p, prior to receipt and processing of the update response.

4. EXPERIMENTAL EVALUATION

The previous sections describe SAAM$_{RBAC}$ algorithms and estimate their performance analytically. This section presents an experimental evaluation of the cache construction and decision algorithms. We studied two performance aspects of our algorithms: the achieved *hit rate* and the *computational cost*.

We define the hit rate as the ratio between the number of requests solved locally (regardless of the specific allow/deny decision) by the SDP and the total number of requests received. A high cache 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.

Our analysis in Section 3.5 suggested that the hit rate is influenced by the following factors: (1) the cache warmness, that is, the ratio between the number of authorization responses cached at the SDP and 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 4.2 presents results of our experiments investigating the impact of these factors on the hit rate.

The second performance aspect we investigated was the *computational cost*. We measured two types of computational costs. First was the *inference time*—the time that it took the SDP to infer an approximate response (allow or deny) using its cache. The second was the *update time*—how long it took the SDP to incorporate a new primary response in its cache. The lower the inference time, the more efficient the SDP is in accelerating the access control sys-

tem. In particular, we present in Section 4.3 the influence of cache warmness on inference and update time as cache warmness appears to be the main influencing factor. For the sake of brevity, we refer to both times as "response time."

4.1 Experimental Setup

To conduct the experiments, we have modified the SAAM evaluation engine used in [8] to support SAAM$_{RBAC}$. Each run of the evaluation engine involved four stages. In the first stage, the engine created an RBAC policy and assigned both users (UA) and permissions (PA) to roles. Second, the engine created the *warming set* of users and permissions: that is the set containing all possible unique requests in the system. We also created a testing set, which comprised a sampling of requests. In our experiment, the testing set contained 20,000 requests, which were uniformly selected from the request space. For a confidence interval of 95%, this yielded a maximum error margin of 0.69% in the hit rate experiments and $5\mu s$ in the response time experiments. Next, the simulation engine started operating by alternating between *warming* and *testing* modes. In the warming mode (stage three), the engine used a subset of the requests from the warming set, evaluated them using a simulated PDP, and sent the responses to the SAAM$_{RBAC}$ SDP to build up the cache. Additionally, at this stage, the evaluation engine 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 testing mode (stage four) where the SDP cache was not updated anymore. We used this stage to evaluate the hit rate and the inference time at controlled, fixed levels of cache warmness. The engine submitted requests from the test set, recorded the inference time, and calculated the hit rate as the ratio of the test requests resolved by the SDP to all test requests and the average inference time at the end of this phase. These last two stages were then repeated for different levels of cache warmness, from 0% to 100% in increments of 5%.
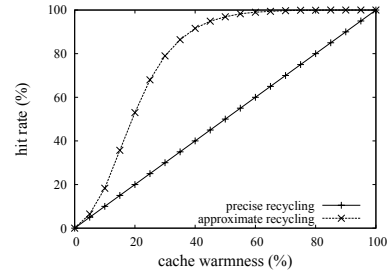

Figure 5: Hit rate as a function of cache warmness for an RBAC system tested with 100 users, 3,000 permissions, and 50 roles.

Note that our evaluation assumed that a user launched only one process which activated all the roles assigned to the user. Therefore the subject in each request had all the roles of the user.

For our experiments we used a commodity PC with an AMD Athlon Dual Core processor 3800+ 2.00 GHz and 3GB of RAM, running Windows XP. The evaluation framework ran on Sun's 1.5.0 Java Runtime Environment (JRE).

The reference RBAC policy in our experiments contained 100 users, 3,000 permissions, and 50 roles. Thus the overall size of the request space was 300,000. Each user was assigned to five roles and each permission was assigned to two roles, both assignments following uniform distribution. Figure 5 presents the hit rate as
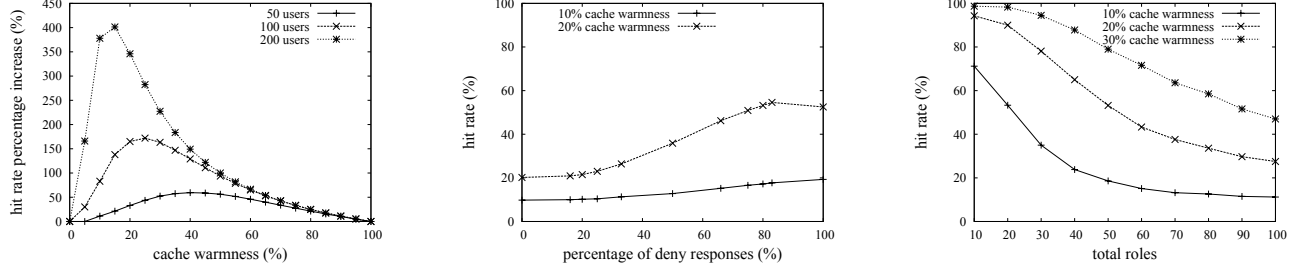
(a) Hit rate percentage increase as the SDP cache warmness varies, for 50, 100, and 200 users in the RBAC system.

(b) Hit rate variation with the percentage of primary deny responses in the SDP cache.

(c) Hit rate variation with the total number of roles in the RBAC system
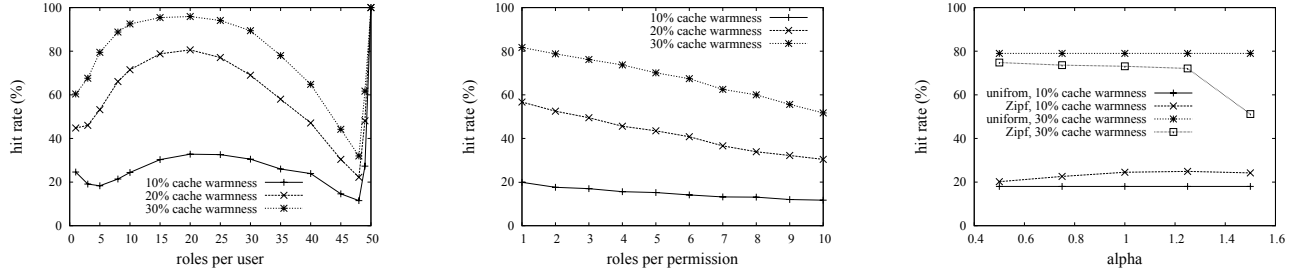
(d) Hit rate variation with the number of roles per user

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

(f) Hit rate variation with the coefficient α in Zipf distribution of role popularity $P = 1/r^{\alpha}$

Figure 6: The impact of various system characteristics on the hit rate. 6(b) to 6(f) are for an RBAC system with the reference configuration (100 users, 3,000 permissions, and 50 roles) and fixed cache warmness (10%, 20%, and 30%). Subfigures 6(a) to 6(e) are with uniform role assignment and 6(f) is with Zipf role assignment.

a function of *cache warmness* for both approximate recycling and precise recycling with the reference configuration. As expected, hit rate of approximate recycling increased with cache warmness and was always higher than that of precise recycling.

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 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 we do not expect that the overall number of permissions in a system to influence the achieved hit rate while it may influence the response time as a large number of permissions leads to less efficient memory use by the SDP.

4.2 Hit Rate

We studied the impact of varying *the number of users* while maintaining the other configuration parameters constant. Figure 6(a) shows the *percentage increase* for the hit rate compared to 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 been cached thus leading to a higher hit rate. When averaged over the full range of cache warmness, the percentage increase was 30%, 74%, and 128% 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 choose to explore hit rate for relatively low cache warmness values (at 10%, 20%, 30% respectively) as this is the region where we estimate the system is 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*. Figure 6(b) 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 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 the curve for 30% cache warmness is missing because we were unable to warm the SDP cache to 30% using only deny responses.

Second, we studied the impact of *the total number of roles* on the hit rate by varying it from 10 to 100 (Figure 6(c)) and keeping constant the number of users and the number of roles a user is assigned to. The results indicate that, as the number of roles increases, the hit rate decreases. This confirms our analytical prediction that, 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.

Third, we studied the impact of *the number of roles each user is assigned to* by varying it from one to all the roles that exist in the
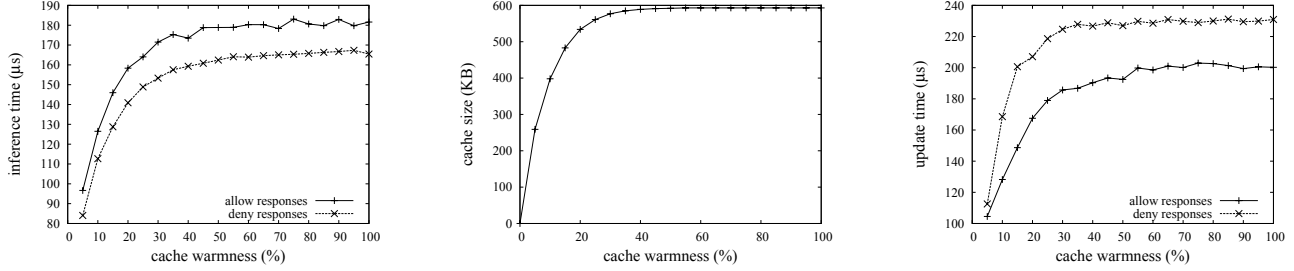
(a) Inference time (time to generate approximate responses) variation with cache warmness

(b) Cache size variation with cache warmness

(c) Update time (time to add primary responses) variation with cache warmness

Figure 7: The impact of cache warmness on response time and cache size.

system (50 roles) while keeping all other parameters constant. The results in Figure 6(d) suggest that the influence of this parameter on the hit rate is more complex. Our explanation for the shape of the hit rate curves is the following. The hit rate is low when each user is assigned to few (1-3) roles because both $Cache^+$ and $Cache^-$ for each individual permission have few entries relevant to the request in question. 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 (when each user has less than ten roles), the deny responses dominate the content of the SDP cache, resulting in a higher hit rate, as our analytical model predicted and the results in Figure 6(b) confirmed. 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 (as we predicted in Section 3.5). What the analytical analysis did not predict is the sharp increase to 100% in the hit rate on the right side of the graph. This increase is due to the fact that each user is assigned to (almost) all the roles in the system and, as a result, (almost) every user has the same set of roles.

Fourth, we studied the impact of *the number of roles each permission is assigned to*. Figure 6(e) 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 other experiments, users and permissions were uniformly assigned to roles and all roles were equally "popular" in UA and PA relations. However, in reality some roles could be more popular than others. 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's Law [7] distribution, which expresses a power-law relationship form $P = 1/r^\alpha$ for some constant α between the popularity P of an item (i.e., its frequency of occurrence) and its rank r based on the frequency of occurrence.

We varied the coefficient for α between 0.5 and 1.5, the value of the coefficient found in popularity distributions for other networked systems (e.g., web cached item popularity, website popularity, search keyword popularity). The results in Figure 6(f) suggest that Zipf's popularity distribution of roles leads to higher hit rates at a lower cache warmness while uniform assignment gets better hit rate at higher cache warmness. However, the difference was small. We expect real-world distributions of role "popularity" to be somewhere in the range between uniform and Zipf.

4.3 Response time

Figure 7(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 the 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 by the SDP (that is, SDP data does not fit in the host's cache anymore). When cache warmness reached about 40%, the response time stabilized. This was because at about 40% warmness the SDP was able to resolve all possible requests (see Figure 5) so new responses provided no new information to the cache. We validated this hypothesis by measuring the cache size: Figure 7(b) shows that after cache warmness reached about 40%, the cache size stabilized.

Figure 7(c) shows the time for updating the SDP cache using both allow and deny primary responses as a function of cache warmness. The SDP used more time to process deny than allow responses. The reason is that in the case of processing each deny response $-(s, p)$, the SDP had to replace each $(s', p) \in Cache^+$ with $(s' - s, p)$.

We have experimented with various other configurations and, even when stressing the system with large numbers of users, roles per user, and permissions, the response time for inferences remained under 1ms. 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 needs to wait until the SDP obtains 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.

4.4 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. These results confirmed the predictions of our mathematical analysis in Section 3.5 and also extended our understanding of the factors that influence the hit rate:

- 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.

- The hit rate is low when each user is assigned to few (1-3) 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 (when each user has less than ten roles), 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 at a lower cache warmness while uniform assignment gets better hit rate at higher cache warmness, albeit with the difference being small.

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, e.g., the total number of roles, the number of roles per user, and roles per permission, might be engineered (e.g., by role engineering [28]) 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 SAAM$_{RBAC}$ deployment in real enterprise systems, even though our experimental testbed was relatively small compared to large-scale systems deployed in organizations (e.g., [24]).

Our experiments with SAAM$_{RBAC}$ also demonstrated response times well under 1ms for all operations and in a number of configurations. While we believe that response times can be further reduced by optimizing algorithm implementations, this performance level demonstrates the usefulness of SAAM techniques for reducing the response time of the overall access control system in those network-based deployments where network latencies are much larger (e.g., MAN, WAN).

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 [10, 14, 17], as well as several academic access control systems [2, 6, 26]. However, all these systems only compute precise authorizations and therefore are only effective for resolving repeated requests. Beznosov [5] introduces the concept of recycling approximate authorizations, and later Crampton et al. [8] formally define SAAM and introduce 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. To further improve the performance and availability of access control systems, Wei et al. [30] explore the cooperation between multiple SDPS and combine 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 [4, 3], SAAM$_{BLP}$ [8] uses the relationships between subjects and objects of previous responses to infer approximate responses. Other work [16, 20, 21] uses the relationships between (database) objects to infer new authorizations. In contrast, SAAM$_{RBAC}$ caches information derived from primary responses in order to infer relationships between sets of roles and the permissions assigned to those roles, thereby enabling the computation of approximate responses.

In general, SAAM is a domain-specific approach to improving performance and fault tolerance of those access control mechanisms that employ remote authorization servers. Three 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 [12]: when the PDP is unavailable, the SDP would be able to mask the fault by providing the requested access control decision. 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 on 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 has long been used to improve access control infrastructure availability and performance. This paper extends this approach by enabling the inference of approximate authorizations for RBAC systems. We propose new algorithms to compactly cache authorization decisions and to efficiently infer approximate decisions from cached data. Our evaluation results demonstrate a percentage increase of 30-128% 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 SAAM$_{RBAC}$ improves the availability and scalability of RBAC systems, and in turn the performance of entire enterprise systems.

Since SAAM$_{RBAC}$ caches both allow and deny responses, the cache size could become very large. We plan to explore cache replacement algorithms that reduce this side-effect. We also plan to study the policy changes involving R and RH. In addition, we plan to evaluate our algorithms with a more realistic enterprise-scale RBAC system and corresponding policy. Our algorithms can handle flat as well as hierarchical role sets. We plan to explore opportunities for further improvements when the role hierarchy relation is available to the SDP. Another avenue for further research is the exploitation of caches from different but cooperating SDPs, similar to the distributed version of SAAM$_{BLP}$ [30].

7. REFERENCES

[1] ANSI. ANSI INCITS 359-2004 for role based access control, 2004.

[2] BAUER, L., GARRISS, S., AND REITER, M. K. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy* (Oakland, CA, 2005), pp. 81–95.

[3] BELL, D., AND LAPADULA, L. Secure computer systems: A mathematical model. Tech. Rep. MTR-2547, Volume II, Mitre Corporation, Bedford, Massachusetts, 1973.

[4] BELL, D., AND LAPADULA, L. Secure computer systems: Mathematical foundations. Tech. Rep. MTR-2547, Volume I, Mitre Corporation, Bedford, Massachusetts, 1973.

[5] BEZNOSOV, K. Flooding and recycling authorizations. In *Proceedings of the New Security Paradigms Workshop (NSPW)* (Lake Arrowhead, CA, USA, 20-23 September 2005), pp. 67–72.

[6] BORDERS, K., ZHAO, X., AND PRAKASH, A. CPOL: high-performance policy evaluation. In *Proceedings of the 12th ACM conference on Computer and Communications Security (CCS)* (New York, NY, USA, 2005), ACM Press, pp. 147–157.

[7] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the Conference on Computer Communications (INFOCOM)* (1999), pp. 126–134.

[8] CRAMPTON, J., LEUNG, W., AND BEZNOSOV, K. Secondary and approximate authorizations model and its application to Bell-LaPadula policies. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT)* (Lake Tahoe, CA, USA, June 7–9 2006), pp. 111–120.

[9] DEMICHIEL, L. G., YALÇINALP, L. Ü., AND KRISHNAN, S. *Enterprise JavaBeans Specification, Version 2.0.* Sun Microsystems, 2001.

[10] ENTRUST. GetAccess design and administration guide. Tech. rep., Entrust, September 20 1999.

[11] FERRAIOLO, D., AND KUHN, R. Role-based access controls. In *Proceedings of the 15th NIST-NCSC National Computer Security Conference* (Baltimore, MD, USA, 1992), National Institute of Standards and Technology/National Computer Security Center, pp. 554–563.

[12] JOHNSON, B. *Fault-tolerant computer system design.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996, ch. An introduction to the design and analysis of fault-tolerant systems, pp. 1–87.

[13] KALBARCZYK, Z., LYER, R. K., AND WANG, L. Application fault tolerance with Armor middleware. *IEEE Internet Computing 9*, 2 (2005), 28–38.

[14] KARJOTH, G. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and Systems Security 6*, 2 (2003), 232–57.

[15] MARKOFF, J., AND HANSELL, S. Google's not-so-very-secret weapon, 2006.

[16] MOTRO, R. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *Proceedings of the 5th International Conference on Data Engineering* (1989), pp. 339–347.

[17] NETEGRITY. Siteminder concepts guide. Tech. rep., Netegrity, 2000.

[18] NICOMETTE, V., AND DESWARTE, Y. An authorization scheme for distributed object systems. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Oakland, CA, 1997), pp. 21–30.

[19] OMG. Common object services specification, security service specification v1.8, 2002.

[20] RIZVI, S., MENDELZON, A., SUDARSHAN, S., AND ROY, P. Extending query rewriting technique for fine-grain access control. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data* (Paris, France, 2004).

[21] ROSENTHAL, A., AND SCIORE, E. Administering permissions for distributed data: Factoring and automated inference. In *Proceedings of 15th Annual Working Conference on Database and Application Security* (Niagara Falls, Ontario, Canada, 2001), pp. 91–104.

[22] SALTZER, J., AND SCHROEDER, M. The protection of information in computer systems. *Proceedings of the IEEE 63*, 6 (1975), 1278–1308.

[23] SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. Role-based access control models. *IEEE Computer 29*, 2 (1996), 38–47.

[24] SCHAAD, A., MOFFETT, J., AND JACOB, J. The role-based access control system of a European bank: a case study and discussion. In *Proceedings of the 6th ACM symposium on Access control models and technologies (SACMAT)* (2001), pp. 3–9.

[25] SECURANT. Unified access management: A model for integrated web security. Tech. rep., Securant Technologies, June 25 1999.

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

[27] STRONG, P. How Ebay scales with networks and the challenges. In *the 16th IEEE International Symposium on High-Performance Distributed Computing* (Monterey, CA, USA, 2007). Invited talk.

[28] VAIDYA, J., ATLURI, V., AND GUO, Q. The role mining problem: Finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT)* (Sophia Antipolis, France, June20-22 2007), pp. 175–184.

[29] VOGELS, W. How wrong can you be? Getting lost on the road to massive scalability. In *the 5th International Middleware Conference* (Toronto, Canada, October 20 2004). Keynote address.

[30] WEI, Q., RIPEANU, M., AND BEZNOSOV, K. Cooperative secondary and approximate authorization recycling. In *Proceedings of the 16th IEEE International Symposium on High-Performance Distributed Computing (HPDC)* (Monterey Bay, CA, June 27-29 2007), pp. 65–74.