Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Constructing cascade bloom filters for efficient access enforcement

Nima Mousavi[a,1,*], Mahesh Tripunitara[b]

[a] Google, Canada
[b] ECE, University of Waterloo, Waterloo, Canada

ARTICLE INFO

Article history:
Received 9 October 2017
Revised 2 August 2018
Accepted 5 September 2018
Available online 24 October 2018

ABSTRACT

We address access enforcement — the process of determining whether a request for access to a resource by a principal should be granted. While access enforcement is essential to security, it must not unduly impact performance. Consequently, we address the issue of time- and space-efficient access enforcement, and in particular, study a particular data structure, the Cascade Bloom filter, in this context. The Cascade Bloom filter is a generalization of the well-known Bloom filter, which is used for time- and space-efficient membership-checking in a set, while allowing for a non-zero probability of false positives. We consider the problems, in practice, of constructing Bloom, and Cascade Bloom filters, with our particular application, access enforcement, in mind. We identify the computational complexity of the underlying problems, and propose concrete algorithms to construct instances of the data structures. We have implemented our algorithms, and conducted empirical assessments, which also we discuss in this paper. Our code is available for public download. As such, our work is a contribution to efficient access enforcement.

1. Introduction

Access control, which is recognized as a fundamental component of security, deals with whether a user may carry out an action on a resource. For example, Alice may be allowed to read a file, but not write to it. When a user attempts access, a process called *access enforcement* mediates her attempt, and determines whether it succeeds. The basis on which access enforcement makes it decision is an authorization policy, which enunciates who has access to what.

The efficiency with which access enforcement is carried out is of crucial importance to a system, with regards to balancing security and performance. We want access enforcement to be time- and space-efficient, while allowing only accesses that are authorized by the policy. Consequently, data structures and algorithms that underlie access enforcement are an important topic of study in information security.

In this work, our focus is a particular data structure, called the Cascade Bloom filter (Komlenovic et al., 2011; Tripunitara and Carbunar, 2009), that has been proposed in prior work for time- and space-efficient access enforcement. The Cascade Bloom filter is a generalization of a Bloom filter (Bloom, 1970). We describe both data structures and the associated algorithms in more detail in Sections 3 and 4, and provide an introductory description here.

The Bloom filter is a data structure for checking, in constant-time, whether an item is a member of a set. Apart from the classical time-space trade-off, the Bloom filter introduces a third axis: a probability of error when a 'yes' answer

* Corresponding author.
E-mail addresses: nmousavi@google.com (N. Mousavi), tripunit@uwaterloo.ca (M. Tripunitara).
[1] Nima Mousavi's contributions to this work were made when he was a Ph.D. student in the ECE department at the University of Waterloo.

https://doi.org/10.1016/j.cose.2018.09.015
0167-4048/© 2018 Elsevier Ltd. All rights reserved.

is returned for a membership check. That is, it is possible, for example, to get high time- and space-efficiency in our checks, provided we are willing to suffer a high probability of error in the checks. Or, we may suffer higher space-usage, while retaining high time-efficiency, and a low probability of error.

The Cascade Bloom filter is designed particularly with the application in mind: access enforcement. In access enforcement, we always want a correct ‘yes’ or ‘no’ answer as to whether a user is authorized. That is, we cannot tolerate any error. The Cascade Bloom filter adapts the classical Bloom filter in a creative way to achieve this. Specifically, we encode, as another Bloom filter, the set that results in erroneous checks (see Section 4).

Our contributions As we mention above, prior work (Komlenovic et al., 2011; Tripunitara and Carbunar, 2009) has proposed the Cascade Bloom filter for access enforcement, and presented empirical evidence for its effectiveness in practice, particularly for Role-Based Access Control (RBAC), in distributed settings. The focus of this work is the creation of instances of the data structure. We first carefully identify the various inputs that are meaningful when we seek to create an instance of a Cascade Bloom filter. We then articulate the problems that underlie the creation of such an instance.

We study these problems, and find that creating a Cascade Bloom filter is **NP**-hard in general (see Sections 3 and 4). Indeed, it remains **NP**-hard even if we make assumptions about the input that may be meaningful in practice for our application, access enforcement. While this may be seen as a discouraging result, we observe that a corresponding decision problem is in **NP**, and devise a suite of algorithms for constructing Cascade Bloom filters based on reduction to Boolean Satisfiability, SAT Cormen et al. (2009), with the intent of employing a corresponding constraint-solver. We have implemented our algorithms and our code is available for public download (Mousavi, 2018). In this paper, we present an empirical evaluation based on our implementation. In our empirical evaluation, we identify the manner in which our performance is impacted by various choices of inputs.

Organization The remainder of the paper is organized as follows. In Section 2, we discuss access enforcement. In Section 3, we describe the Bloom filter, its use for access enforcement, and consider the problem of constructing a Bloom filter, including the computational complexity of the underlying problems. In Section 4, we describe the Cascade Bloom filter, and consider the problem of constructing instances of that data structure. In Section 5, we discuss our implementations, and present the results of an empirical evaluation we have conducted. We discuss related work in Section 6, and conclude with Section 7.

2. Access enforcement

Access enforcement is the process by which we decide ‘yes’ or ‘no’ to a request for access to a resource from a user, or her agent. The decision is based on an authorization policy, which enunciates who has access to what. In the literature in access control, several different syntaxes have been proposed for expressing an authorization policy. Two well-known examples are the access matrix (Harrison et al., 1976), and RBAC (Sandhu et al., 1996). We focus on the latter in this work; however, our contributions on the Cascade Bloom filter apply to the other syntaxes as well, and indeed to other situations in which efficient membership checking is needed.

In RBAC users acquire permissions via roles. In Fig. 1, we show an RBAC policy with three users, four roles and five permissions. A user creates a session to exercise a set of permissions. In the session, the user activates a subset of the roles to which she is authorized. In Fig. 1, we show two sessions, s_a and s_b, and with bold line segments, the user that created them, and the roles that have been activated for those sessions. The session s_a has been created by Alice, and in that session, she has chosen to activate the Project Manager role only. The session s_b has been created by Bob in which he has activated the role, IT Consultant. The permissions to which those roles are authorized are then available for exercise to those sessions. For example, the session s_a is authorized to the permission Team Organization only.

When the user seeks to exercise a permission in the context of a session, the enforcement mechanism needs to check whether there exists an active role associated with the session that is authorized to that permission. If such a role exists, the request by the user is allowed. This is exactly the process of access enforcement in the context of RBAC. For example, if Alice, in the context of the s_a session, seeks to exercise the Team Organization permission, she is allowed to do so. If she seeks to exercise the Project Planning permission in the context of the session s_a, she is disallowed. Of course, she can create a session in which she activates the Software Engineer role, wherein she can exercise the Project Planning permission.

From the standpoint of access enforcement, an access request for a permission p in a session s can be represented by a pair $\langle s, p\rangle$. Let S and P denote the set of active sessions, and the set of all permissions authorized to any session, respectively. The set of valid requests $VRQ = \{\langle s, p\rangle : s \text{ is authorized to } p\}$ is a subset of $S \times P$. Thus, the problem of access enforcement can be perceived as membership-checking in the set VRQ. In the example in Fig. 1, the set of sessions, $S = \{s_a, s_b\}$, the set of all authorized permissions, $P = \{\text{Team Organization}, \text{Project Review}\}$. And the set of valid requests, $VRQ = \{\langle s_a, \text{Team Organization}\rangle, \langle s_b, \text{Project Review}\rangle\}$.

3. Bloom filter

A Bloom filter is a data structure proposed by Bloom (1970) for applications where the amount of memory required to store a set with any error-free methods is impractically large. Since it was first proposed, as testimony to its enduring utility, there has been considerable work on the Bloom filter; in particular, its utility has been recognized in various contexts (see, for example, Alzahrani et al., 2018; Klonowski and Piotrowska, 2018; Rathgeb and Busch, 2014; Roh et al., 2013; Xu et al., 2017; Yu et al., 2017).

From our standpoint of access enforcement, we perceive a Bloom filter as a space efficient data structure that is used

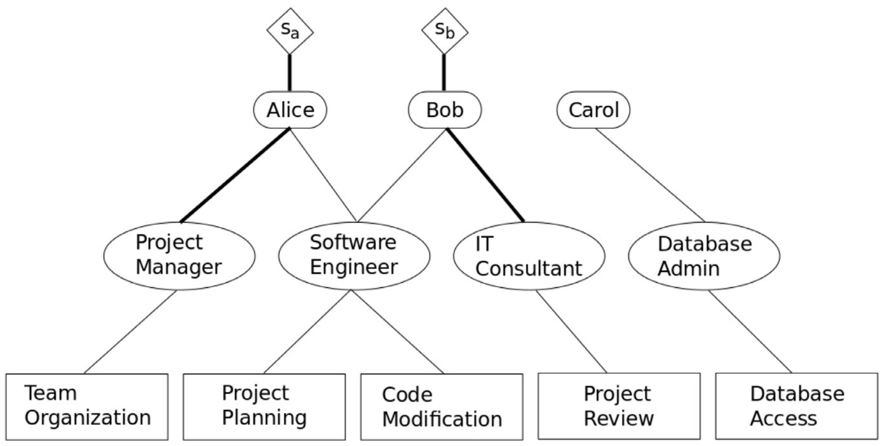

Fig. 1 – An example RBAC policy, and sessions, s_a and s_b. Bold line-segments indicate the roles that have been activated in each session.

Session	Permission	h_1	h_2
s_a	Team Organization	3	1
s_a	Project Review	3	3
s_b	Team Organization	0	2
s_b	Project Review	0	3

index →	0	1	2	3	4
	1	1	0	1	0

Fig. 2 – An example of encoding the set of authorized session-permission pairs as a Bloom filter.

to represent a subset A of elements in a universe $\mathcal{U}$. We say that the Bloom filter *represents* A *against* $\mathcal{U} - A$; the Bloom filter supports membership queries, that is, whether an element of $\mathcal{U}$ is in A or not. A Bloom filter is an array M of m bits associated with a set H of hash functions $h : \mathcal{U} \to \{0, 1, \ldots, m-1\}$, and is represented by a tuple $\langle M, H\rangle$. All bits in the array are initially set to zero. To add an element $a \in A$ into the Bloom filter, the indices $h(a)$ for all $h \in H$ are computed and the corresponding bits in the array are set to one. To query for an element a, the indices $h(a)$ for all $h \in H$ are computed, and if any of the indices for a is not set to one, the element is surely not in A. If all are one, then the element is reported to be in A. It is possible that some elements not in A pass the membership query by coincidence. Such elements are called *false positives*.

Figure 2 shows an example of encoding the set of valid requests, each of which is a pair $\langle s, p\rangle$, where s is a session and p is a permission, as a Bloom filter. We assume that we use two hash functions, h_1 and h_2, each of which maps a session-permission pair to one of $\{0, \ldots, 4\}$. The tables to the left show the manner in which the two hash functions map each pair. We then show the Bloom filter itself — index i, for $i \in \{0, \ldots, 4\}$, is set if and only if at least one of the authorized session-permission pairs maps to i for one of the hash functions. The session-permission pair, $\langle s_a, \text{Project Review}\rangle$, which is not authorized, is a false positive, because the corresponding entries under both h_1 and h_2 are set.

Access enforcement cannot tolerate errors; we are required to definitively respond to a membership query with a correct 'yes' or 'no.' A natural way to mitigate the problem of false positives that can exist with a Bloom filter is to simply explicitly maintain those false positives in, for example, a list. (In Section 4, we propose the Cascade Bloom filter, which proposes a more elegant solution.) Thus, an element is in A encoded by the Bloom filter if and only if (i) it tests positive with the Bloom filter, and, (ii) it is not in the list E.

In the example in Fig. 2, the only entry in E would be $\langle s_a, \text{Project Review}\rangle$. As additional examples, when we membership-check for $\langle s_b, \text{Project Review}\rangle$, we observe that it tests positive with the Bloom filter, as both indices 0 (its image under h_1), and 3 (its image under h_2) are set, and it is $\notin E$. When we membership-check for $\langle s_b, \text{Team Organization}\rangle$, the Bloom filter does not test positive, as its image under h_2, which is 2, is not set.

Assuming that each hash function is a random function, i.e., maps every element of U uniformly to an index in $\{0, 1, \ldots, m-1\}$, a formula can be derived for the false positive rate, i.e., the probability that an element not in A is a false positive. The formula enables one to minimize the false positive rate for a given A and M, by choosing the following value for the number of uniform hash functions.

$$k = \frac{m}{n} \ln 2$$

Where, k, m, and n are the size of H, M and A respectively. However, the uniformity assumption is not always true in practice — it is often difficult to even ascertain whether a particular hash function is indeed a random function. Thus, we study the problem of finding an optimal Bloom filter for a given set of elements $A \subseteq \mathcal{U}$, and a set of available hash functions, $\mathcal{H}$; we do not make any assumption for the hash functions in $\mathcal{H}$, other than that it is indeed a function the maps $\mathcal{U}$ to $\{0, \ldots, m-1\}$.

We can associate two types of costs to any Bloom filter that represents a set A with a bit array M and hash functions in H: (1) the memory cost of storing all the false positives in the list E, and (2) the computational cost associated with each membership query, which corresponds to the number of hash functions in H. Therefore, our goal is to find Bloom filters with fewer false positives and hash functions. In the following, we define the BF problem that is the optimization problem of finding an optimal Bloom filter with respect to two objectives: the number of false positives, and the number of hash functions used.

BF Specification. An optimization the BF problem is specified by the following inputs:

- $\mathcal{U}$: A finite universe of elements
- A: A subset of $\mathcal{U}$
- M: An array of m bits
- $\mathcal{H}$: A set of hash functions that map each element of $\mathcal{U}$ to a non-negative integer
- *pri*: $pri \in \{n_{fp}, n_h\}$ indicates which of the two optimization objectives, the number of false positives or the number of hash functions, we prioritize over the other. Without this parameter, there can exist two optimal solutions that are incomparable to one another.

Assuming that we prioritize the number of false positives over the number of hash functions, i.e., $pri = n_{fp}$, a solution to the optimization the BF problem is a Bloom filter $\langle M, H\rangle$ with a minimum number of false positives such that the size of $H \subseteq \mathcal{H}$ is minimum over all the solutions with the minimum number of false positives. A decision version of BF that corresponds to the above optimization version does not take the input *pri*. Instead two input integers:

- k_{fp}: this indicates the maximum number of false positives that we seek in a solution
- k_h: this indicates the maximum number of hash functions that a solution can use

A decision instance is either true or false. It is true if there exists a Bloom filter $\langle M, H\rangle$ that represents A against $\mathcal{U}-A$ with at most k_{fp} false positives and k_h hash functions (i.e., $|H| \le k_h$).

From the decision version to the optimization version. There exists a polynomial-time Turing reduction (Cook, 1971) from the optimization versions of BF to the decision version. That is, given an oracle Ω for the decision version, we can solve the optimization version in polynomial time. An approach is performing a two-dimensional binary search for the number of false positives and the number of hash functions. For example, for the case that $pri = n_{fp}$, we first fix k_h at the total number of hash functions, $|\mathcal{H}|$; that is, we accept a solution with any number of hash functions. We then perform a binary search for the optimal number of false positives with $O(\log|\mathcal{U}-A|)$ invocations to Ω. Once we find the optimal number of false positives, opt_{fp}, we search for the optimal number of hash functions with $O(\log|\mathcal{H}|)$ invocations to Ω, while k_{fp} is set to opt_{fp}.

3.1. *Computational complexity of the BF problem*

In this section we discuss the computational complexity of the BF problem. The formal language for the corresponding decision problem is

$$\begin{aligned}\mathrm{BF} = \{\left\langle \mathcal{U}, A, M, \mathcal{H}, k_{fp}, k_h \right\rangle : \ & \text{there exists a Bloom filter } \langle M, H\rangle \\ & \text{that represents } A \text{ against } \mathcal{U}-A \\ & \text{such that the number of false} \\ & \text{positives is at most } k_{fp}, \text{ and the size} \\ & \text{of } H \subseteq \mathcal{H} \text{ is at most } k_h\}.\end{aligned}$$

The following theorem shows that an efficient algorithm for the BF problem is unlikely to exist:

Theorem 1. *The BF problem is* **NP**-hard.

Proof. We prove it by showing that SET-COVER $\le_p$ BF. In SET-COVER (Cormen et al., 2009), we are given as input a set $\mathcal{U}$, a set of subsets of $\mathcal{U}$, $\mathcal{F}$, and an integer k. The question is whether there exists a collection of k subsets of $\mathcal{U}$ from within $\mathcal{F}$ whose union is $\mathcal{U}$. Given an instance of SET-COVER, $\phi = \langle \mathcal{U}, \mathcal{F}, k\rangle$, we construct an instance ψ of BF such that ψ is true if and only if ϕ is true.

We construct $\psi = \left\langle \mathcal{U}', A, M, \mathcal{H}, k_{fp}, k_h \right\rangle$ as follows. The universe $\mathcal{U}'$ consists of an element x_i for each e_i in $\mathcal{U}$, as well as an additional element x_{n+1}, where n denotes the size of $\mathcal{U}$. The set A contains only the element x_{n+1}. That is, $\mathcal{U}' = \{x_1, x_2, \ldots, x_n, x_{n+1}\}$ and $A = \{x_{n+1}\}$. The set of hash functions, $\mathcal{H}$, consists of a hash function h_j for each subset S_j in F. The bit array, M, consists of two bits, and each hash function maps an element to either index zero or one. Each hash function h_j is defined as below:

$$h_j(x_i) = \begin{cases} 1 & \text{if } i \ne n+1 \text{ and } e_i \text{ is in } S_j \\ 0 & \text{otherwise} \end{cases}$$

We set k_{fp} to zero, and k_h to k.

Suppose that $S' \subseteq \mathcal{F}$ is a set cover of size k. The Bloom filter $\langle M, H\rangle$ where $H = \{h_j : S_j \in S'\}$ represents A with no false positives because each x_i in $\mathcal{U}'-A$ is mapped by at least one hash function in H to index one, which is not set in M. Conversely, assume that there exists a Bloom filter $\langle M, H\rangle$ that represents A with no false positives and $|H| \le k$. The bit zero is the only bit in M that is set to one. The fact that the number of false positives is zero implies that there exists a hash function in H for each x_i in $\mathcal{U}-A$ that maps x_i to index one. Therefore, the set $S' = \{S_j : h_j \in H\}$ is a set cover of size at most k. □

The reduction in the proof of Theorem 1 shows that the hardness of the BF problem is related to minimizing the second objective, i.e., the number of hash functions. However, the following theorem shows that optimizing the number of false positives is also **NP**-hard.

Theorem 2. *The BF problem is* **NP**-hard *even if* $k_h = |\mathcal{H}|$.

Proof. We prove it by showing that SET-COVER $\le_p$ BF. Given an instance of SET-COVER instance, $\phi = \langle \mathcal{U}, \mathcal{F}, k\rangle$, we construct an instance of BF, $\psi = \langle \mathcal{U}', A, M, \mathcal{H}, k_{fp}, k_h\rangle$ as follows. Let n and m denote the size of $\mathcal{U}$ and F in ϕ respectively. The universe

$\mathcal{U}'$ consists of $(n+2)m+1$ elements $x_1, x_2, \ldots, x_{(n+2)m+1}$ from which the first m elements, and the last element are in A, i.e., $A = \{x_1, x_2, \ldots, x_m, x_{(n+2)m+1}\}$. Bit array M is an array of $m+1$ bits. The set of hash functions, $\mathcal{H}$, consists of a hash function h_j for each set S_j in $\mathcal{F}$, where h_j is defined as below.

$$h_j(x_i) = \begin{cases} j+1 & \text{if } i \le m \\ i+1-m & \text{if } m+1 \le i \le 2m \\ 1 & \text{if } 2m+1 \le i \le (n+2)m \text{ and } e_k \text{ is in } S_j, \\ & \text{where } k = \lceil \frac{i-2m}{m} \rceil \\ 0 & \text{o.w.} \end{cases}$$

We set k_{fp} to k, and k_h to $|\mathcal{H}|$. We claim that the BF instance is true if and only if the SET-COVER instance is true.

Suppose that there exists a set cover S' of size at most k. Then we show that $\langle M, H\rangle$ where $H = \{h_j : S_j \in S'\}$ is a solution for ψ. After adding all elements of A using hash functions in H, the array M has $|H|+1$ bits set to one: bit zero since any hash function maps $x_{(n+2)m+1}$ to zero, and bit $j+1$ for any hash function h_j in H. Let I denote the indices in M that are set to one, i.e., $I = \{j+1 : h_j \in H\} \cup \{0\}$. Since S' is a set cover, any $x_i \in \mathcal{U}' - A$ for $i \ge 2m+1$ is mapped to one by at least one hash function in H; none of $x_{2m+1}, x_{2m+2}, \ldots, x_{(n+2)m}$ is false positive. Any of $x_{m+1}, x_{m+2}, \ldots, x_{2m}$ is false positive if and only if $i+1-m \in I$. Therefore, the number of false positives is at most k. Conversely, assume that $\langle M, H\rangle$ is a solution to ψ. We show that $S' = \{S_j : h_j \in H\}$ is a set cover. Assume toward a contradiction that there exists an element e_k that S' does not cover. So, all of $x_{(k+1)m+1}, \ldots, x_{(k+2)m}$ are false positives since any hash function in H maps them to zero, which is set to one. Now, we show that the size of H is at most k, so is the size of S'. Assume toward a contradiction that there are at least $k+1$ hash functions in H. The array M has at least $k+2$ bits set to one after adding all elements of A, because each hash function h_j in H sets two bits to one: bit zero, and bit $j+1$. Each nonzero bit of index $j \neq 0$ in M makes x_{m+j-1} to be a false positive. Therefore, the number of false positives is at least $k+1$, which contradicts the assumption that H is a solution to ψ. □

Theorem 1 gives a lower bound for the hardness of BF problem, i.e., BF $\in$ **NP**-hard. However, the following theorem establishes an upper bound for the hardness of the BF problem is in **NP**:

Theorem 3. *The BF problem is in* **NP**.

Proof. We prove that there exists a polynomial certificate for BF, which can be verified in polynomial time. A certificate for BF is a subset of size at most k_h of $\mathcal{H}$. An algorithm to verify the certificate first constructs a M by adding all elements of A using hash functions in H. Then, it checks for each element of $\mathcal{U} - A$ whether it is a false positive. The algorithm accepts the certificate if the number of false positives is at most k_{fp}. The verification algorithm runs in time $O(|\mathcal{U}||H|T_h)$ where T_h is the time complexity of computing each hash function. We assume T_h is polynomial in $|\mathcal{U}|$ and $|M|$, and therefore the verification algorithm is polynomial in the size of the instance. □

Theorem 3 suggests a way for mitigating the intractability of the Bloom Filter problem — efficient reduction to CNF-SAT. We discuss this approach in the next section.

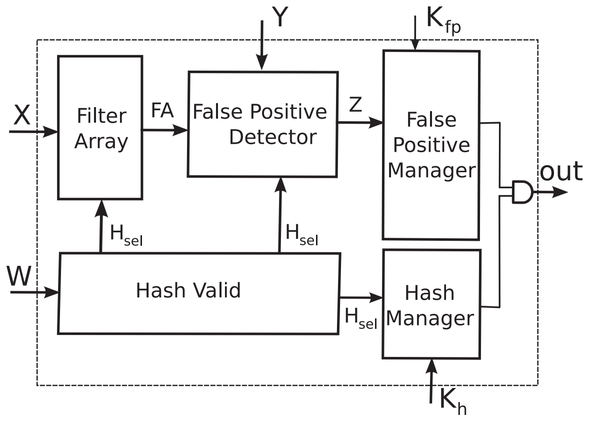

Fig. 3 – A circuit to decide the BF problem.

3.2. *Efficient reduction to CNF-SAT*

In this section, we discuss how the intractability of the BF problem can be mitigated. We investigate an approach in which we reduce the BF problem to CNF-SAT for which solvers exist that are efficient for large classes of instances. The fact that BF is in **NP** implies that there exists a polynomial-time many-one reduction from BF to CNF-SAT. We present an efficient reduction, which is based on designing a circuit that decides BF. Our reduction to CNF-SAT involves reducing BF first to a circuit SAT problem, and then reducing the circuit SAT to CNF-SAT.

Let n denote the size of the universe $\mathcal{U}$. We encode the set A with n binary variables $x_1, x_2, \ldots, x_n$ in our circuit, where circuit variable x_i is one if and only if element x_i is in A. We say $X = \{x_1, x_2, \ldots, x_n\}$ encodes A. Similarly, we encode $\mathcal{U} - A$ with n variables $y_1, y_2, \ldots, y_n \in Y$. The set of circuit variables $K = \{k_1, k_2, \ldots, k_m\}$ is a binary encoding for k if $m = \lceil \log k \rceil$, and $\sum_i k_i 2^i = k$.

The circuit has five inputs: X, which is the encoding of A; Y, the encoding of $\mathcal{U} - A$; W, an encoding of the size of M; K_h, the binary encoding of k_h; and K_{fp}, the binary encoding of k_{fp}. The output is one if and only if there exists a solution to the corresponding BF instance. As shown in Fig. 3, the circuit consists of five components, which we explain in the following:

Hash Valid. A hash function in $\mathcal{H}$ may map an element of $\mathcal{U}$ to an index greater than $|M|$. Hash Valid is a module that determines if a hash functions is valid to be used in the Bloom filter. A Hash Valid module is shown in Fig. 4. The first input to Hash Valid is W, which is the binary encoding of the size of the bit array M. Hash Valid uses a compare module (Mousavi and Tripunitara, 2012; Sinz, 2005) to decide whether a hash function is valid; a hash function is valid if the maximum to which it maps an element of the universe is less than m. For each hash function h_j, we have a circuit variable $h_{j,\text{val}}$ that is one if and only if h_j maps every element of the universe to an integer in $\{0, 1, \ldots, m-1\}$. Each variable $h_{j,\text{val}}$ is an input to the AND gate whose output $h_{j,\text{sel}}$ determines whether the hash function h_j is selected. $H_{sel} = \{h_{1,sel}, h_{2,sel}, \ldots, h_{|\mathcal{H}|,sel}\}$ is the output of Hash Valid, which is also an input to Filter Array and False Positives modules. Hash Valid consists of

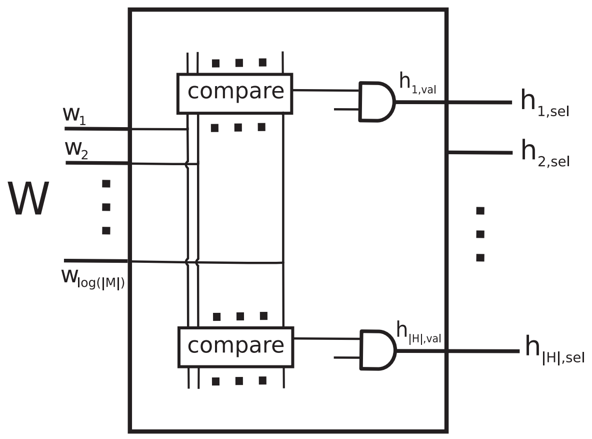

Fig. 4 – The Hash Valid module.

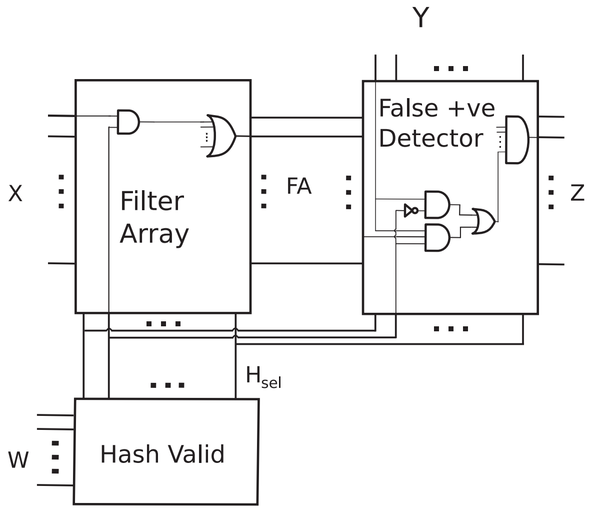

Fig. 5 – Filter array, false positives, and hash valid modules.

$|\mathcal{H}|$ compare modules and $|\mathcal{H}|$ AND gates. Thus, the total number of gates in Hash Valid is $O(|\mathcal{H}|\log|M|)$.

Filter Array. The first input to Filter Array is X, the encoding of A. The second input to the Filter Array is H_{sel}, Hash Valid's output. The output of Filter Array is the set of circuit variables $FA = \{f_1, f_2, \ldots, f_{|M|}\}$ that encodes the bits in the array M. We have the following relation between each f_k and input variables:

$$f_k = \bigvee_{(x_i, h_{j,sel}) \in D} (x_i \wedge h_{j,sel}) \qquad \text{where } D = \left\{(x_i, h_{j,sel}) : h_j(x_i) = k\right\}$$

Filter Array has $O(|\mathcal{U}||\mathcal{H}|)$ gates.

False positive detector. Three inputs to a False Positives Detector module are Y, the encoding of $\mathcal{U} - A$, H_{sel}, and FA. The output is $Z = \{z_1, \ldots, z_{|\mathcal{U}|}\}$ that encodes the set of false positives. A circuit variable z_k is one if and only if element x_k is a false positive. An element $x_k \in \mathcal{U} - A$ passes a hash function h_j if either h_j is not selected or the bit to which x_k is mapped by h_j is set to one. An element x_k is a false positive if it is in $\mathcal{U} - A$ and it passes every hash function. We have the following relation between each circuit variable z_k and input variables:

$$z_k = \bigwedge_j (y_k \wedge (h_{j,sel} \wedge f_l)) \qquad \text{where } l = h_j(x_k)$$

False Positive Detector consists of $O(|\mathcal{U}||\mathcal{H}|)$ gates. False Positive Detector is shown in Fig. 5.

Hash Manager. Hash Manager checks whether the total number of hash functions selected is less than k_h. Two inputs to Hash Manager are H_{sel} and K_h, an binary encoding of integer k_h. The Hash Manager module consists of a Max-Circuit module (Mousavi and Tripunitara, 2012). The output is one if the total number of hash functions selected is less than the k_h. Hash Manager consists of $O(|\mathcal{H}|\log|\mathcal{H}|)$ gates.

False positive manager. False Positive Manager checks whether the total number of false positives is less than k_{fp} using a Max-circuit module (Mousavi and Tripunitara, 2012). False Positive Manager has two inputs: Z, the output of a False Positive Detector module; and K_{fp}, an binary encoding of k_{fp}. The total number of gates in False Positive Manager is $O(|\mathcal{U}|\log(|\mathcal{U}|)$.

We adopt a "textbook" reduction from CIRCUIT-SAT to CNF-SAT Cormen et al. (2009) to generate the CNF formula that corresponds to the circuit. The total number of the gates in the circuit that decides an instance of BF is $O(n^2)$ where n is the size of the instance. Thus, the CNF-SAT formula is of size $O(n^2)$. This proves that the proposed reduction to CNF-SAT is efficient.

4. Cascade Bloom filter

In this section, we discuss the cascade Bloom filter, a generalization of the Bloom filter, which is proposed by Tripunitara and Carbunar (2009). A Cascade Bloom filter represents a set A against $\mathcal{U} - A$ by employing a cascade of Bloom filters. The basic idea of cascading multiple Bloom filters is to use the next Bloom filter to distinguish between two sets that the previous Bloom filter failed to, i.e., the false positives of the previous Bloom filter and the set that the previous Bloom filter represents.

As an example, we return to Fig. 2 in Section 3. Rather than maintaining a list of false positives for the Bloom filter, we maintain another Bloom filter, which encodes, as its set, the set of false positives from the first Bloom filter. In our example, the only false positive is $\langle s_a$, Project Review$\rangle$. Suppose we employ a third hash function h_3 whose range is {0, 1, 2}. And suppose we have the following mappings, and the corresponding Bloom filter which encodes the singleton set that contains $\langle s_a$, Project Review$\rangle$.

Session	Permission	h_3
s_a	Team organization	0
s_a	Project review	1
s_b	Project review	0

index →	0	1	2
	0	1	0

The only session-permission pairs for which the Bloom filter in Fig. 2 returns positive are the three shown in the table above. And the only false positive is $\langle s_a, \text{Project Review}\rangle$. Therefore, the second-level Bloom filter, for which we use the hash function h_3, has the bit at index 1 as the only one that is set. Thus, if we check with the first Bloom filter, and then this second, for a pair that tests positive with the first, we learn immediately that $\langle s_a, \text{Project Review}\rangle$ is a false positive, while neither of the others is.

More generally, a cascade Bloom filter is specified with d Bloom filters $\mathcal{BF}_1, \mathcal{BF}_2, \ldots, \mathcal{BF}_d$, where Bloom filter $\mathcal{BF}_1$ represents A against $\mathcal{U} - A$, and Bloom filter $\mathcal{BF}_i$ represents FP_{i-1}, the false positives in the previous level, against A_i. Similar to the traditional Bloom filter, the false positives in the last Bloom filter, FP_d, are stored in an explicit list E in order to perform membership query with no error.

The idea of using multiple Bloom filters has been proposed (Chazelle et al., 2004; Cohen and Matias, 2003). The Cascade Bloom filter is an adapted version of the Bloomier filter (Chazelle et al., 2004) suited to the purpose of access checking. Prior work (Tripunitara and Carbunar, 2009) gives an example in which a cascade Bloom filter with only two levels outperforms a Bloom filter by 33% fewer false positives, while it uses the same number of hash functions and allocates the same amount of memory to filter arrays.

Definition 1 (Cascade Bloom Filter). Tripunitara and Carbunar (2009)

A cascade Bloom filter is $\langle \mathcal{B}, E\rangle$, where $\mathcal{B} = \mathcal{BF}_1, \mathcal{BF}_2, \ldots, \mathcal{BF}_d$ is a list of Bloom filters and $E \subseteq \mathcal{U}$ is a set of elements from a universe $\mathcal{U}$. Each $l = 1, \ldots, d$ is called a level and d is called the depth of the cascade. Each $\mathcal{BF}_i = \langle M_i, H_i\rangle$ represents a set $A_i \subseteq \mathcal{U}$ against a set $B_i \subseteq \mathcal{U}$, such that for $i = 2, \ldots, d$, A_i is the set of false positives in $\mathcal{BF}_{i-1}$ and B_i is equal to A_{i-1}, with $A_1 = A$ and $B_1 = \mathcal{U} - A$. The set E is the set of false positives in BF_d. We say that the cascade Bloom filter represents A against $\mathcal{U} - A$.

The *total memory size* of a cascade Bloom filter is the sum of the array size at each level, i.e., $\Sigma_i m_i$. The *total number of hash functions used* in a cascade Bloom filter is the sum of the number of hash functions used at each level, i.e., $\Sigma_i |H_i|$. Before defining the problem of finding an optimal cascade Bloom filter, we discuss two examples. The first example shows how a cascade Bloom filter can reduce the number of false positives when a traditional Bloom filter can not. It also explain why the filter BF_i, which is to distinguish between FP_{i-1} and A_{i-1}, should represent FP_{i-1}, not A_{i-1}.

Example 1. Assume we want to represent $A = \{x_1, x_2\}$ against $\mathcal{U} - A = \{x_3\}$. The set of available hash functions is $\mathcal{H} = \{h_1, h_2\}$ where h_1 and h_2 are binary hash functions. $h_1(x) = 1$ if and only if $x = x_1$, and $h_2(x) = 0$ if and only if $x = x_2$. The element x_3 is a false positive for any Bloom filter $\langle M, H\rangle$. However, we can have zero false positive using a cascade Bloom filter with three Bloom filters: $\mathcal{BF}_1 = \langle M_1, H_1\rangle$, where $|M_1| = 0$ and $H_1 = \emptyset$; $\mathcal{BF}_2 = \langle M_2, H_2\rangle$, $|M_2| = 2$ and $H_2 = \{h_1\}$; and $\mathcal{BF}_3 = \langle M_3, H_3\rangle$, $|M_3| = 2$ and $H_3 = \{h_2\}$.

The following example shows that the number of false positives may not increase monotonically as the number of levels increase in the cascade Bloom filter.

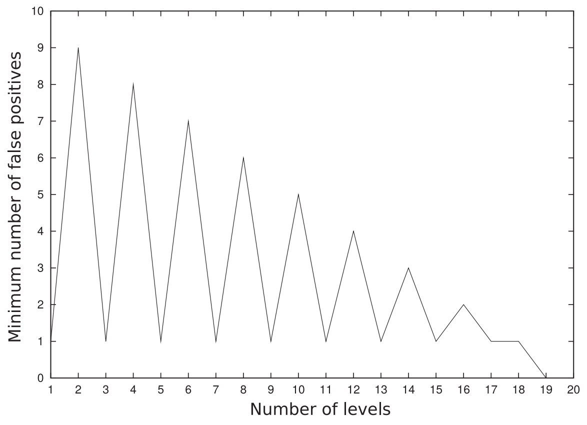

Fig. 6 – Example 2 above, with $n = 10$.

Example 2. Let $\mathcal{U} = \{x_1, x_2, \ldots, x_{n+1}\}$, $A = \{x_1, x_2, \ldots, x_n\}$. The set of available hash functions is $\mathcal{H} = \{h_1, h_2, \ldots, h_n\}$, where each hash function h_i is defined as below:

$$h_j(x_i) = \begin{cases} j \mod 2 & i = j \\ j-1 \mod 2 & \text{o.w.} \end{cases} \qquad (1)$$

If $Min_{fp}(d)$ denotes the minimum number of false positives that can be achieved with a cascade Bloom filter of depth d, we have the following:

$$Min_{fp}(d) = \begin{cases} 1 & \text{if } 0 < d < 2n-1 \text{ and } d \text{ is odd} \\ n-k & \text{if } 0 < d < 2n-1 \text{ and } d \text{ is even} \\ 0 & \text{if } d \geq 2n-1 \end{cases}$$

The minimum number of false positives can be achieved by choosing h_j for the Bloom filter at level $2j$ and any hash function for the Bloom filter at level $2j-1$ for $j = 1, \ldots, \lceil d/2 \rceil$. The size of the bit array at each level is 2. Fig. 6 shows $Min_{fp}(d)$ as d increases for $n = 10$.

Similar to the traditional Bloom filter, we define the problem of finding an optimal cascade Bloom filter of depth d with respect to two objectives: the number of false positives, and total number of hash functions used. It is a two dimensional optimization problem, which we define formally below.

CBF Specification. An optimization Cascade Bloom Filter problem (CBF) is specified by the following inputs:

- $\mathcal{U}$: A finite universe of elements
- A: A subset of $\mathcal{U}$
- d: A positive integer
- M: An array of m bits
- $\mathcal{H}$: A set of hash functions that map each element of $\mathcal{U}$ to a non-negative integer
- *pri*: $pri \in \{n_{fp}, n_h\}$ indicates which of the two optimization objectives, the number of false positives or the number of hash functions, we prioritize over the other.

A solution to the optimization version of the CBF problem is a cascade Bloom filter of depth d with total memory of m

that minimize the number of false positives and hash functions used. A decision version of the CBF problem that corresponds to the above optimization version takes all inputs of the optimization version, but the input *pri*, as well as two input integers:

- k_{fp}: this indicates the maximum number of false positives that we seek in a solution
- k_h: this indicate the maximum number of hash function that a solution can use

The decision and optimization versions of Cascade Bloom Filter problem are related closely. Given an oracle Ω for the decision version, we can solve the optimization version using two dimensional binary search approach.

4.1. *Computational complexity of the CBF problem*

The formal language for the corresponding decision version is

$$\text{CBF} = \{\langle \mathcal{U}, A, d, M, \mathcal{H}, k_{fp}, k_h \rangle : \text{there exists a cascade Bloom filter } \langle B, E\rangle \text{ of depth } d \text{ that represents } A \text{ against } \mathcal{U} - A \text{ such that the number of false positives is at most } k_{fp}, \text{ the total hash functions used is at most } k_h, \text{ and the total memory size is at most } |M|\}.$$

The BF problem is a special case of the CBF problem with $d = 1$. Since the Bloom Filter Problem is **NP**-hard, it is unlikely that there exists an efficient algorithm for the Cascade Bloom Filter.

Theorem 4. *The CBF problem is* **NP**-complete.

Proof. CBF is **NP**-hard because it generalizes the BF problem, which is proved to be **NP**-hard in Theorem 1.

We show that CBF $\in$ **NP**. A certificate for an instance of CBF is a list of d Bloom filters, $\mathcal{BF}_1, \mathcal{BF}_2, \ldots, \mathcal{BF}_d$. The size of certificate is $O(d(|\mathcal{H}| + |\mathcal{M}|))$, which is polynomial in the size of the instance since $d = O(|M|)$.

The verification algorithm first checks whether $\Sigma_i|H_i| \leq k_h$ and $\Sigma_i m_i \leq |M|$. It then computes the sets A_i and B_i for each $i = 1, \ldots, d$, and checks whether hash functions selected for each level are valid; that is, $h \in H_i$ maps each elements of $A_i \cup B_i$ to an integer less than m_i. Finally, it computes the set of false positives, and checks if its size is less than k_{fp}. The verification algorithm runs in time $O(d|\mathcal{U}||\mathcal{H}|T_h)$, where T_h is the time complexity of computing each hash function. We assume T_h is polynomial in $|\mathcal{U}|$ and $|M|$. Therefore, the verification can be performed in polynomial time. □

Theorem 4 establishes an upper bound for the complexity of the Cascade Bloom Filter problem.

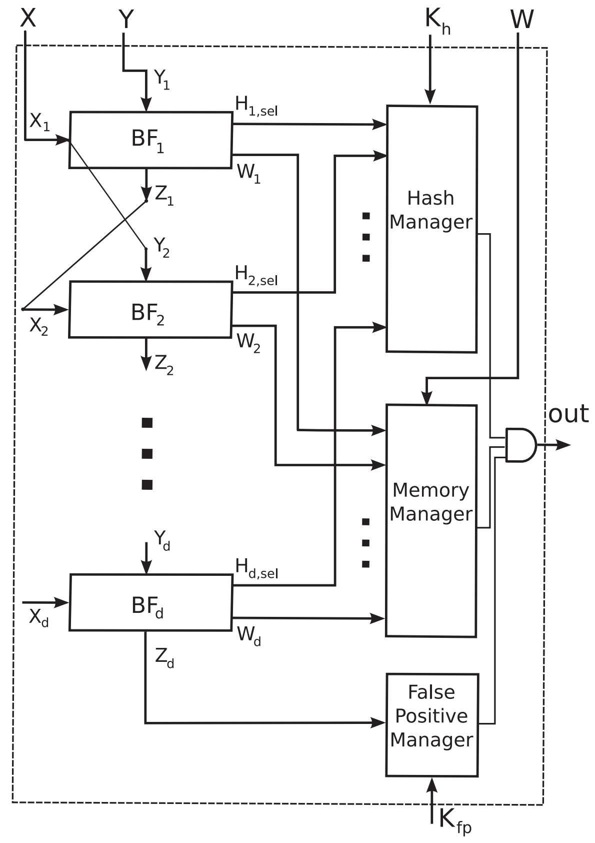

Fig. 7 – A circuit to decide the CBF problem.

4.2. *Efficient reduction to CNF-SAT*

In this section, we discuss how the intractability of the CBF problem can be mitigated. Our approach is to reduce an instance of the CBF problem to a CNF-SAT formula and solve the SAT formula using a SAT solver. Since CBF is in **NP**, there exists an efficient reduction from CBF to CNF-SAT. We find the efficient reduction by designing an efficient circuit that decides CBF and then adopting a “textbook” reduction from CIRCUIT-SAT to CNF-SAT (Cormen et al., 2009).

The circuit that decides CBF is shown in Fig. 7. It has five inputs: X, which is the encoding of A; Y, an encoding of $\mathcal{U} - A$; W, the encoding of $|M|$; K_h, the binary encoding of k_h; and K_{fp}, the binary encoding of k_{fp}. The output of the circuit is one if and only if there exists a solution to the corresponding CBF instance. The circuit consists of d modules of Bloom filters, $BF_1, BF_2, \ldots, BF_d$, a Hash Manager module, a Memory Manager module, and a False Positive Manager module. We explain each module in the following:

Bloom filter A Bloom Filter module is similar to the one described in the Section 3.2, except that it does not have False Positive Manager and Hash Manager modules (see Fig. 8). A Bloom Filter module has two inputs: X_i, the encoding of A_i; Y_i, the encoding of B_i, and three outputs: Z_i, the encoding of FP_i; $H_{i,\,sel}$, the encoding of the set of hash functions selected at level i; W_i, the binary encoding of m_i. A Bloom filter module

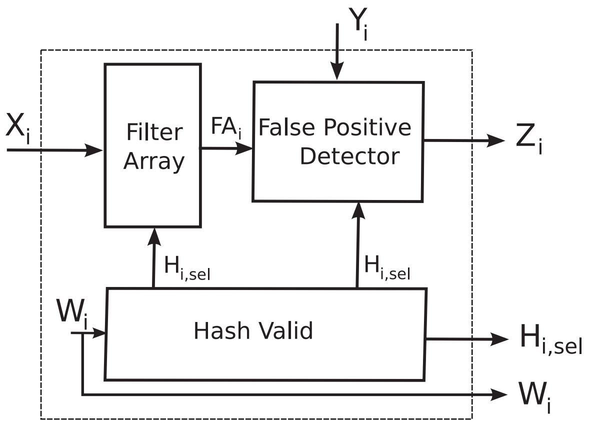

Fig. 8 – The Bloom filter module at level i, BF_i.

consists of three modules: a Filter Array, a False Positives Detector, and a Hash Valid, which are described in the Section 3.2.

Hash Manager Hash Manager checks if the total number of hash functions used is at most k_h. It consists of a Max-circuit with inputs $H_{1,sel}, H_{2,sel}, \ldots, H_{d,sel}$ and K_h. A Hash Manager module consists of $O(d|\mathcal{H}| \log d|\mathcal{H}|)$ gates.

Memory Manager Memory Manager checks if the total memory size is at most $|M|$. The inputs to a Memory Manager module are the binary encoding of memory size of each level, i.e., $W_1, W_2, \ldots, W_d$, and the binary encoding of the maximum memory size allowed, W. Memory Manager outputs one if and only if the total memory size is less than $|M|$. A Memory Manager module consists of a Max-circuit module (Mousavi and Tripunitara, 2012), and therefore has $O(d \log|M| \; (\log d + \log\log|M|))$ gates.

False Positive Manager False Positive Manager checks if the number of false positives is at most k_{fp}. The inputs to a False positive Manager module are Z_d, the encoding of the false positives at level d, and K_{fp}, the binary encoding of k_{fp}. False Positive Manager outputs one if and only if the number of false positives at most k_{fp}. A False Positives Manager module consists of a Max-circuit module Mousavi and Tripunitara (2012), and therefore has $O(|U|\log|U|)$ gates.

The total number of gates in the circuit for CBF is $O(dn^2)$, where n is the size of an instant of CBF. Since $d = O(|M|)$, the total size of the circuit, and therefore the size of the corresponding CNF SAT formula is $O(n^3)$.

5. Empirical evaluation

We have implemented our CNF SAT approach to the Cascade Bloom Filter problem. Our code is available for public download (Mousavi, 2018). We used MiniSat (2013), an open source SAT solver, to solve the SAT formula from the reduction in Section 4.2. The input for the empirical assessment has been generated based on the benchmark of RBAC instances in Komlenovic et al. (2011). Each generated RBAC consists of two users, $2n$ roles, $10n$ permissions. Each user is assigned to a role with probability 0.5. Each permission is connected to k random roles where k is less than 8. There is also a role-hierarchy of depth 3, generated according to stanford model (see Komlenovic et al., 2011). The generated RBAC policies are used to create CBF instances for our approach. For each RBAC instance with $2n$ roles, we create a session profile by instantiating n sessions. Each session profile comprises access pairs $\langle$*session id, permission*$\rangle$ for the permissions allowed in the sessions. In each session we choose the set of allowed permissions as below:

1. Randomly pick one of the two users, as the user for the session (call it u)
2. Randomly pick one role, r, from the roles to which u is authorized
3. Collect all the junior roles S for r (including r itself)
4. Collect all the permissions to which at least a role in S is authorized
5. Output that set of permissions for that session

The sets A and $\mathcal{U}$ in the corresponding CBF instance are the set of all access pairs (i.e., $\langle$*session id, permission*$\rangle$) in the session profile and the set of all possible access pairs respectively. Each generated CBF instance is represented with the quantity “Problem size”, which is the number of different sessions in the corresponding session profiles (i.e., n). All data points in our graphs represent a mean across at least 10 different inputs generated randomly.

Our empirical evaluations were conducted on a desktop PC with an Intel Xeon CPU E5-1650 v2 processor, that clocks between 1347 and 3900 MHz, and has a cache of 12 MB. The system has 32 GB RAM, and runs the 64-bit Debian 9 operating system.

Overall observations. Our approach results in cascade Bloom filters with significantly fewer false positives than prior work. We observe that a cascade Bloom filter with more levels produces fewer false positives, and in some cases requires fewer hash functions to achieve the same number of false positives. Furthermore, our implementation can be adapted to trade off between efficiency (CPU time) and the quality of the solution, i.e., the number of false positives and hash functions used in a solution.

We present our specific observations in the following sections:

5.1. *Comparison with a prior approach*

We were given access to the implementation of the prior work (Tripunitara and Carbunar, 2009). We first compared the performance of two approaches in optimizing the number of false positives when there is no constraint for the number of hash functions used. Fig. 9 shows the result of that comparison. We observe that our approach is resilient to the size of the problem, and is always able to find a cascade Bloom filter with a small number of false positives as the problem size increases.

To compare the performance of the two approaches in minimizing the number of hash functions, we set a constraint for the number of false positives to be no more than half of the

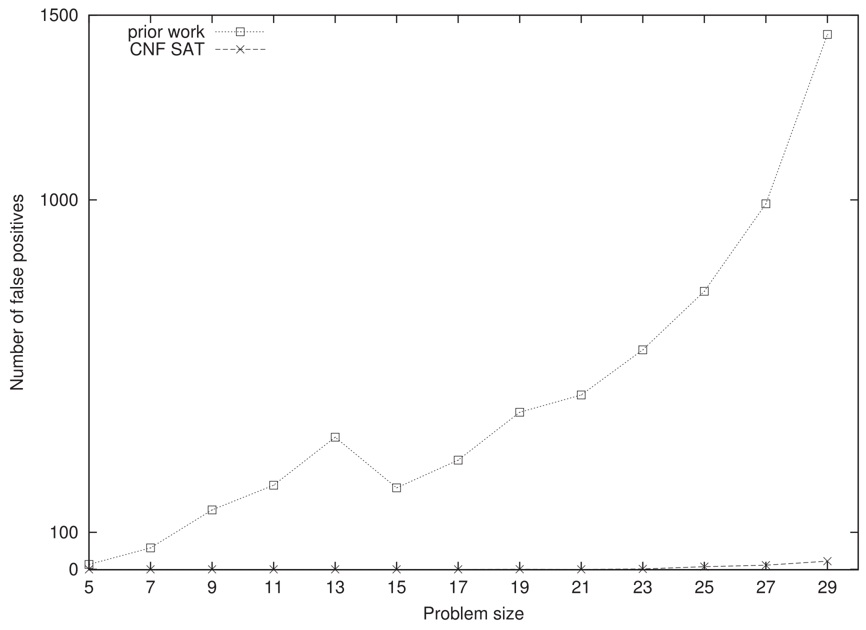

Fig. 9 – Number of false positives of our approach, labelled “CNF SAT”, and a prior approach.

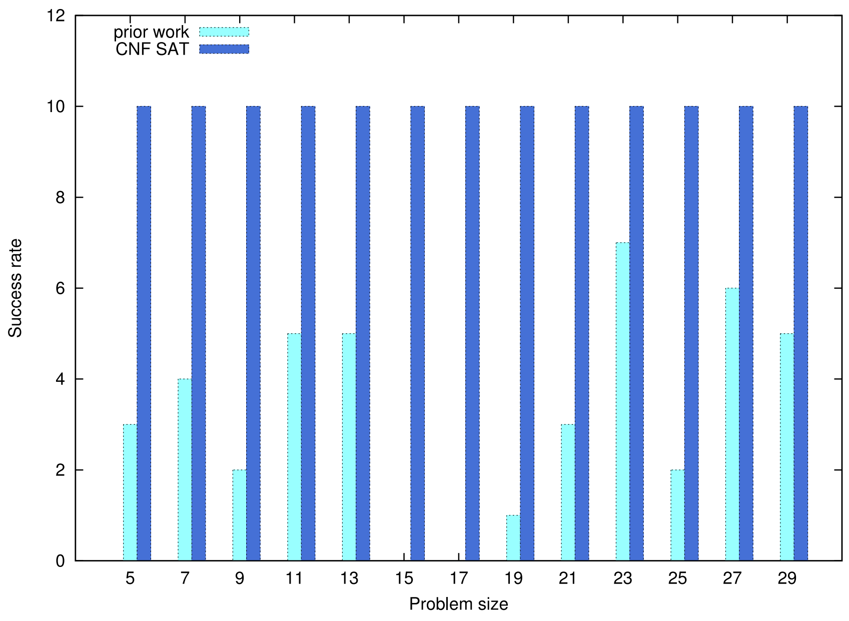

Fig. 10 – Success rate of our approach, labelled “CNF SAT”, and a prior approach.

maximum number of false positives, i.e., $|\mathcal{U} - A|/2$ where $\mathcal{U}$ is the universe and A is the set to be represented by the cascade Bloom filter. However, We were not be able to compare two approaches because the prior work fails for most instances. Fig. 10 shows the success rates for the two approaches.

5.2. *Efficiency of our approach*

We observe that the prior work always returns in few seconds, while it takes more time for our approach to find the optimal solution. However, our empirical results in the previous section show that the prior approach does not find an optimal solution, and it is not even complete; it may not return a solution for hard instances.

Fig. 11 shows the CPU time for our approach to find a cascade Bloom filter with minimum number of false positives as the problem size increases. Fig. 12 shows the CPU time for our approach to find a cascade Bloom filter with minimum number of hash functions. We observe that it takes few minutes for our approach to find the optimal cascade Bloom filter for large problem size. We are able to trade off between the CPU time and the quality of solution, i.e., the number of false positives

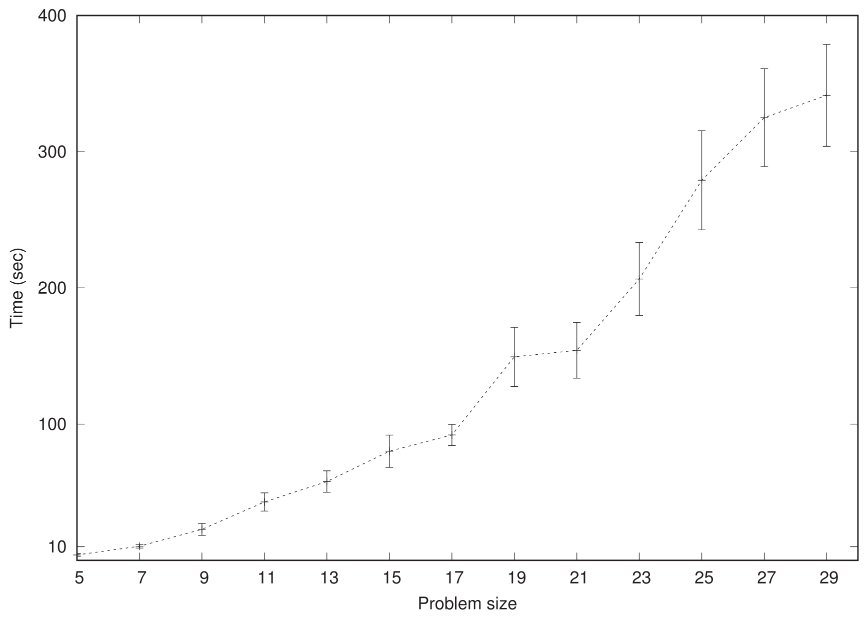

Fig. 11 – CPU time (average of 100 runs) for the minimum number of false positives for different problem sizes, with 95% confidence intervals.

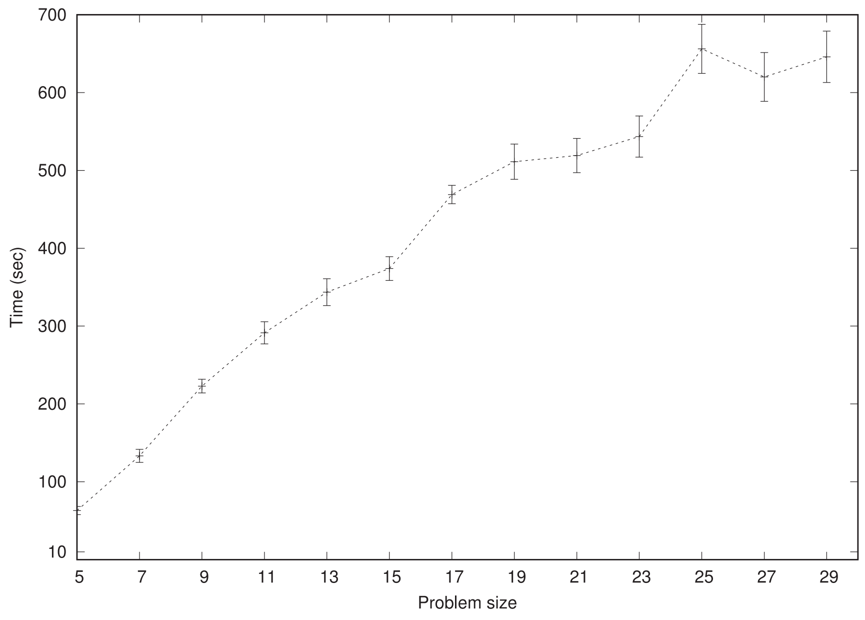

Fig. 12 – CPU time (average of 100 runs) with a minimum number of hash functions for different problem sizes, with 95% confidence intervals.

and the number of hash functions, by introducing a time limit for each invocation to the Sat solver, MiniSat, in our binary search for finding the optimal solution. If MiniSat does not return an answer within the time limit, the approach would deem that the instance provided to MiniSat is unsatisfiable. Indeed, the time limit affects the quality of the solution as there might be the case that the instance deemed unsatisfiable is just a hard satisfiable instance for MiniSat. Fig. 13 shows how the time limit affects the number of false positives in the solution that our approach returns.

5.3. *Effect of levels on the performance of the cascade Bloom filter*

Fig. 14 shows how the number of false positives changes as the number of levels increases. We observe that the number of false positives may increase first, but eventually decrease as the number of level increases. It is discussed in Example 2 and Fig. 6. Fig. 15 shows how the number of false positives changes for different problem sizes. We observe that regarding the number of false positives, cascade Bloom filter is more

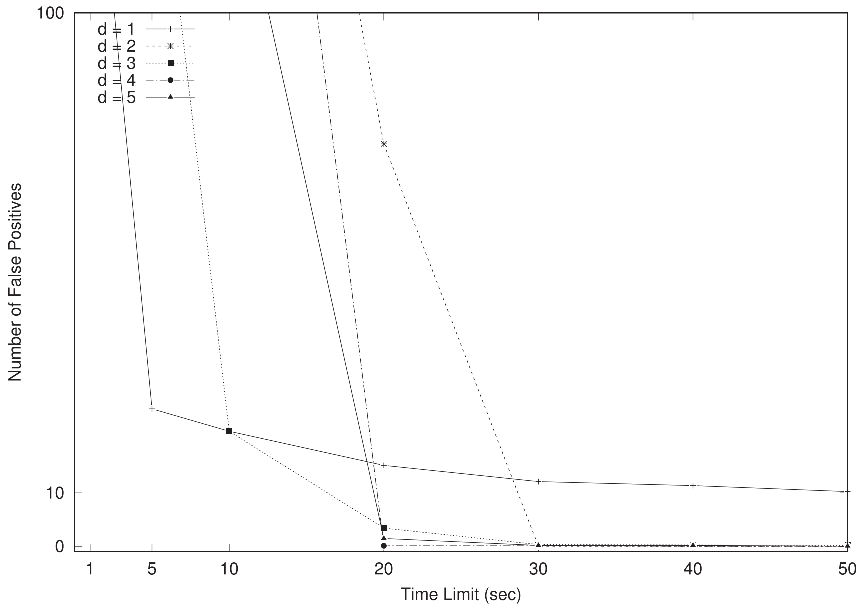

Fig. 13 – Achievable false positives for different number of levels, *d*, given a time-limit.

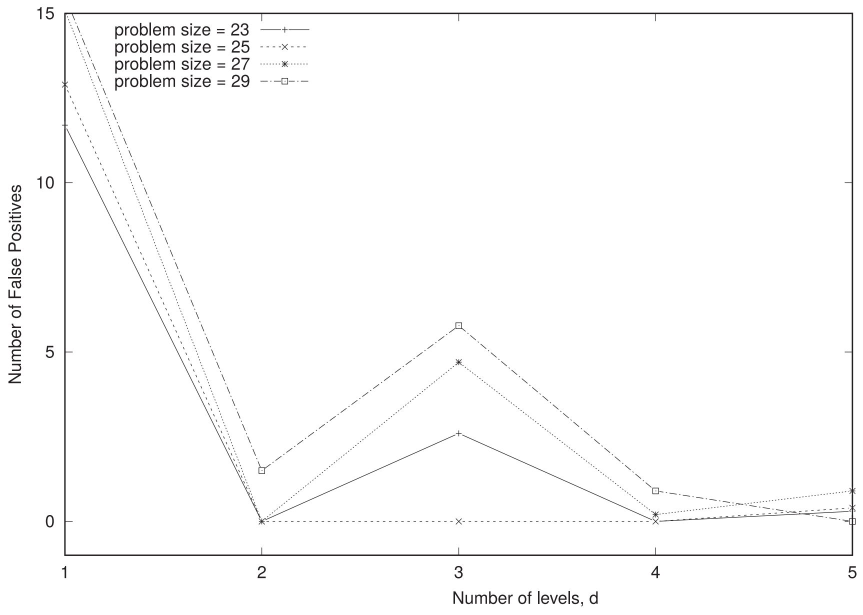

Fig. 14 – Number of levels vs. minimum number of false positives.

resilient to an increase in the problem size than the traditional Bloom filter. Fig. 16 shows how the number of hash functions required to achieve a *k* false positives decrease as *k* increases. We observe that for large value of *k*, a cascade Bloom filter with fewer levels uses fewer hash functions. However, for smaller value of *k*, a cascade Bloom filter with more levels uses fewer hash functions.

6. Related work

Our work pertains to efficient access enforcement, using a generalization of the Bloom filter. The Bloom filter was originally proposed by Bloom (1970). Since that work, there has been considerable work on the Bloom filter. A comprehensive discussion is well beyond the scope of this work; however, we point out, broadly, that more recent work on the Bloom filter can be categorized into two: variants of, or extensions to, the Bloom filter (see, for example, Chazelle et al., 2004; Cohen and Matias, 2003; Mitzenmacher, 2002; Sadhya and Singh, 2017; Yang and Chen, 2017), and new domains in which the Bloom filter or a variant has found applicability (see, for example, Alzahrani et al., 2018; Klonowski and Piotrowska, 2018; Rathgeb and Busch, 2014; Roh et al., 2013; Xu et al., 2017; Yu et al., 2017). None of such pieces of work are directly relevant to our work.

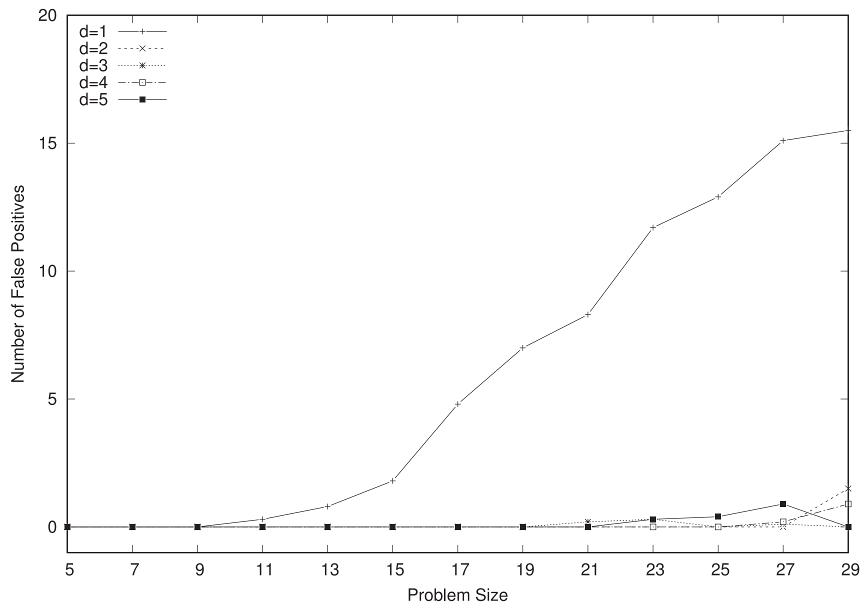

Fig. 15 – Problem size vs. minimum number of false positives, for different numbers of levels, *d*.

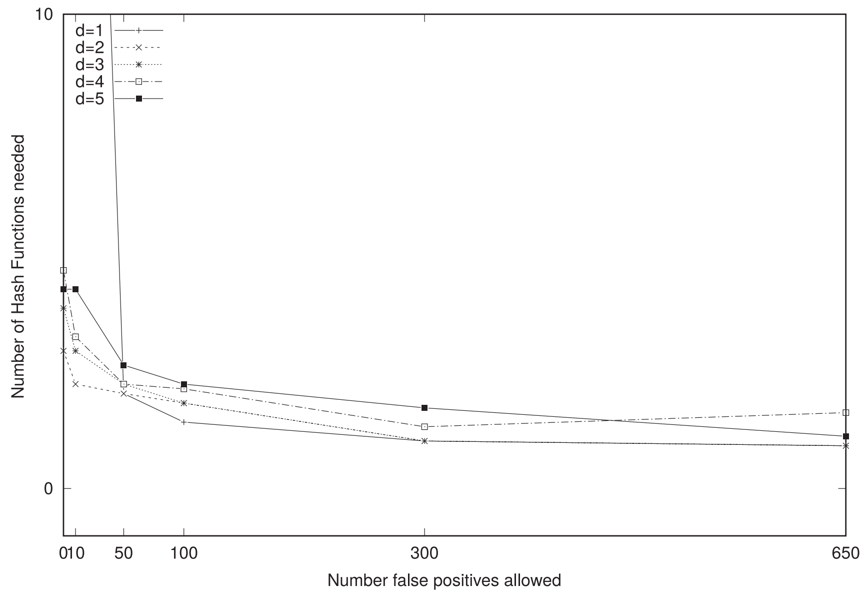

Fig. 16 – Number of of false positives allowed vs. minimum number of hash functions required.

To our knowledge, the Cascade Bloom filter was first proposed by Tripunitara and Carbunar (2009). That work considers the application that we consider as well, access enforcement for RBAC policies. However, while that work proposes an algorithm for constructing Cascade Bloom filters, it acknowledges that the algorithm there is not complete — the algorithm may fail even though a Cascade Bloom filter that corresponds to the inputs exists. That work also leaves the underlying questions regarding computational complexity, which are intimately tied to problem of devising an algorithm, unanswered.

From the standpoint of efficient access enforcement, there are indeed a few pieces of prior work. CPOL (Borders et al., 2005) is an approach to access enforcement in distributed settings. CPOL employs caching and a structure called an AccessToken, that is application-specific, to speed-up access enforcement. The work on CPOL points out also that simply using database querying does not suffice for fast access enforcement. Our work is related also to those of Wei et al. (2008), Komlenovic et al. (2011), and Liu et al. (2006), that address the access enforcement problem in RBAC. Wei et al. (2008) propose an architecture for distributed enforcement. In that context, they propose authorization recycling as an approach to speed up enforcement. Komlenovic et al. (2011) empirically study various prior approaches, including the Cascade Bloom filter, from the standpoint of efficient access enforcement. Liu et al. (2006) propose a technique that they call transformations for access checking in RBAC. All these pieces of work are complementary to ours, and do not address the issues we address.

7. Conclusions

We have addressed the construction of the Cascade Bloom filter, which prior work touts as effective for efficient access enforcement. We have identified the computational complexity of the underlying problems, and proposed concrete algorithms, which leverage off-the-shelf SAT solvers. We have implemented our algorithms, and carried out an empirical assessment. Our work suggests that instances of these data structures can indeed be constructed practically, and access enforcement based on them can perform well.

Access enforcement remains an interesting topic for further research, as do further investigations into the Cascade Bloom filter. New authorization schemes continue to be proposed, and it is worthwhile to ask how efficiently access enforcement can happen for those schemes. Indeed, our work considers RBAC only. Particularly in the context of distributed access enforcement, it may be meaningful to investigate whether a Cascade Bloom filter can be fragmented, so each piece can be employed at a different location for access enforcement. Of interest also is asking what parameters in the construction of a Cascade Bloom filter actually need to be assumed to be unbounded. Also, it is interesting to investigate whether applications other than access enforcement can benefit from the Cascade Bloom filter.

REFERENCES

Alzahrani B, Alreshoodi M, Vassilaki V, Almuhaimeed A, Alarfaj F. Toward secure packet delivery in future internet communications. In: Proceedings of the 2018 IEEE international conference on consumer electronics (ICCE); 2018. p. 1–2. Jan

Bloom B. Space/time trade-offs in hash coding with allowable errors. Commun ACM 1970;13(7):422–6.

Borders K, Zhao X, Prakash A. CPOL: high-performance policy evaluation. In: Proceedings of the 12th ACM conference on Computer and communications security (CCS05). ACM Press; 2005. p. 147–57.

Chazelle B, Kilian J, Rubinfield R, Tal A. 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). ACM Press; 2004. p. 30–9.

Cohen S, Matias Y. Spectral bloom filters. In: Proceedings of the 2003 ACM SIGMOD international conference on management of data (SIGMOD); 2003. p. 241–52.

Cook SA. The complexity of theorem-proving procedures. In: Proceedings of the third annual ACM symposium on Theory of computing, STOC '71. New York, NY, USA: ACM; 1971. p. 151–8.

Cormen TH, Leiserson CE, Rivest RL, Stein C. Introduction to algorithms. 3rd. The MIT Press; 2009. September

Harrison MA, Ruzzo WL, Ullman JD. Protection in operating systems. Commun ACM 1976;19(8).

Klonowski M, Piotrowska AM. Light-weight and secure aggregation protocols based on bloom filters. Comput Secur 2018;72:107–21.

Komlenovic M, Tripunitara M, Zitouni T. An empirical assessment of approaches to distributed enforcement in role-based access control (RBAC). In: Proceedings of the first ACM conference on data and application security and privacy, CODASPY '11. New York, NY, USA: ACM; 2011. p. 121–32.

Liu Y, Wang C, Gorbovitski M, Rothamel T, Cheng Y, Zhao Y, Zhang J. Core role-based access control: efficient implementations by transformations. In: Proceedings of the ACM SIGPLAN symposium on partial evaluation and semantics-based program manipulation; 2006. p. 112–20. May

MiniSat. 2013. January, http://minisat.se/.

Mitzenmacher M. Compressed bloom filters. IEEE/ACM Trans Netw 2002;10(5):604–12.

Mousavi N.. Code to construct cascade bloom filters. 2018. https://ece.uwaterloo.ca/~tripunit/cbf/, May.

Mousavi N, Tripunitara MV. Mitigating the intractability of the user authorization query problem in role-based access control (RBAC). In: Proceedings of the international conference on network and system security, NSS'12. Springer-Verlag; 2012. p. 516–29.

Rathgeb C, Busch C. Cancelable multi-biometrics: mixing iris-codes based on adaptive bloom filters. Comput Secur 2014;42:1–12.

Roh Bh, Kim JW, Ryu KY, Ryu JT. A whitelist-based countermeasure scheme using a bloom filter against sip flooding attacks. Comput Secur 2013;37:46–61.

Sadhya D, Singh SK. Providing robust security measures to bloom filter based biometric template protection schemes. Comput Secur 2017;67:59–72.

Sandhu RS, Coyne EJ, Feinstein HL, Youman CE. Role-based access control models. IEEE Comput 1996;29(2):38–47. February

Sinz C. Towards an optimal CNF encoding of Boolean cardinality constraints. In: van Beek P, editor. In: CP, volume 3709 of LNCS. Springer; 2005. p. 827–31.

Tripunitara M, Carbunar B. Efficient access enforcement in distributed role-based access control (RBAC) deployments. In: Proceedings of the 14th ACM symposium on access control models and technologies, SACMAT '09. New York, NY, USA: ACM; 2009. p. 23–32.

Wei Q, Crampton J, Beznosov K, Ripeanu M. Authorization recycling in RBAC systems. In: Proceedings of the 13th ACM symposium on access control models and technologies, SACMAT '08. New York, NY, USA: ACM; 2008. p. 63–72.

Xu Z, Chen B, Meng X, Liu L. Towards efficient detection of Sybil attacks in location-based social networks. In: Proceedings of the 2017 IEEE symposium series on computational intelligence (SSCI); 2017. p. 1–7. Nov

Yang Y, Chen S. Multiple bloom filters. In: Proceedings of the 2017 VI international conference on network, communication and computing, ICNCC, 2017. New York, NY, USA: ACM; 2017. p. 59–63.

Yu X., Chen X., Shi J., Shen L., Wang D.. Efficient and scalable privacy-preserving similar document detection. Proceedings of the IEEE global communications conference, GLOBECOM 2017–20172017:1–7Dec.

Nima Mousavi is a Software Engineer at Google, in Kitchener, Canada. He has a Ph.D. in Electrical and Computer Engineering from the University of Waterloo, where his Ph.D. dissertation explored algorithmic aspects of access control. He has an Master's from Sharifi University, Iran, and a Bachelor's from Iran University of Science and Technology, both in Electrical and Electronics Engineering.

Mahesh Tripunitara is Associate Professor in the Electrical and Computer Engineering Department at the University of Waterloo. He works mostly in information security, in problems in access control, the security of digital ICs, and applied cryptography. His work, with students, has been awarded "Best Paper" and "Best Paper, Runner-up" at the 2013 and 2015 ACM Symposium on Access Control Models and Technologies (SACMAT) respectively, and "Best Student Paper" at the 2013 Usenix Security Symposium.