Efficient Access Enforcement in Distributed Role-Based Access Control (RBAC) Deployments

Mahesh V. Tripunitara
ECE Dept., Univ. of Waterloo
Waterloo, ON, Canada
tripunit@uwaterloo.ca

Bogdan Carbunar
ARTC, Motorola Inc.
Schaumburg, IL, USA
carbunar@motorola.com

ABSTRACT

We address the distributed setting for enforcement of a centralized Role-Based Access Control (RBAC) protection state. We present a new approach for time- and space-efficient access enforcement. Underlying our approach is a data structure that we call a cascade Bloom filter. We describe our approach, provide details about the cascade Bloom filter, its associated algorithms, soundness and completeness properties for those algorithms, and provide an empirical validation for distributed access enforcement of RBAC. We demonstrate that even in low-capability devices such as WiFi network access points, we can perform thousands of access checks in a second.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems

General Terms

Security, Algorithms

Keywords

Access Control, Enforcement, Bloom filter, Efficiency

1. INTRODUCTION

Distributed enforcement of a system-wide protection state is an important setting in access control [1, 2, 3, 6, 13, 14]. We consider this setting in the context of Role-Based Access Control (RBAC) [12, 21]. In RBAC users acquire permissions via roles, and the protection state is characterized by the triple $\langle UA, PA, RH \rangle$ where UA is the user-role assignment relation, PA is the permission-role assignment relation and RH is a relation between roles called the role hierarchy. A users creates a session, for which he activates a subset of the roles to which he is authorized. When the user requests the right to exercise a permission, it is done as part of a session, and the enforcement mechanism needs to check whether a role associated with the session is authorized to the permission. We discuss an example of an RBAC protection state and sessions in Section 2.1.

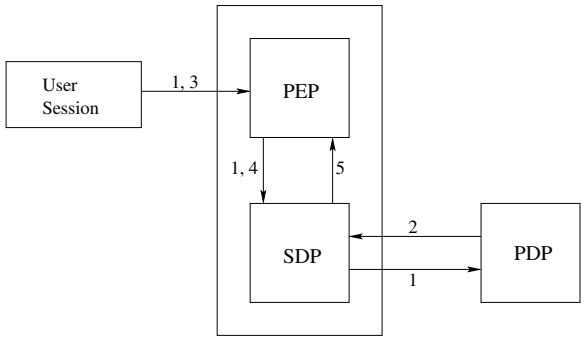
We are not the first to consider such a setting for RBAC; indeed, our work can be seen as a follow-up to the work of Wei et al. [26]. We adopt the setting they propose that we reproduce in Figure 1. The centralized protection state is maintained in a Policy Decision Point (PDP). Such centralization eases administration of the system-wide protection state. Enforcement is performed at distributed sites each of which comprises a Police Enforcement Point (PEP) and a Secondary Decision Point (SDP). The PEP receives and interprets an access request from a user and consults the SDP to resolve whether the request should be allowed or denied. The SDP is sent a portion of the protection state by the PDP, and it uses the portion it has to make a decision. If the SDP is unable to make a decision, the request is forwarded to the PDP.

We address two challenges that such a setting poses for access enforcement in RBAC. One is that we want to avoid reliance on the PDP as much as possible for decisions on individual access requests. The motivation is that the PDP can become a single point of failure in such a situation. Another challenge is efficiency. We want access decisions to be made quickly (say, within milliseconds). It is typical for networks within enterprises to experience propagation delays of tens of milliseconds. If every request must be routed to the PDP, this significantly decreases the frequency of access decisions that can be made.

The use of an SDP that is collocated with a PEP is certainly the start of a solution. We must resolve exactly what data the PDP sends the SDP, and how frequently. We must also resolve how the SDP is updated if changes are made to the protection state stored in the PDP (the well-known problem of "cache consistency"). This is the focus of our work. We propose the use of a data structure that we call a *cascade Bloom filter* (see Section 3) that is based on the Bloomier filter [8]. A cascade Bloom filter is constructed by the PDP and sent to an SDP, and then used by the SDP for access enforcement. Our objective is efficiency of time and space in access enforcement. We demonstrate empirically that access enforcement of RBAC configurations can be performed in constant time and with reasonable space requirements.

Our particular interest is in evaluating the feasibility of our approach for the deployment of an SDP in inexpensive, low capability devices. Network access devices such as WiFi access points (for example [16]) are such devices. Such devices have limited memory (about 64 MBytes), and low processing power (about 200 MHz). As such devices perform other functions as well and are not dedicated to RBAC access enforcement, our assumption is that the space available to us is only a few megabytes.

SACMAT'09, June 3–5, 2009, Stresa, Italy.



1: Session initiation request
2: Access enforcement structure
3: Access request
4: Validated/translated access request
5: Access decision

Figure 1: The distributed access enforcement architecture.

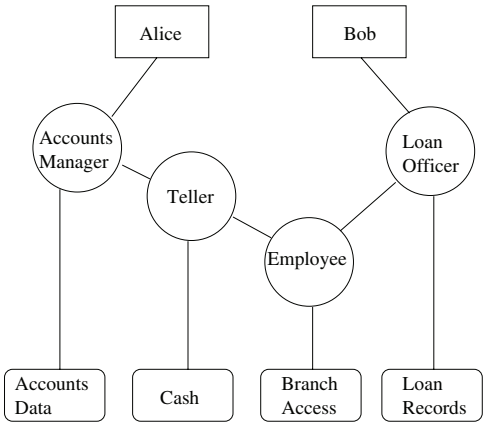


Figure 2: An example of an RBAC state with users $Alice$ and Bob. $Alice$ is assigned to the role $AccountsManager$, which inherits from $Teller$. Bob is assigned to $LoanOfficer$. All other roles inherit from $Employee$. The permission $Branch\ Access$ is assigned to $Employee$, $Accounts\ Data$ is assigned to $Accounts\ Manager$, $Cash$ is assigned to $Teller$ and $Loan\ Records$ is assigned to $Loan\ Officer$.

We have two motivations for considering such devices as our enforcement points. One is that such devices already perform access enforcement in other contexts - for example, firewalling and network adminission control. It seems natural to enhance their capability rather than dedicate a completely separate machine for other kinds of access enforcement. Our second motivation is that we seek to investigate access enforcement for RBAC in novel settings, and we feel that such low capability devices may be used as access enforcement points in situations such as remote or temporary office locations for enterprises.

We seek to also address what we consider a particular shortcoming of the approach of Wei et al. [26] - the issue of "cache warmness." In their approach, a PDP does not push state to the SDP. Rather, if the SDP does not have sufficient state to make an access decision, it forwards the request to the PDP, and then augments its state based on the response. This means that the state at the SDP (the "cache") may not be "warm," i.e., have sufficient information to make decisions on an access request. Our approach, which we discuss in Section 2, is to anticipate requests based on session establishment, and thereby eliminate the issue of cache warmness. Apart from the obvious benefits in terms of response time, this approach is also fair - all legitimate requests associated with a session have the same response time.

Yet another issue regards the encoding of elements such as roles, users, sessions and permissions. We seek to decouple the time and space needed to store what is needed for access enforcement in an SDP as much as possible from the encoding that may be adopted for these RBAC elements. It is in this regard that our data structure that is based on the Bloom filter [5] is particularly useful.

The remainder of the paper is organized as follows. In the next section, we discuss our approach in the setting shown in Figure 1 and present an example of how our approach to access enforcement works. In Section 3, we discuss the cascade Bloom filter, and present algorithms for it. We discuss our implementation and present empirical results in Section 4. We discuss related work in Section 5, and conclude with Section 6.

2. OUR APPROACH

As we show in Figure 1, the user first establishes a session by issuing a request to the PEP. A session establishment request is of the form $\langle U, \{R_1, \ldots, R_n\}\rangle$ where U is the user that attempts to establish the session and $\{R_1, \ldots, R_n\}$ is the set of roles the use wants associated with that session. The PEP forwards this request to the SDP, which in turn forwards it to the PDP. The PDP validates the request - it checks whether U is indeed authorized to $R_1, \ldots, R_n$ in the RBAC protection state.

The PDP then computes an access enforcement structure that the SDP can use to enforce access control. The PDP communicates this to the SDP. The user may then issue access requests for the session. The PEP validates each request before passing it on to the SDP. The SDP uses the access enforcement structures it has from the PDP to evaluate the request and issue an "allow" or "deny" access decision.

We impose no restrictions on how core RBAC elements (users, roles and permissions) are encoded. We feel that this allows for flexibility in RBAC deployments and makes our approach broadly applicable. We assume that sessions are identified using some kind of session ID. We assume also that given a session ID, it is possible to identify the unique SDP with which the session is associated. That is, there exists a function from the set of all valid session IDs to the set of SDPs.

We also do not mandate any format for an access request that is made to a PEP. This is again to allow implementations the flexibility to adopt the best approach for their particular situations. The PEP communicates an access request to its SDP as $\langle s, p\rangle$ where s is the session ID and p is an RBAC permission. To construct an access request of the form $\langle s, p\rangle$, the PEP validates that the requester is indeed bound to the session ID s. The PEP also constructs p from the request it receives. For example, if the request pertains to an object o and a right r, and in the implementation, a permission is a pair $\langle o, r\rangle$, then the PEP constructs p as such for the SDP.

Consequently, an access request received by the SDP is of the form $\langle s, p\rangle$ where s is a valid session ID and $p \in \mathbf{P}$. The SDP then uses the access enforcement structure it has from the PDP to make an access decision.

2.1 An example

We discuss an example to help explain how our approach works. Consider an RBAC protection state as shown in Figure 2. The user $Alice$ initiates a session $s_{1,Alice}$ at a PEP-SDP site and acti-

vates the $AccountsManager$ role for that session. This associates the permissions $Accounts\ Data$, $Cash$ and $Branch\ Access$ with $s_{1,Alice}$. In our approach, the PDP sends a data structure that represents the set $A = \{\langle s_{1,Alice}, Accounts\ Data\rangle, \langle s_{1,Alice}, Cash\rangle, \langle s_{1,Alice}, Branch\ Access\rangle\}$ to the SDP.

If Bob initiates a session $s_{1,Bob}$ and activates the role $LoanOfficer$, then the PDP updates the state at the SDP, with the new state as $A' = A \cup \{\langle s_{1,Bob}, Loan\ Records\rangle, \langle s_{1,Bob}, Branch\ Access\rangle\}$.

Finally, if $Alice$ creates a new session $s_{2,Alice}$ and activates the role $Teller$ in it, then the PDP updates the SDP's state so that it is $A'' = A' \cup \{\langle s_{2,Alice}, Cash\rangle, \langle s_{2,Alice}, Branch\ Access\rangle\}$. If $Alice$'s session with session ID $s_{2,Alice}$ requests an access that requires the permission $Accounts\ Data$, then it is denied by the SDP. This is consistent with the notion of RBAC sessions [12, 21], and indeed, this nuance does not appear to have been captured in previous work [26].

2.2 Our Approach - More Details

We now clarify in more detail how the PDP and the SDP work in our approach. When a user creates a session, the PDP sends the SDP a new cascade Bloom filter, or updates the SDP's existing cascade Bloom filter as its access enforcement structure. The cascade Bloom filter is based on the traditional Bloom filter [5]; we give a precise characterization of it and the associated routines in Section 3. In this section, we give a summary of the operations and how the PDP and SDP use them. We then present the empirical validation of our approach in Section 4. Once the SDP has a new or updated cascade Bloom filter, it is able to service access requests, and issue decisions that are consistent with the RBAC state that is maintained by the PDP.

A cascade Bloom filter is used to represent a set of elements A that is drawn from a universal set (henceforth, simply universe) U. In our case, the set A is one of the following:

Positive authorizations This is the set of doubles $\langle session\ id, permission\rangle$ for valid $session\ id$ and $permission$ that is authorized for that session. In our example shown in Figure 2, if the only session that has been activated is $s_{1,Alice}$, then $A = \{\langle s_{1,Alice}, Accounts\ Data\rangle, \langle s_{1,Alice}, Cash\rangle, \langle s_{1,Alice}, Branch\ Access\rangle\}$.

Negative authorizations This is the set of doubles $\langle session\ id, permission\rangle$ for valid $session\ id$ and $permission$ that are **not** authorized for that session. In our example from Figure 2, if the only session that has been activated is $s_{1,Alice}$, then $A = \{\langle s_{1,Alice}, Loan\ Records\rangle\}$.

The universe, U, is the set of all $\langle session\ id, permission\rangle$ for all valid $session\ id$, and all permissions from the RBAC protection state. In our example from Figure 2, if the valid sessions are $s_{1,Alice}$ and $s_{1,Bob}$, then $U = \{s_{1,Alice}, s_{2,Bob}\} \times \{Accounts\ Data, Cash, Branch\ Access, Loan\ Records\}$, where $\times$ is the cartesian product of two sets. Clearly, whether A is instantiated to be positive or negative authorizations, $A \subseteq U$.

The PDP instantiates A to the smaller of the positive and negative authorization sets. That is, if P is the set of positive authorizations, then A is instantiated to P if $|P| \leq |U-P|$ and to $U-P$ otherwise. The reason is that the cascade Bloom filter is more time- and space-efficient with such a choice; we clarify this further in Section 3.

For the PDP to construct a cascade Bloom filter from A and U, we provide:

`ConstructCascadeBF` - a sound algorithm for creating a cascade Bloom filter from a set A with universe U. We present the algorithm and assert its soundness property in Section 3.

For the SDP to use the cascade Bloom filter provided to it by the PDP, we provide:

`MemberCascadeBF` - a sound and complete algorithm that takes as input a pair $\langle s, p\rangle \in U$ and returns `true` if $\langle s, p\rangle \in A$ and `false` if $\langle s, p\rangle \in U - A$. The SDP then issues an "allow" or "deny" decision based on whether A is the set of positive or negative authorizations. For example, if A is the set of negative authorizations and `MemberCascadeBF` returns `true`, then the request for access is denied. It is important that only members of U are presented as input to `MemberCascadeBF`; its behaviour is undefined for any input not in U. This is ensured by the PEP. We discuss `MemberCascadeBF` and assert its soundness and completeness properties in Section 3.

The access enforcement structure at the SDP may need to be updated, for example, if a new session is created at that site, or if there is a change to the global RBAC state at the PDP that affects the SDP. For this, we provide:

`InsertIntoCascadeBF` - a sound and complete algorithm to add elements to the set A that is represented by the cascade Bloom filter, and also optionally add elements to U. The feature of simultaneously expanding U when expanding A is necessary when a new session is created because U is the cartesian product of valid session IDs and all permissions, and the creation of a new session results in the creation of a new session ID. It is also needed, for example, when a new permission is created and assigned to roles that have been activated in sessions that are currently valid. An example of when it is not needed, is when we authorize a role that has been activated in a session to an existing permission. In this case, A would need to be expanded if it is the set of positive authorizations. We discuss `InsertIntoCascadeBF` and assert its soundness and completeness properties in Section 3.

There is also the complementary operation, `RemoveFromCascadeBF` to shrink A and, optionally, U. Its description is straightforward given our discussions on `InsertIntoCascadeBF`, and we present it and assert its soundness and completeness properties in the technical report version of this paper [25].

3. THE CASCADE BLOOM FILTER

We now provide details of the data structure that underlies our approach, the cascade Bloom filter. We recognize that this data structure may have broader applicability; therefore, our discussions in this section are somewhat more general than in the others. We first discuss the Bloom filter, and the counting Bloom filter data structures that form the bases for the cascade Bloom filter.

A Bloom filter [5] is a space- and time-efficient data structure that is used to represent a set of items, A; we say that the Bloom filter *represents* A. A Bloom filter is an array of m bits and it is associated with a family of *hash functions*, h_i: $U \rightarrow \{0, \ldots, m-1\}$ for $i = 1, \ldots, k$. To store an element $a \in A$ in the Bloom filter, the indices $h_i(a)$ for $i = 1, \ldots, k$ are computed and the corresponding bit in the array is set for each index. To check whether an element e is in A using the Bloom filter, the indices $h_i(e)$ are computed and the value of the bit at each index in the array is tested. If any bit is zero, we deem that $e \notin A$ as represented by the Bloom filter; otherwise we deem that $e \in A$. It is possible that some $e \notin A$ tests positive with the Bloom filter for membership in A.

Such an element is called a *false positive*, and the probability of such an event is called the *false positive rate*. The false positive rate is the same for all elements not in A if the hash functions h_i are uniform - each element in the domain has the same probability of mapping to the different elements in the range. Uniform hash functions can be realized in practice [18]. Given the uniformity assumption for the hash functions, and with an optimal value for k, a Bloom filter with a false positive rate of ϵ can be realized with $log_2(e) \times log_2(1/\epsilon) \approx 1.44 log_2(1/\epsilon)$ bits per element to be stored in the Bloom filter [5].

Adding elements to a Bloom filter is easy, but removing elements is not. We address this issue by adopting a counting Bloom filter [11]. A counting Bloom filter is an array of m counters. The initial value of each counter is 0. To insert an element, e, we compute the indices $h_i(e)$ for $i = 1, \ldots, k$, and increment each counter by 1. To remove an element, we compute the indices h_i and decrement each by 1. To test for membership, we compute the indices h_i and check that the counter at every index is at least 1. In this manner, the counting Bloom filter trades off space for ease of removal. Counters can be implemented in $O(log\,(C))$ space, with constant-time increment, decrement and check for 0 operations, where C is the maximum value we expect to store in the counter [9, 19]. Henceforth, when we refer to a Bloom filter, we mean a counting Bloom filter.

We assume that the following functions related to Bloom filters are available to us. (We use an object-oriented style; these methods are invoked on a Bloom filter object.)

`CreateBF`(A, U, m, E) - instantiates the Bloom filter to be of m bits, and represents the set $A \subseteq U$, with the set of false positives $E \subseteq U - A$. We call U the *universe*. We assume that each counter within the Bloom filter is allocated a fixed number of bits (e.g., 8) that is specified a priori. We expect (in a probabilistic sense) that $|E| = \epsilon|U - A|$ where ϵ is the false positive rate of the Bloom filter. We assume that the parameter k (number of hashes) is determined and stored internally by the Bloom filter. Our choice of k in our algorithm that constructs a cascade Bloom filter (see Algorithm 2) is the one that minimizes ϵ.

`MemberBF`(e) - returns `true` if e is in the Bloom filter and `false` otherwise.

`AddToBF`(e) - adds the element e to the Bloom Filter.

`RemoveFromBF`(e) - removes e from the Bloom Filter.

To achieve our goal of encoding a set of elements in a space efficient manner, but without incurring the penalty imposed by the false positive rate of a Bloom filter, we employ a cascade of Bloom filters. The idea of using multiple Bloom filters has been proposed before [8, 10]. Indeed, what we call a cascade Bloom filter can be seen as an adapted version of the Bloomier filter [8]. We discuss in what way we have adapted the Bloomier filter for our purposes in Section 5 on related work. We certainly do not require the generalizations that are discussed by Chazelle et al. [8] (for example, the ability to test for membership in arbitrary functions). Therefore, we provide efficient algorithms that we feel are suited to the purpose of access checking in this section.

We specify a *depth* d and a Bloom filter at each *level* $1, \ldots, d$. We denote the Bloom filter at level i as BF_i and the set that BF_i represents as A_i. The Bloom filter BF_1 at level 1 represents the set A from the universe U; that is, $A_1 = A$. A_2 is the set of elements from $U - A_1$ that are false positives of BF_1. Subsequently, each A_i comprises those elements from A_{i-2} that are false positives of BF_{i-1}. We specify the elements of A_{d-1} that are false positives of B_d as an explicit list stored, for example, as a hashtable; we call this set A_{d+1}. Figure 3 illustrates the structure of a set A with universe U represented by a cascade Bloom filter of depth 4.

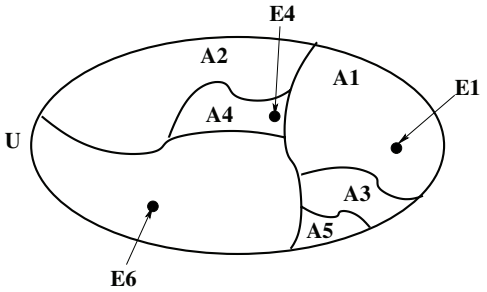


Figure 3: How the universe U is divided into the A_i sets for a cascade Bloom filter with depth 4. Each A_{i+1} is the set of false positives from A_{i-1} in the Bloom filter that represents A_i, with $A_0 = U$. We point out that $A_1 \supseteq A_3 \supseteq A_5$, and $A_2 \supseteq A_4$. The last set, A_5 does not have a corresponding Bloom filter, but is instead represented explicitly (e.g., as a hashtable). The element $E_1 \in A_1$, and none of the other sets. Consequently, a test for its membership evaluates to `true` for BF_1 and `false` for BF_2. The element $E_4 \in A_4 \subseteq A_2$. Therefore, a test for its membership evaluates to `true` in all of $BF_1, \ldots, BF_4$, and the fact that $E_4 \notin A_5$ establishes that it is not a false positive of BF_3 and therefore $E_4 \notin A_1 = A$. Finally, the element $E_6 \notin A_i$ for any A_i, and therefore its test for membership evaluates to `false` in BF_1.

To see the advantage of a cascade of Bloom filters, we consider an example. Let A and U be such that $|A| = 400$, and $|U| = 1000$. If we restrict our space usage to 2500 bits for the Bloom filter, then the best false positive rate we can achieve is about 0.05. We achieve this for $k = 4$. This results in an explicit list to deal with false positives of approximately 30.

We can instead use a cascade of depth $d = 2$, with the Bloom filter B_1 at level 1 encoding the set $|A|$, and the Bloom filter B_2 at level 2 encoding the false positives of B_1 from $U - A$. If we allocate 2000 bits for B_1 and 500 bits for B_2, we have the same total number of bits allocated for Bloom filters as for the single Bloom filter above. Also, we specify that the false positive rate for B_2 is 0.05; this results in an explicit list of size 20, which is 33% more space efficient than the first approach. We observe that we can achieve this with number of hashes for B_1 as $k_1 = 2$, and number of hashes for B_2 as $k_2 = 2$. In other words, we achieve space efficiency with no additional hashes. The reason is that we have a false positive rate $\epsilon_1 \approx 0.105$ for B_1. This in turn results in the set represented by B_2 conaining about 63 elements, which can be done with $\epsilon_2 \approx 0.05$ with 500 bits allocated for B_2.

We now provide a more precise definition for a cascade Bloom filter. In the next section, we present the algorithms that we associate with cascade Bloom filters, to which we referred in Section 2.2.

DEFINITION 1 (CASCADE BLOOM FILTER). *A cascade Bloom filter C is $\langle \mathcal{B}, E \rangle$, where $\mathcal{B} = \langle BF_1, \ldots, BF_d \rangle$ is a list of Bloom filters and $E \subseteq U$ is a set of elements from a universe U. d is called the depth of the cascade and each $l = 1, \ldots, d$ is called a level. Each BF_i represents a set $A_i \subseteq U$, such that for $i = 2, \ldots, d$, A_i is the set of false positives from A_{i-2} in BF_{i-1}, with $A_0 = U$. The set E is the set of false positives from A_{d-1} in BF_d. We say that C represents $A = A_1$ with universe U.*

3.1 Operations on Cascade Bloom Filters

In this section, we present the MemberCascadeBF, ConstructCascadeBF and InsertIntoCascadeBF routines. We also discuss the soundness and completeness properties for them.

Algorithm 1 An algorithm to verify membership of an element e in the set represented by a cascade Bloom filter.

```
1  Operation MemberCascadeBF(e)
2      # Invoked on a cascade Bloom filter which
3      # has data field BF[ ], the list of Bloom
4      # filters, and E, the set of elements.
5
6      for level := 1 to d do
7          if (BF[level].MemberBF(e) = false) then
8              if level is even then return true;
9              else return false;
10             fi
11         fi
12     od
13     if d is even then
14         if E.contains(e) then return true;
15         else return false;
16         fi
17     else
18         if E.contains(e) then return false;
19         else return true;
20         fi
21     fi
```

MemberCascadeBF(e). Algorithm 1 presents our algorithm for checking membership of $e \in U$ in a cascade Bloom filter $C = \langle \mathcal{B}, E \rangle$. In lines 6–12 we iterate through each level of $\mathcal{B}$. If, at any level, e tests false for membership, we are able to immediately make an inference about the membership of e in A. If the level at which false is returned is even, then we know that $e \notin U - A$, and therefore $e \in A$. Conversely, if the level at which false is returned is odd, then we have two cases. One case is that the level is 1, in which case we know that $e \notin A$ (a Bloom filter has no false negatives). If the level > 1, then we know that is not a false positive of $BF_{\texttt{level}-1}$ and therefore $e \in U - A$.

If, at all levels, the Bloom filters return true in lines 6–12, then we test for membership in the list E. Again, we decide e's membership in A based on whether E is at an even or odd level. We emphasize that if MemberCascadeBF returns true, this does not necessarily mean that the access decision on $e = \langle s, p \rangle$ is "allow." As we mention in Section 2.2, the SDP then checks whether A comprises positive or negative authorizations, and the final "allow" or "deny" decision is based on this as well.

We now assert the soundness and completeness property of MemberCascadeBF.

THEOREM 1. *For a cascade Bloom filter $C = \langle \mathcal{B}, E \rangle$ that represents A with universe U, C.*MemberCascadeBF(e) *is:*

Sound - *if it returns* true *then $e \in A$, and if it returns* false *then $e \in U - A$.*

Complete - *if $e \in A$ then it returns* true*, and if $e \in U - A$ then it returns* false*.*

The proof for the theorem is in [25]. The intuition is expressed in our description of the algorithm above.

ConstructCascadeBF(A, U, m, $Elen$). Algorithm 2 is our algorithm for constructing a cascade Bloom filter. It encodes a heuristic, and in lines 36–39 attempts to find a cascade Bloom filter of some depth upto a maximum. It is constrained by the inputs m and $Elen$ that specify the maximum number of counters that may be used for the Bloom filters, and the maximum size of the final list of false positives, respectively. We fix the sizes of counters to a constant value a priori.

Algorithm 2 An algorithm for creating a cascade Bloom filter based on the sizes of A and U, assuming uniform hash functions. ConstructCascadeBF is invoked from the outside. ConstructCascadeBF in turn invokes tryToConstruct.

```
30 Operation ConstructCascadeBF(A, U, m, Elen)
31     # m is the total number of bits for Bloom filters,
32     # Elen is the max length of E.
35
36     for depth := 1 to MAX_DEPTH do
37         tryToConstruct(1, depth, |A|, |U|, m, Elen);
38         On success goto line 42;
39     od
40     return failure;
41
42     # On success for some depth
43     A_0 := U;  A_1 := A;
43     for level := 1 to depth do
44         for each e ∈ A_level do
45             BF_level.AddToBF(e);
46         od
47         for each e ∈ A_level−1 − A_level do
48             if BF_level.MemberBF(e) then
49                 if level = depth then E.add(e);
50                 else
51                     BF_level+1.AddToBF(e);
52                     A_level+1.add(e);
53     od od fi  fi

60 Operation tryToConstruct(level, depth, Asize,
61                          Usize, m, Elen)
62     # try to construct a Bloom filter
63     # at level and below
64
65     if m ≤ 0 then return failure;
66     trials := ⌊m/Asize⌋ − 1;
67     for i := trials − 1 to 0 do
68         m_i := (⌊m/Asize⌋ − i) × Asize;  # m_i ≥ 2 × Asize
69         k_i := round(ln 2 × m_i/Asize);  # minimizes ε_i
70         ε_i := (1 − e^(−k_i × Asize/m_i))^k_i ;
71         newAsize := ⌈ε_i × (Usize − Asize)⌉;
72         if level = depth then
73             if newAsize < Elen then
74                 construct Bloom filter of m_i bits and k_i
75                 hashes at level;  set |E| := newASize;
76                 return success;
77             fi
78         else
79             if tryToConstruct(level + 1, depth, newAsize,
80                 newAsize + Asize, m − m_i, Elen) succeeds
81             then construct Bloom filter of m_i bits and k_i
82                 hashes at level and return success;
83     od fi  fi
84     return failure;
```

The method tryToConstruct is invoked recursively. Each execution of tryToConstruct allocates counters for the Bloom filter at a particular level, and identifies the expected size of the set that needs to be represented at the next level. This it does, in line 70, by estimating the false positive rate for the current level. Our choice of number of hashes in line 69 is always the closest integer that minimizes the false positive rate. Our choice for the number of counters to be allocated at each level in line 68 is always a multiple of the number of elements in the set to be represented, with a number of double the size of that set. This is part of our heuristic. The recursion in tryToConstruct bottoms out in lines 72–77. If

the condition $newAsize \leq Elen$ in line 73 evaluates to `true`, this means that the size of the set to be represented is at most the maximum size allowed by the argument $Elen$, which in turn means that the algorithm has successfully identified a cascade Bloom filter that satisfies the constraints.

We now assert the soundness property of `ConstructCascadeBF`.

THEOREM 2. `ConstructCascadeBF` *is* **sound** *- if it returns successfully with a cascade Bloom filter* $C = \langle \mathcal{B}, E \rangle$ *that represents* A *with universe* U*, with* m *total bits used by all the Bloom filters in* $\mathcal{B}$*, then it was invoked as* `ConstructCascadeBF` $(A, U, M, Elen)$ *where* $M \geq m$ *and* $Elen \geq |E|$.

The proof for the theorem is in [25]. We point out that `ConstructCascadeBF` is not complete; there may be inputs for which a cascade Bloom filter exists, but the algorithm does not find it. We see a complete algorithm for the construction of a cascade Bloom filter as one of multivariate optimization, and leave an investigation into the exact complexity (or whether it is even decidable in general) as future work.

InsertIntoCascadeBF($level, I, U'$**).** In Algorithm 3 we present our algorithm for the elements in the set I into a cascade Bloom filter. The new universe U' is also provided as input; U' is allowed to be a superset of the current universe U, and this is also incorporated in to the cascade Bloom filter that is returned by `InsertIntoCascadeBF`.

Each execution of `InsertIntoCascadeBF` processes two levels at a time; an odd level and the next even level. The exceptions are when only the explicit list, E, at the end remains to be processed. This latter case is addressed by lines 100–105. In lines 107–112, the algorithm adds new elements from I into the set $A_{\texttt{level}}$ and the Bloom filter $BF_{\texttt{level}}$. Note that we refer to the sets A_i that are represnted by the Bloom filters at each level for convenience only. The PDP may or may not explicitly store these sets. If it does not store these sets, it can infer these sets based on the cascade Bloom filter when it needs to insert new elements (i.e., invoke `InsertIntoCascadeBF`). This would be a straightforward trade-off between time and space.

After inserting element at the current layer, what the algorithm does in lines 113–142 is consider the next (even) level. Two things need to be done at the next level. One is to add any new false positives as a result of adding elements from I to this level. Second is to remeove any old false positives that are no longer false positives. An example of such an element is a member of I.

Finally, in lines 144–155, algorithm recurses through the remainder of the cascade. It sets up the corresponding I (line 148) and U' (line 151) sets for the ensuing levels. We now assert the soundness and completeness properties of `InsertIntoCascadeBF`.

THEOREM 3. *Given a cascade Bloom filter* C *that represents* A *with universe* U*,* C.`InsertIntoCascadeBF` `(e)` *is:*

Sound *- if it returns with* C *that represents* A' *with universe* U'*, then it was invoked as* `InsertIntoCascadeBF` $(1, (A' - A) \cup S, U')$*, where* $S \subseteq A$.

Complete *- if it is invoked as* `InsertIntoCascadeBF` $(1, I, U')$ *where* $A \cup I \subseteq U'$*, then it returns with* C *that represents* $A \cup I$ *with universe* U'.

The proof for the theorem is in [25]. We use induction on the number of recursions we perform. We observe that the algorithm is proportional in time to the size of the new universe, U'.

Algorithm 3 Algorithm to insert elements into the cascade Bloom filter. The parameter $level$ is the level at which to begin insertion. The operation is invoked at first with $level = 1$; recursive invocations invoke the operation with $level > 1$. I is the set of elements to be inserted and U' is the, possibly new, universe.

```
90  Operation InsertIntoCascadeBF(level, I, U')
91      # Insert the elements of I into a cascade Bloom
92      # filter that represents A from universe U.
93      # The set U' ⊇ U is the new universe.
94      # It is first invoked with level = 1.
95
96      if (level is even or level > d + 1) return error;
97      if I ⊈ U' return error;
98
99      # If this is the last layer
100     if (level = d + 1) then
101         for each e ∈ I do
102             if (e ∉ E) then E.add(e); fi
103         od
104         return;
105     fi
106     # Add I to this layer
107     for each e ∈ I do
108         if (e ∉ A[level]) then
109             A[level].add(e);
110             BF[level].addToBF(e);
111         fi
112     od
113     # Now do the next level as well
114     # If this is the last layer
115     if (level = d) then
116         for each e in I do
117             if (e ∈ E) then E.remove(e); fi
118         od
119         # Consider new false positives
120         F := U' − (A_level ∪ E) ;
121         for each e in F do
122             if (BF[level].MemberBF(e)) then E.insert(e); fi
123         od
124         return;
125     fi
126     # Remove false positives that are no longer so
127     for each e in I do
128         if (A[level + 1].contains(e)) then
129             A[level + 1].remove(e);
130             BF[level + 1].RemoveFromBF(e);
131         fi
132     od
133     # Add new false positives
134     F := U' − (A_level ∪ A_level+1) ;
135     for each e in F do
136         if (BF[level].contains(e)) then
137             if (e ∉ A[level + 1]) then
138                 A[level + 1].add(e);
139                 BF[level + 1].AddToBF(e);
140             fi
141         fi
142     od
143     # Prepare for recursion
144     I' := ∅;
145     # Generate set to be inserted in A[level + 2]
146     for each e in A[level] do
147         if (BF[level + 1].MemberBF(e)) then
148             I' := I' ∪ e;
149         fi
150     od
151     U'' := I' ∪ A[level + 1];
152     # Recurse
153     if (I' ≠ ∅ || U'' ≠ ∅) then
154         InsertIntoCascadeBF(level + 2, I', U'');
155     fi
```

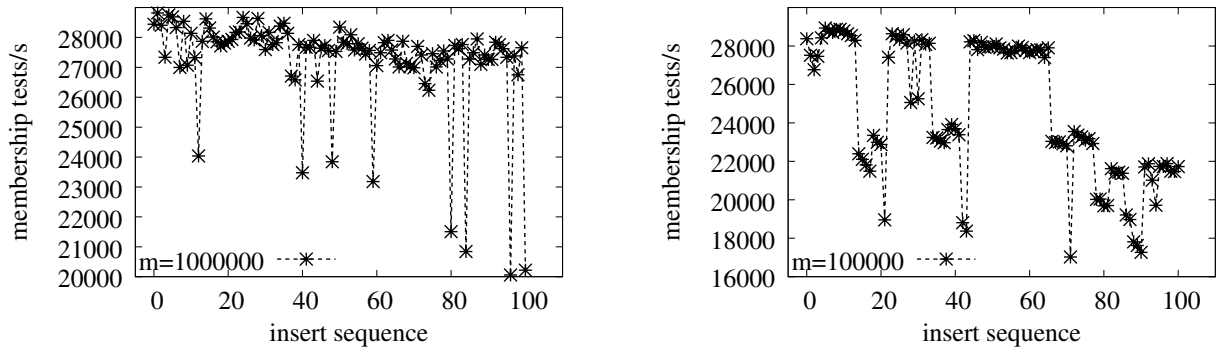


Figure 4: Evolution in time of the number of membership queries an SDP can perform, as the number of sessions in the system increases from 1 to 100, for (a) the first strategy, m=1 million counters and (b) the second strategy, m=100000 counters.

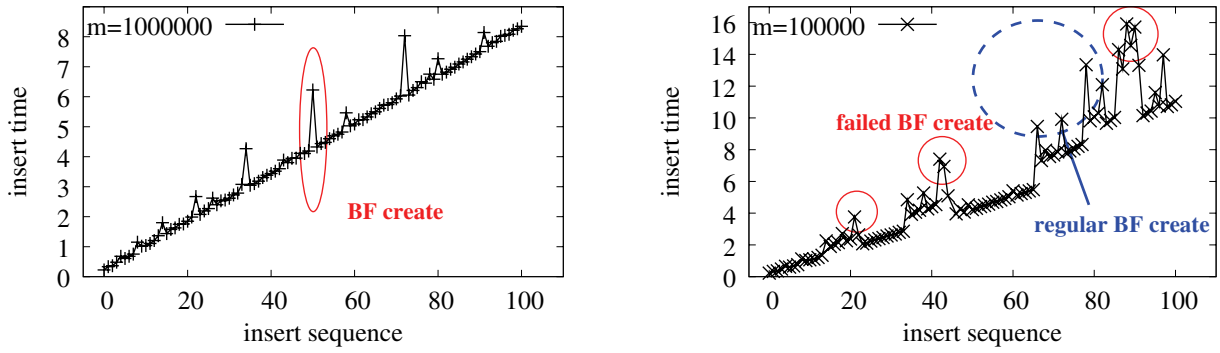


Figure 5: Evolution in time of the time to initiate a new session. The x axis shows the insertion index (between 1 and 100 sessions) and the y axis shows the time to perform an insertion when: (a) the cascade Bloom filter is initialized with a total of 1 million counters for the Bloom filters and the maximum allowed size of the explicit list is 2000 elements. The red ovals indicate calls to the create function. (b) the cascade Bloom filter is initialized with a total of 100000 counters for the Bloom filters and the maximum allowed size of the explicit list is 2000 elements. The blue, dotted ovals indicate several regular calls of the create function. The red, plain ovals indicate the 3 calls that fail to create a cascade structure satisfying the current requirements.

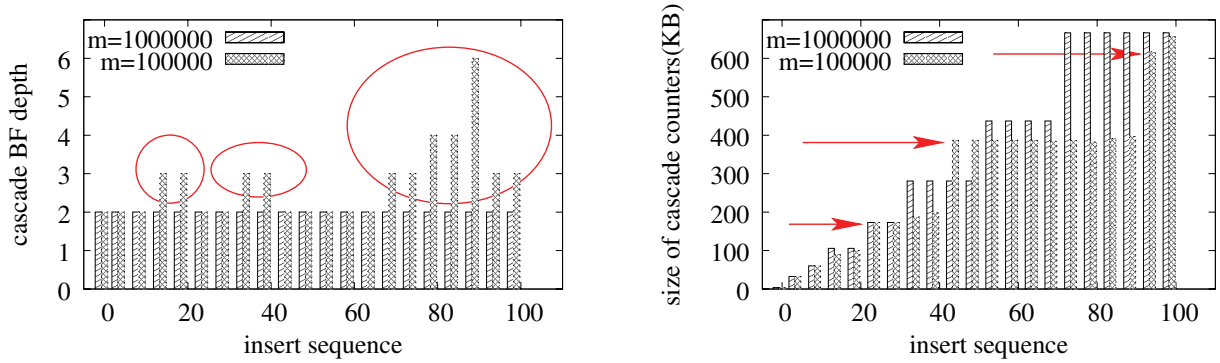


Figure 6: Evolution in time for the two strategies of the (a) cascade Bloom filter depth and (b) size of the counting Bloom filters. The read ovals and arrows show the points where calls to the create function fail for the second strategy.

A complementary operation is one to remove elements from the set A that is represented by a cascade Bloom filter, and optionally shrink the universe U. This is necessary when a session is terminated by a user. We present it in [25]. It is a straightforward complement of the `InsertIntoCascadeBF` algorithm. One difference is that it does not need to take the new (possibly smaller) universe as a parameter as no new false positives are introduced when elements are removed. Consequently, we can infer the new universe simply from correctly removing any elements that are no longer false positives as a consequence of the removal of elements.

4. EMPIRICAL VALIDATION

We have implemented the cascade Bloom filter and its associated algorithms, and tested it for various RBAC protection states and session profiles. We present some of these results here. Our baseline test is an RBAC protection state with 100 users, 3,000 permissions and 50 roles – the same baseline used by Wei at al [26]. We have also tried our algorithms on larger RBAC sizes; for example, each of the users, permissions and roles an order of magnitude larger.

Our tests were conducted on a commodity laptop personal computer that has an Intel Celeron processor that clocks at 1.85 GHz, and has 1.25 MBytes of RAM running a Linux distribution. The code is in Java, and runs on Sun's 1.5.0 Java Runtime Environment (JRE). To simulate a lower capability machine such as a WiFi network access point, we used the `cpulimit` program [23]. We used SHA-512 [24] as our hash function.

Comprehensive performance results, and a comparison with other approaches that are tuned for high performance such as CPOL [6] are beyond the scope of this paper. We also do not include comparisons to more basic approaches, such as the traditional Bloom filter. The reason is that our choice of the Cascade Bloom filter is motivated by other considerations than performance — specifically, that it saves space over a traditional Bloom filter. Such comparisons are certainly valid and interesting, and are topics for subsequent work.

4.1 Performance at the SDP

Of primary concern, of course, is the performance of `MemberCascadeBF` at an SDP. This is the routine that is used to perform access enforcement.

Our observations for the baseline RBAC protection state are shown in Figure 4. We begin with $|U| = 3,000$ and $|A| = 600$, and with each insertion (shown on the horizontal axis), we increment U by $3,000$ and A by 600. That is, we assume that each user is authorized to 600 permissions that he activates in a session. We have tried cascade Bloom filters with both 1 million and 100,000 counters. The space consumed by the former is about 4 MBytes, and the latter is about 400 KBytes. We allocated an explicit list, E, of length 2000.

Our main observation from Figure 4 is that access enforcement is very fast, even for the smaller number of counters. We find that we are able to check for membership in about 50μs. When we dampen the processing power using `cpulimit` to 10%, we observe that the speed falls commensurately, i.e., to about 10%. This still means that we are able to make one access decision is less than 1 ms. That is, even if we were to support up to 100 sessions on a low-capability device such as a WiFi network access point, we would be able to processes thousands of access requests per second. This compares favourably to today's network access devices such as packet filters and firewalls, and is therefore encouraging from the standpoint of prospects for distributed deployments of RBAC.

We point out that our results are about 1 to 2 orders of magnitude better than what is reported by Wei et al. [26], given that their tests were conducted on a more powerful machine. The reason is, of course, that we use the cascade Bloom filter, while there, our understanding is that the built-in set data structures in Java were used.

We observe that both graphs in Figure 4 follow a slow downward slope, with rapid dips at some points. These low points correspond to when the `ConstructCascadeBF` returned a cascade Bloom filter of higher depth. That is, there is a correlation between the depth of the cascade Bloom filter and the number of membership queries we are able to process.

4.2 Performance at the PDP

While the performance at the SDP is of most concern, we are also concerned about the performance of the algorithms at the PDP. We have followed the same two stratgies at the PDP as at the SDP. In one strategy the cascade Bloom filter is created by allowing it to allocate as much as 1 million counters for the Bloom filters, while reserving up to 2,000 entries for elements to be stored in the explicit list. In the other strategy we used the same maximum size of 2,000 for the explicit list, but we allowed only up to 100,000 counters for all the counting Bloom filters in the cascade. For both strategies, whenever the insertion of a new session, i.e., a invocation of `InsertIntoCascadeBF`, exceeds the size of the explicit list, a new cascade Bloom filter is created with an invocation to `ConstructCascadeBF`.

Figure 5 shows the evolution of the time to insert a new session for the two strategies, as sessions are generated by users. The first strategy, of $m = 10^6$ is shown in Figure 5(a). The spikes indicated in red for the first strategy denote the moments when the size of the explicit list is exceeded, leading to the creation of a new Bloom filter with an invocation to `ConstructCascadeBF`. Calls to create a new cascade Bloom filter never failed for the first strategy. Calls to `ConstructCascadeBF` are more expensive than calls to `InsertIntoCascadeBF`; however, they occur infrequently. The overall increase in time to perform an `InsertIntoCascadeBF` is linear and the increment between consecutive inserts that occur between two spikes is very small – around 200ms. This increment is due mainly to the fact that whenever a new session is created, 600 elements are inserted into the top level Bloom filter of the cascade, leading to increased false positive rates in that level and the subsequent levels. Then, more elements from the universe that are not inserted in the top level, will need to be inserted in the cascade's second level.

Figure 5(b) shows the evolution of the time to insert for the second strategy, where initially m is 100,000. On three occasions (denoted with red circles) the creation of a new cascade filter fails, leading to the allocation of more space for the counting Bloom filters (we double m each time a failure of `ConstructCascadeBF` happens). The reason for this is that in the second strategy, the invocation to `ConstructCascadeBF` receives a smaller upper bound value for the space dedicated to counters than the first strategy. However, the approach of severely limiting the space allocated for counters leads to more frequent re-allocations (invocations to `ConstructCascadeBF`) of the cascade Bloom filter. In the end, this leads to larger values for the time to insert a new session. Specifically, while in the first strategy the time to perform an `InsertIntoCascadeBF` ranges between half a second to 8 seconds, for the second strategy the maximum time is around 10 seconds.

It is discouraging that by the 100^{th} session, the PDP takes between 8 and 14 seconds to update the cascade Bloom filter. This means that a user must wait at least that long before he is able to

exercise his permissions. Improving this performance is certainly a topic for future work. We should point out, also, that our tests regarding the PDP were conducted on the same 1.85 GHz laptop with 1.25 MBytes of RAM. It is reasonable to assume that a commercial PDP will be more powerful, and therefore decrease the time significantly.

Figure 6(a) compares the two strategies on the evolution in time of the depth of the cascade Bloom filter for this experiment. For each data point on the horizontal axis that represents the index of a session insertion, we represent two values (bars) one being the depth of the cascade structure for the first strategy (left bar) and one for the second strategy (right bar). The depth of the cascade Bloom filter generated in the first strategy, where m is set initially to 1 million counters, is always 2. That is, `ConstructCascadeBF` always has enough space to allocate to the counting Bloom filter of the first level of the cascade, and can afford to have just one more level, which is the explicit list. However, the sparse memory available for counters in the second strategy requires `ConstructCascadeBF` to allocate more levels in the cascade Bloom filter. In the worst case for the scenarios we studied, the depth is 6. However, it quickly drops to 3 due when `ConstructCascadeBF` is (re-) invoked. The spikes in depth of the cascade structure correspond to when `ConstructCascadeBF` fails.

The higher depth values of the cascade Bloom filter generated by the second strategy is an important factor in explaining the higher times for insertion displayed in Figure 5(b). In the second strategy an element may have to be inserted into up to 6 levels of the cascade structure, whereas in the first strategy an element may be inserted in at most two levels. However, the more economic use of memory of the second strategy pays off when it comes to the total size of the counting Bloom filters. Figure 6(b)shows the total space allocated to counting Bloom filters for the two strategies that we considered. It shows that throughout our experiment, the second strategy (second bar for each data point) consistently requires less space, which is often half the space required by the first strategy (left bar).

In conclusion, each strategy has advantages, which have to be carefully weighed when designing the RBAC access enforcement system. If speed of inserting a new session (session initiation) is important, the first strategy should be chosen. If however, the space at the SDP, where the counting Bloom filters will be stored, is of concern, the second strategy should be the choice. Certainly, other strategies do exist. For instance, instead of doubling the value of m whenever the create function fails, the value of m could be incremented by a constant or increased by a factor higher than 2. While the first approach would be more space-efficient, the second approach would most certainly reduce the time required to insert a new session.

We should mention also that we implemented two optimizations at the PDP with regards to cascade Bloom filters. First, whenever an element is inserted, removed or its presence in the counting Bloom filter at any level in the cascade structure is checked, the positions corresponding to that element in the filter, derived from the invocation of a SHA-512 cryptographic hash, are stored locally. Subsequently, if that element is ever accessed at that level, its positions in the counting Bloom filter are no longer computed using relatively expensive SHA-512 invocations, but instead are retrieved from the local storage. This is a straightforward trade-off of time and space at the PDP.

The other optimization is that of reducing the number of SHA-512 calls from k to 1. SHA-512 returns a 64 byte hash. If we only need 4 bytes to generate a position in a counting Bloom filer, a single SHA-512 call can handle k values smaller than or equal to 16. For larger k values we need to invoke SHA-512 only $\lceil k/16 \rceil$ times. As it turns out, in our experiments the value of k for any level in the cascade structure never exceeded 16.

5. RELATED WORK

There has been considerable research in distributed deployments for access control. See for example [1, 2, 3, 6, 13, 14]. Moreover, as Wei et al. [26] point out, previous work has considered caching at the PEP. However, the caching is based on individual access requests. In our case, the PDP proactively pushes out the entire portion of the state that pertains to a session at the SDP. While this can be seen as a form of caching, it is quite different from what has been considered in the past. Also, to our knowledge, only Wei et al. [26] consider such distributed access enforcement for RBAC.

Their work is indeed the closest to ours, and our work can be seen as a follow-up, which specifically addresses the time- and space-efficiency issues around such distributed access enforcement. The focus of Wei et al. [26] is on authorization recycling, and not performance. And, as we point out in Section 4.1, our access enforcement performance is an order to two orders of magnitude better than the results they report for similar RBAC protection states and request profiles. Our approach also does not suffer from the issue of "cache warmness," as we discuss in Section 1.

The other piece of work that is related to ours regards Bloom filters. The original work is due to Bloom [5], and since then, there has been a lot of work that analyzes and extends Bloom filters, for example [7, 8, 10, 11, 17, 19, 20, 22]. Our work builds upon counting bloom filters [11]. Our work is most closely related to that of Chazelle et al. [8] on Bloomier filters. Indeed, the cascade Bloom filter we discuss in Section 3 can be seen as a specialization of the Bloomier filter. The most significant difference is that we explicitly consider the universe, U. A consequence of this is that while the Bloom filters get exponentially larger in a Bloomier filter as we go down levels, ours get smaller. Their exponential increase can be attributed to their desire for a fast convergence to a desired false positive rate. Our setting does not appear to require such fast convergence; our empirical studies suggest that we achieve fairly fast convergence in any case for our application of distributed RBAC enforcement.

Another difference between our discussion of cascade Bloom filters and the work on Bloomier filters [8] is that their objective is to represent and test for membership in an arbitrary function, while our goal is for a binary check for access. A consequence is that we are able to present pragmatic algorithms with soundness and completeness properties for creation of and insertion into a cascade Bloom filter. It is unclear to us how pragmatic the algorithms for the general Bloomier filter are.

There has also been work on high-performance policy evaluation, notably, CPOL [6]. Borders et al. [6] compare CPOL to KeyNote [4], and it is unclear in what way CPOL would support RBAC, rather than a Trust Management system. As indicated by Li and Tripunitara [15], the mapping of a Trust Management scheme to RBAC can be non-trivial.

6. CONCLUSIONS

We have addressed the issue of time- and space-efficient access enforcement of a centralized RBAC protection state. We have proposed the use of a novel data structure, the cascade Bloom filter, for this. We have presented algorithms to manipulate cascade Bloom filters and asserted their soundness and completeness properties. We have empirically validated our approach with example RBAC configurations, and demonstrated that even low-capability devices can perform up to a thousand access checks per second.

There are a number of issues that remain to be investigated in future work. A rather foundational issue regards the complexity, in an asymptotic sense, of the multivariate optimization problem that we need to solve to create a cascade Bloom filter to which we allude in Section 3.1. Assuming that the problem is not undecidable in general, we seek an efficient, complete algorithm for creating a cascade Bloom filter. Another issue regards the efficiency at the PDP. As we point out in Section 4.2, the time for creation and insertion is somewhat discouraging. We first need to investigate how much difference a more powerful machine than the one on which we tested would make.

With regards to performance, as we discuss in Section 4, a comprehensive comparison with other approaches such as CPOL [6] and the traditional Bloom filter are certainly called for. While those issues are beyond the scope of this paper, they make for interesting future work. Still another issue is the protocol that the PDP uses to communicate with the SDP. For example, it may be more efficient for the PDP, in some cases, to have the SDP update its cascade Bloom filter. In other cases, it may be more efficient to have the SDP completely replace its cascade Bloom filter with a new one. A thorough study of such issues is a topic for future work.

Acknowledgements

We thank Mike Atallah of Purdue's Computer Science department for suggesting the use of Bloom filters for this problem. We thank also the anonymous reviewers for their very useful feedback.

7. REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Oct. 1993.

[2] E. Al-Shaer, H. Hamed, R. Boutaba, and M. Hasan. Conflict classification and analysis of distributed firewall policies. *IEEE Journal on Selected Areas in Communications (JSAC)*, 23(10), October 2005.

[3] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access- control systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 81–95, 2005.

[4] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, Sept. 1999.

[5] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

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

[7] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. In *Proceedings of the 40th Annual Allerton Conference on Communication, Control and Computing*, pages 636–646. ACM Press, 2002.

[8] B. Chazelle, J. Kilian, R. Rubinfield, and A. Tal. The bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 30–39, 2004.

[9] M. J. Clancy and D. E. Knuth. A programming and problem-solving seminar. Technical Report CS-TR-77-606, Stanford University, Department of Computer Science, 1977.

[10] S. Cohen and Y. Matias. Spectral bloom filters. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 241–252, 2003.

[11] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[12] D. F. Ferraiolo, R. S. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and Systems Security*, 4(3):224–274, Aug. 2001.

[13] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti. dRBAC: Distributed Role-Based Access Control for Dynamic Coalition Environments. In *Proceedings of the International Conference on Distributed Computing Systems*, July 2002.

[14] G. Karjoth. Access control with IBM Tivoli Access Manager. *ACM Transactions on Information and System Security*, 6(2):232–257, May 2003.

[15] N. Li and M. V. Tripunitara. Security analysis in role-based access control. *ACM Transactions on Information and Systems Security (TISSEC)*, 9(4):391–420, Nov. 2006.

[16] LinkSys. The wireless-g access point - wap54g. `http://www.linksysbycisco.com/US/en/support/WAP54G`, 2009.

[17] M. Mitzenmacher. Compressed bloom filters. *IEEE/ACM Transactions on Networking*, 10(5):604–612, 2002.

[18] A. Pagh and R. Pagh. Uniform hashing in constant time and optimal space. *SIAM Journal on Computing*, 38(1):85–96, Mar. 2008.

[19] A. Pagh, R. Pagh, and S. S. Rao. An optimal bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 823–829, 2005.

[20] M. V. Ramakrishna. Practical performance of bloom filters and parallel free-text searching. *Communications of the ACM*, 32(10):1237–1239, Oct. 1989.

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

[22] K. Shanmugasundaram, H. Bronnimann, and N. Memon. Payload attribution via hierarchical bloom filters. In *Proceedings of the 11th ACM conference on Computer and communications security (CCS'04)*, pages 31–41. ACM Press, 2004.

[23] Sourceforge. Cpu usage limiter for linux. `http://cpulimit.sourceforge.net/`, 2009.

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

[25] M. V. Tripunitara and B. Carbunar. Efficient access enforcement in distributed role-based access control (RBAC) deployments. Technical report, ECE Department, University of Waterloo, 2009. Available from `http://ece.uwaterloo.ca/~tripunit/papers/TC09a.pdf`.

[26] Q. Wei, J. Crampton, K. Beznosov, and M. Ripeanu. Authorization recycling in rbac systems. In *Proceedings of the 13th ACM Symposium on Access Control, Models and Technologies (SACMAT'08)*, pages 63–72, 2008.