Bloom Filters in Adversarial Environments

Moni Naor[1]* and Eylon Yogev[1]

Weizmann Institute of Science, Rehovot, Israel.**
{moni.naor,eylon.yogev}@weizmann.ac.il

Abstract. Many efficient data structures use randomness, allowing them to improve upon deterministic ones. Usually, their efficiency and/or correctness are analyzed using probabilistic tools under the assumption that the inputs and queries are *independent* of the internal randomness of the data structure. In this work, we consider data structures in a more robust model, which we call the *adversarial model.* Roughly speaking, this model allows an adversary to choose inputs and queries *adaptively* according to previous responses. Specifically, we consider a data structure known as "Bloom filter" and prove a tight connection between Bloom filters in this model and cryptography.
A Bloom filter represents a set S of elements approximately, by using fewer bits than a precise representation. The price for succinctness is allowing some errors: for any $x \in S$ it should always answer 'Yes', and for any $x \notin S$ it should answer 'Yes' only with small probability.
In the adversarial model, we consider both efficient adversaries (that run in polynomial time) and computationally unbounded adversaries that are only bounded in the amount of queries they can make. For computationally bounded adversaries, we show that non-trivial (memory-wise) Bloom filters exist if and only if one-way functions exist. For unbounded adversaries we show that there exists a Bloom filter for sets of size n and error ε, that is secure against t queries and uses only $O(n \log \frac{1}{\varepsilon} + t)$ bits of memory. In comparison, $n \log \frac{1}{\varepsilon}$ is the best possible under a non-adaptive adversary.

1 Introduction

Data structures are one of the most basic objects in Computer Science. They provide means to organize a large amount of data such that it can be queried efficiently. In general, constructing efficient data structures is key to designing efficient algorithms. Many efficient data structures use randomness, a resource that allows them to bypass lower bounds on deterministic ones. In these cases, their efficiency and/or correctness are analyzed in expectation or with high probability.

* Incumbent of the Judith Kleeman Professorial Chair.

** Supported in part by a grant from the I-CORE Program of the Planning and Budgeting Committee, the Israel Science Foundation, BSF and the Israeli Ministry of Science and Technology.

To analyze randomized data structures one must first define the underlying model of the analysis. Usually, the model assumes that the inputs and queries are *independent* of the internal randomness of the data structure. That is, the analysis is of the form: For any sequence of inputs, with high probability (or expectation) over its internal randomness, the data structure will yield a correct answer. This model is reasonable in a situation where the adversary picking the inputs gets no information about the randomness of the data structure (in particular, the adversary does not get the responses on previous inputs).

In this work, we consider data structures in a more robust model, which we call the *adversarial model.* Roughly speaking, this model allows an adversary to choose inputs and queries *adaptively* according to previous responses. That is, the analysis is of the form: With high probability over the internal randomness of the data structure, for any adversary adaptively choosing a sequence of inputs, the output of the data structure will be correct. Specifically, we consider a data structure known as "Bloom filter" and prove a tight connection between Bloom filters in this model and cryptography: We show that Bloom filters in an adversarial model exist if and only if one-way functions exist.

Bloom Filters in Adversarial Environments. The approximate set membership problem deals with succinct representations of a set S of elements from a large universe U, where the price for succinctness is allowing some errors. A data structure solving this problem is required to answer queries in the following manner: for any $x \in S$ it should always answer 'Yes', and for any $x \notin S$ it should answer 'Yes' only with small probability. The latter are called *false positive* errors.

The study of the approximate set membership problem began with Bloom's 1970 paper [4], introducing the so called "Bloom filter", which provided a simple and elegant solution to the problem. (The term "Bloom filter" may refer to Bloom's original construction, but we use it to denote any construction solving the problem.) The two major advantages of Bloom filters are: (i) they use significantly less memory (as opposed to storing S precisely) and (ii) they have very fast query time (even constant query time). Over the years, Bloom filters have been found to be extremely useful and practical in various areas. Some main examples are distributed systems [32], networking [10], databases [19], spam filtering [30], web caching [13], streaming algorithms [21, 9] and security [17, 31]. For a survey about Bloom filters and their applications see [6] and a more recent one [28].

Following Bloom's original construction many generalizations and variants have been proposed and extensively analyzed, proving better memory consumption and running time, see e.g. [8, 27, 24, 1]. However, as discussed, all known constructions of Bloom filters work under the assumption that the input query x is fixed, and then the probability of an error occurs over the randomness of the construction. Consider the case where the query results are made public. What happens if an adversary chooses the next query according to the responses of previous ones? Does the bound on the error probability still hold? The traditional analysis of Bloom filters is no longer sufficient, and stronger techniques are required.

Let us demonstrate this need with a concrete scenario. Consider a system where a Bloom filter representing a *white list* of email addresses is used to filter spam mail. When an email message is received, the sender's address is checked against the Bloom filter, and if the result is negative it is marked as spam. Addresses not on the white list have only a small probability of being a false positive and thus not marked as spam. In this case, the results of the queries are public, as an attacker might check whether his emails are marked as spam[1]. The attacker (after a sequence of queries) might be able to find a bulk of email addresses that are not marked as spam although they are not in the white list, and thus, bypass the security of the system and flood users with spam mail.

Alternatively, Bloom filters are often used for holding the contents of a cache. For instance, a web proxy holds on a (slow) disk, a cache of locally available webpages. To improve performance, it maintains in (fast) memory a Bloom filter representing all addresses in the cache. When a user queries for a webpage, the proxy first checks the Bloom filter to see if the page is available in the cache, and only then does it search for the webpage on the disk. A false positive is translated to a cache miss, that is, an unnecessary (slow) disk lookup. In the standard analysis, one would set the error to be small such that cache misses happen very rarely (e.g., one in a thousand requests). However, by timing the results of the proxy, an adversary might learn the responses of the Bloom filter, enabling her to cause a cache miss for almost every query and, eventually, causing a Denial of Service (DoS) attack.

Under the adversarial model, we construct Bloom filters that are resilient to the above attacks. We consider both efficient adversaries (that run in polynomial time) and computationally unbounded adversaries that are only bounded in the amount of queries they can make. We define a Bloom filter that maintains its error probability in this setting and say it is *adversarial resilient* (or just resilient for shorthand).

The security of an adversarial resilient Bloom filter is defined as a game with an adversary. The adversary is allowed to make a sequence of t adaptive queries to the Bloom filter and get their responses. Note that the adversary has only oracle access to the Bloom filter and cannot see its internal memory representation. Finally, the adversary must output an element x^* (that was not queried before) which she believes is a false positive. We say that a Bloom filter is (n, t, ε)-adversarial resilient if when initialized over sets of size n then after t queries the probability of x^* being a false positive is at most ε. If a Bloom filter is resilient for any polynomially many queries we say it is *strongly resilient*.

A simple construction of a strongly resilient Bloom filter (even against computationally unbounded adversaries) can be achieved by storing S precisely. Then, there are no false positives at all and no adversary can find one. The drawback of this solution is that it requires a large amount of memory, whereas

[1] For example, the attacker can spam his personal email account and see if the messages are being filtered.

Bloom filters aim to reduce the memory usage. We are interested in Bloom filters that use a small amount of memory but remain nevertheless, resilient.

1.1 Our Results

We introduce the notion of *adversarial-resilient Bloom filter* and show several possibility results (constructions of resilient Bloom filters) and impossibility results (attacks against any Bloom filter) in this context.

Our first result is that adversarial-resilient Bloom filters against computationally bounded adversaries that are non-trivial (i.e., they require less space than the amount of space it takes to store the elements explicitly) must use one-way functions. That is, we show that if one-way functions do not exist then any Bloom filter can be 'attacked' with high probability.

Theorem 1 (Informal). *Let* **B** *be a* non-trivial *Bloom filter. If* **B** *is strongly resilient against computationally bounded adversaries then one-way functions exist.*

Actually, we show a trade-off between the amount of memory used by the Bloom filter and the number of queries performed by the adversary. Carter et al. [7] proved a lower bound on the amount of memory required by a Bloom filter. To construct a Bloom filter for sets of size n and error rate ε one must use (roughly) $n \log \frac{1}{\varepsilon}$ bits of memory (and this is tight). Given a Bloom filter that uses m bits of memory we get a lower bound for its error rate ε and thus a lower bound for the (expected) number of false positives. As m is smaller the number of false positives is larger and we prove that adversary can perform fewer queries.

In the other direction, we show that using one-way functions one can construct a strongly resilient Bloom filter. Actually, we show that you can transform any Bloom filter to be strongly resilient with almost exactly the same memory requirements and at a cost of a single evaluation of a pseudorandom permutation (which can be constructed using one-way functions). Specifically, we show:

Theorem 2. *Let* **B** *be an* (n, ε)*-Bloom filter using* m *bits of memory. If pseudorandom permutations exist, then for large enough security parameter* λ *there exists an* $(n, \varepsilon + \mathsf{neg}(\lambda))$*-strongly resilient Bloom filter that uses* $m' = m + \lambda$ *bits of memory.*

Bloom filters consist of two algorithms: an initialization algorithm that gets a set and outputs a compressed representation of the set, and a membership query algorithm that gets a representation and an input. Usually, Bloom filters have a *randomized* initialization algorithm but a *deterministic* query algorithm that does not change the representation. We say that such Bloom filters have a "steady representation". We consider also Bloom filters with "unsteady representation" where the query algorithm is randomized and can change the underlying representation on each query. A randomized query algorithm may be more sophisticated and, for example, incorporate differentially private [12] algorithms in order to protect the internal memory from leaking. Differentially private algorithms are designed to protect a private database against adversarial and also adaptive queries from a data analyst. One might hope that such techniques can eliminate the need of one-way functions in order to construct resilient Bloom filters. However, we extend our results and show that they hold even for Bloom filter with unsteady representations, which proves that this approach cannot gain additional security.

In the context of unbounded adversaries, we show a positive result. For a set of size n and an error probability of ε most constructions use about $O(n \log \frac{1}{\varepsilon})$ bits of memory. We construct a resilient Bloom filter that does not use one-way functions, is resilient against t queries, uses $O(n \log \frac{1}{\varepsilon} + t)$ bits of memory, and has query time $O(\log \frac{1}{\varepsilon})$.

Theorem 3. *For any $n, t \in \mathbb{N}$, and $\varepsilon > 0$ there exists an (n, t, ε)-resilient Bloom filter (against unbounded adversaries) that uses $O(n \log \frac{1}{\varepsilon} + t)$ bits of memory.*

1.2 Related Work

One of the first works to consider an adaptive adversary that chooses queries based on the response of the data structure is by Lipton and Naughton [16], where adversaries that can measure the *time* of specific operations in a dictionary were addressed. They showed how such adversaries can be used to attack hash tables. Hash tables have some method for dealing with collisions. An adversary that can measure the time of an insert query, can determine whether there was a collision and might figure out the precise hash function used. She can then choose the next elements to insert accordingly, increasing the probability of a collision and hurting the overall performance.

Mironov et al. [18] considered the model of *sketching* in an adversarial environment. The model consists of several honest parties that are interested in computing a joint function in the presence of an adversary. The adversary chooses the inputs of the honest parties based on the common randomness shared among them. These inputs are provided to the parties in an on-line manner, and each party incrementally updates a compressed sketch of its input. The parties are not allowed to communicate, they do not share any secret information, and any public information they share is known to the adversary in advance. Then, the parties engage in a protocol in order to evaluate the function on their current inputs using only the compressed sketches. Mironov et al. construct explicit and efficient (optimal) protocols for two fundamental problems: testing equality of two data sets, and approximating the size of their symmetric difference.

In a more recent work, Hardt and Woodruff [14] considered linear sketch algorithms in a similar setting. They consider an adversary that can adaptively choose the inputs according to previous evaluations of the sketch. They ask whether linear sketches can be robust to adaptively chosen inputs. Their results are negative: They show that no linear sketch approximates the Euclidean norm of its input to within an arbitrary multiplicative approximation factor on a polynomial number of adaptively chosen inputs.

One may consider adversarial resilient Bloom filters in the framework of computational learning theory. The task of the adversary is to learn the private memory of the Bloom filter in the sense that it is able to predict on which elements the Bloom filter outputs a false positive. The connection between learning and cryptographic assumptions has been explored before (already in his 1984 paper introducing the PAC model Valiant's observed that the nascent pseudorandom random functions imply hardness of learning [29]). In particular Blum et al. [5] showed how to construct several cryptographic primitives (pseudorandom bit generators, one-way functions and private-key cryptosystems) based on certain assumptions on the difficulty of learning. The necessity of one-way functions for several cryptographic primitives has been shown in [15].

2 Model and Problem Definitions

Our model considers a universe U of elements, and a subset $S \subset U$. We denote the size of U by u, and the size of S by n. For the security parameter we use λ (sometimes we omit the explicit use of the security parameter and assume it is polynomial in n). We consider mostly the static problem, where the set is fixed throughout the lifetime of the data structure. We note that the lower bounds imply the same bounds for the dynamic case and the cryptographic upper bound (Theorem 4) can be adapted to the dynamic case.

A Bloom filter is a data structure that is composed of a setup algorithm and a query algorithm $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$. The setup algorithm $\mathbf{B}_1$ is randomized, gets as input a set S, and outputs a compressed representation of it $\mathbf{B}_1(S) = M$. To denote the representation M on a set S with random string r we write $\mathbf{B}_1(S; r) = M_r^S$ and its size in bits is denoted as $|M_r^S|$.

The query algorithm answers membership queries to S given the compressed representation M. Usually in the literature, the query algorithm is deterministic and cannot change the representation. In this case we say $\mathbf{B}$ has a *steady representation*. However, we also consider Bloom filters where their query algorithm is *randomized* and can change the representation M after each query. In this case we say that $\mathbf{B}$ has an *unsteady representation*. We define both variants.

Definition 1 (Steady-representation Bloom filter). *Let* $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ *be a pair of polynomial-time algorithms where* $\mathbf{B}_1$ *is a randomized algorithm that gets as input a set* S *and outputs a representation, and* $\mathbf{B}_2$ *is a deterministic algorithm that gets as input a representation and a query element* $x \in U$. *We say that* $\mathbf{B}$ *is an* (n, ε)*-Bloom filter (with a steady representation) if for any set* $S \subset U$ *of size* n *it holds that:*

1. *Completeness: For any* $x \in S$*:* $\Pr[\mathbf{B}_2(\mathbf{B}_1(S), x) = 1] = 1$
2. *Soundness: For any* $x \notin S$*:* $\Pr[\mathbf{B}_2(\mathbf{B}_1(S), x) = 1] \leq \varepsilon$,

where the probabilities are over the setup algorithm $\mathbf{B}_1$.

False Positive and Error Rate. Given a representation M of S, if $x \notin S$ and $\mathbf{B}_2(M, x) = 1$ we say that x is a *false positive*. Moreover, we say that ε is the *error rate* of $\mathbf{B}$.

Definition 1 considers only a single fixed input x and the probability is taken over the randomness of $\mathbf{B}$. We want to give a stronger soundness requirement that considers a sequence of inputs $x_1, x_2, \ldots, x_t$ that is not fixed but chosen by an adversary, where the adversary gets the responses of previous queries and can adaptively choose the next query accordingly. If the adversary's probability of finding a false positive x^* that was not queried before is bounded by ε, then we say that $\mathbf{B}$ is an (n, t, ε)-resilient Bloom filter (this notion is defined in the challenge $\mathsf{Challenge}_{A,t}$ which is described below). Note that in this case, the setup phase of the Bloom filter and the adversary get the security parameter 1^λ as an additional input (however, we usually omit it when clear from context). For a steady representation Bloom filter we define:

Definition 2 (Adversarial-resilient Bloom filter with a steady representation). *Let $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ be an (n, ε)-Bloom filter with a steady representation (see Definition 1). We say that $\mathbf{B}$ is an (n, t, ε)-adversarial resilient Bloom filter (with a steady representation) if for any set S of size n, for all sufficiently large $\lambda \in \mathbb{N}$ and for any probabilistic polynomial-time adversary A we have that the advantage of A in the following challenge is at most ε:*

1. *Adversarial Resilient:* $\Pr[\mathsf{Challenge}_{A,t}(\lambda) = 1] \le \varepsilon$,

where the probabilities are taken over the internal randomness of $\mathbf{B}_1$ and A and where the random variable $\mathsf{Challenge}_{A,t}(\lambda)$ is the outcome of the following game:

$\mathsf{Challenge}_{A,t}(\lambda)$*:*

1. $M \leftarrow \mathbf{B}_1(S, 1^\lambda)$.
2. $x^* \leftarrow A^{\mathbf{B}_2(M,\cdot)}(1^\lambda, S)$ *where A performs at most t queries $x_1, \ldots, x_t$ to the query oracle $\mathbf{B}_2(M, \cdot)$.*
3. *If $x^* \notin S \cup \{x_1, \ldots, x_t\}$ and $\mathbf{B}_2(M, x^*) = 1$ output 1, otherwise output 0.*

Unsteady representations. When the Bloom filter has an unsteady representation, then the algorithm $\mathbf{B}_2$ is randomized and moreover can change the representation M. That is, $\mathbf{B}_2$ is a query algorithm that outputs the response to the query as well as a new representation. Thus, the user or the adversary do not interact directly with the $\mathbf{B}_2(M, \cdot)$ but with an interface $Q(\cdot)$ (initialized with M) to a process that on query x updates its representation M and outputs only the response to the query (i.e. it cannot issue successive queries to the same memory representation but to one that keeps changing). Formally, $Q(\cdot)$ initialized with M on input x acts as follows:

The interface $Q(x)$ (initialized with M):

1. $(M', y) \leftarrow \mathbf{B}_2(M, x)$.

2. $M \leftarrow M'$.
3. Output y.

We define an analogue of the original Bloom filter for unsteady representations and then define an adversarial resilient one.

Definition 3 (Bloom filter with an unsteady representation). *Let $S \subset U$ be a set of size n. Let $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ be a pair of probabilistic polynomial-time algorithms such that $\mathbf{B}_1$ gets as input the set S and outputs a representation M_0, and $\mathbf{B}_2$ gets as input a representation and query x and outputs a new representation and a response to the query. Let $Q(\cdot)$ be the process initialized with M_0. We say that $\mathbf{B}$ is an (n, ε)-Bloom filter (with an unsteady representation) if for any such set S the following two conditions hold:*

1. *Completeness: After any sequence of queries $x_1, x_2, \ldots$ performed to $Q(\cdot)$ we have that for any $x \in S$: $\Pr[Q(x) = 1] = 1$.*
2. *Soundness: After any sequence of queries $x_1, x_2, \ldots$ performed to $Q(\cdot)$ we have that for any $x \notin S$: $\Pr[Q(x) = 1] \leq \varepsilon$,*

where the probabilities are taken over the internal randomness of $\mathbf{B}_1$ and $\mathbf{B}_2$.

Definition 4 (Adversarial-resilient Bloom filter with an unsteady representation). *Let $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ be an (n, ε)-Bloom filter with an unsteady representation (see Definition 3). We say that $\mathbf{B}$ is an (n, t, ε)-adversarial resilient Bloom filter (with an unsteady representation) if for any set $S \subset U$ of size n, for all sufficiently large $\lambda \in \mathbb{N}$ and for any probabilistic polynomial-time adversary A it holds that:*

1. *Adversarial Resilient: $\Pr[\mathsf{Challenge}_{A,t}(\lambda) = 1] \leq \varepsilon$,*

where the probabilities are taken over the internal randomness of $\mathbf{B}_1, \mathbf{B}_2$ and A and where the random variable $\mathsf{Challenge}_{A,t}(\lambda)$ is the outcome of the following process:

$\mathsf{Challenge}_{A,t}(\lambda)$*:*

1. *$M_0 \leftarrow \mathbf{B}_1(S, 1^\lambda)$.*
2. *Initialize $Q(\cdot)$ with M_0.*
3. *$x^* \leftarrow A^{Q(\cdot)}(1^\lambda, S)$ where A performs at most t (adaptive) queries $x_1, \ldots, x_t$ to the interface $Q(\cdot)$.*
4. *If $x^* \notin S \cup \{x_1, \ldots, x_t\}$ and $Q(x^*) = 1$ output 1, otherwise output 0.*

If $\mathbf{B}$ is not (n, t, ε)-resilient then we say there exists an adversary A that can (n, t, ε)-attack $\mathbf{B}$.

If $\mathbf{B}$ is resilient for any polynomial number of queries we say it is *strongly resilient.*

Definition 5 (Strongly resilient). *We say that $\mathbf{B}$ is an (n, ε)-strongly resilient Bloom filter, if for large enough security parameter λ and any polynomial $t = t(\lambda)$ we have that $\mathbf{B}$ is an (n, t, ε)-adversarial resilient Bloom filter.*

Remark 1. Notice that in Definitions 2 and 4 the adversary gets the set S as an additional input. This strengthens the definition of the resilient Bloom filter such that even given the set S it is hard to find false positives. An alternative definition might be to not give the adversary the set and also not require that $x^* \notin S$. However, our results of Theorem 1 hold even if the adversary does not get the set. That is, the algorithm that predicts a false positive makes no use of the set S, either then checking that $x^* \notin S$. Moreover, the construction in Theorem 2 holds in both cases, even against adversaries that do get the set.

An important parameter is the memory use of a Bloom filter $\mathbf{B}$. We say $\mathbf{B}$ uses $m = m(n, \lambda, \varepsilon)$ bits of memory if for any set S of size n the largest representation is of size at most m. The desired properties of Bloom filters is to have m as small as possible and to answer membership queries as fast as possible. Let $\mathbf{B}$ be a (n, ε)-Bloom filter that uses m bits of memory. Carter et al. [7] proved a lower bound on the memory use of any Bloom filter showing that $m \geq n \log \frac{1}{\varepsilon}$ (or written equivalently as $\varepsilon \geq 2^{-\frac{m}{n}}$). This leads us to defining the minimal error of $\mathbf{B}$.

Definition 6 (Minimal error). *Let $\mathbf{B}$ be an (n, ε)-Bloom filter that uses m bits of memory. We say that $\varepsilon_0 = 2^{-\frac{m}{n}}$ is the minimal error of $\mathbf{B}$.*

Note that using Carter's lower bound we get that for any (n, ε)-Bloom filter its minimal error ε_0 always satisfies $\varepsilon_0 \leq \varepsilon$. Also, a trivial Bloom filter can always store the set S precisely using $m = \log \binom{u}{n} \approx n \log \left(\frac{u}{n}\right)$ bits. Using the $m \geq n \log \frac{1}{\varepsilon}$ lower bound we get that a Bloom filter is trivial if $\varepsilon > \frac{n}{u}$. Moreover, if u is super-polynomial in n, and ε is negligible in n then any polynomial-time adversary has only negligible chance in finding any false positive, and again we say that the Bloom filter is trivial.

Definition 7 (Non-trivial Bloom filter). *Let $\mathbf{B}$ be an (n, ε)-Bloom filter that uses m bits of memory and let ε_0 be the minimal error of $\mathbf{B}$ (see Definition 6). We say that $\mathbf{B}$ is* non-trivial *if there exists a constant $c \geq 1$ such that $\varepsilon_0 > \max\left\{\frac{n}{u}, \frac{1}{n^c}\right\}$.*

3 Our Techniques

3.1 One-Way Functions and Adversarial Resilient Bloom Filters

We present the main ideas and techniques of the equivalence of adversarial resilient Bloom filters and one-way functions (i.e., the proof of Theorems 1 and 2). The simpler direction is showing that the existence of one-way functions implies the existence of adversarial resilient Bloom filters. Actually, we show that any Bloom filter can be efficiently transformed to be adversarial resilient with essentially the same amount of memory. The idea is simple and works in general for other data structures as well: apply a pseudo-random permutation of the input and then send it to the original Bloom filter. The point is that an adversary has almost no advantage in choosing the inputs adaptively, as they are all randomized by the permutation, while the correctness properties remain under the permutation.

The other direction is more challenging. We show that if one-way functions do not exist then any non-trivial Bloom filter can be 'attacked' by an efficient adversary. That is, the adversary performs a sequence of queries and then outputs an element x^* (that was not queried before) which is a false positive with high probability. We give two proofs: One for the case where the Bloom filter has a steady representation and one for an unsteady representation.

The main idea is that although we are given only oracle access to the Bloom filter, we are able to construct an (approximate) simulation of it. We use techniques from machine learning to (efficiently) 'learn' the internal memory of the Bloom filter, and construct the simulation. The learning task for steady and unsteady Bloom filters is quite different and each yield a simulation with different guarantees. Then we show how to exploit each simulation to find false positives without querying the real Bloom filter.

In the steady case, we state the learning process as a 'PAC learning' [29] problem. We use what's known as 'Occam's Razor' which states that any hypothesis consistent on a large enough random training set will have a small error. Finally, we show that since we assume that one-way functions do not exist then we are able to find a consistent hypothesis in polynomial-time. Since the error is small, the set of false positive elements defined by the real Bloom filter is approximately the same set of false positive elements defined by the simulator.

Handling Bloom filters with an unsteady representation is more challenging. Recall that such Bloom filters are allowed to randomly change their internal representation after each query. In this case, we are trying to learn a distribution that might change after each sample. We describe two examples of Bloom filters with unsteady representations which seem to capture the main difficulties of the unsteady case.

The first example considers any ordinary Bloom filter with error rate $\varepsilon/2$, where we modify the query algorithm to first answer 'Yes' with probability $\varepsilon/2$ and otherwise continue with its original behavior. The resulting Bloom filter has an error rate of ε. However, its behaviour is tricky: When observing its responses, elements can alternate between being false positive and negatives, which makes the learning task much harder.

The second example consists of two ordinary Bloom filters with error rate ε, both initialized with the set S. At the beginning only the first Bloom filter is used, and after a number of queries (which may be chosen randomly) only the second one is used. Thus, when switching to the second Bloom filter the set of false positives changes completely. Notice that while first Bloom filter was used exclusively, no information was leaked about the second. This example proves that any algorithm trying to 'learn' the memory of the Bloom filter cannot perform a fixed number of samples (as does our learning algorithm for the steady representation case).

To handle these examples we apply the framework of adaptively changing distributions (ACDs) presented by Naor and Rothblum [20], which models the task of learning distributions that can adaptively change after each sample was studied. Their main result is that if one-way functions do not exist then there exists an efficient learning algorithm that can approximate the next activation of the ACD, that is, produce a distribution that is statistically close to the distribution of the next activation of the ACD. We show how to facilitate (a slightly modified version of) this algorithm to learn the unsteady Bloom filter and construct a simulation. One of the main difficulties is that since we get only a statistical distance guarantee, then a false positive for the simulation need not be a false positive for the real Bloom filter. Nevertheless, we show how to estimate whether an element is a false positive in the real Bloom filter.

3.2 Computationally Unbounded Adversaries

In Theorem 3 we construct a Bloom Filter that is resilient against any unbounded adversary for a given number (t) of queries. One immediate solution would be to imitate the construction of the computationally bounded case while replacing the pseudo-random permutation with a $k = (t+n)$-wise independent hash function. Then, any set of t queries along with the n elements of the set would behave as truly random under the hash function. The problem with this approach is that the representation of the hash function is too large: It is $O(k \log |U|)$ which is more than the number of bits needed for a precise representation of the set S. Turning to almost k-wise independence does not help either. First, the memory will still be too large (it can be reduced to $O(n \log n \log \frac{1}{\varepsilon} + t \log n \log \frac{1}{\varepsilon})$ bits) and second, almost k-wise guarantees works only for sets chosen in advance, where the point of a resilient Bloom filter is to handle adaptively chosen sets.

Carter et al. [7] presented a general transformation from any exact dictionary to a Bloom filter. The idea was simple: storing x in the Bloom filter translates to storing $g(x)$ in a dictionary for some (universal) hash function $g : U \to V$, where $|V| = \frac{n}{\varepsilon}$. The choice of the hash function and underlying dictionary are important as they determine the performance and memory size of the Bloom filter. Notice that, at this point replacing g with a $k = (t+n)$-wise independent hash function (or an almost k-independent hash function) yields the same problems discussed above. Nevertheless, this is our starting point where the final construction is quite different. Specifically, we combine two main ingredients: Cuckoo hashing and a highly independent hash function tailored for this construction.

For the underlying dictionary in the transformation we use the Cuckoo hashing construction [26, 25]. Using cuckoo hashing as the underlying dictionary was already shown to yield good constructions for Bloom filters by Pagh et al. [24] and Arbitman et al. [1]. Among the many advantages of Cuckoo hashing (e.g., succinct memory representation, constant lookup time) is the simplicity of its structure. It consists of two tables T_1 and T_2 and two hash functions h_1 and h_2 and each element x in the Cuckoo dictionary resides in either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. However, we use this structure a bit differently. Instead of storing $g(x)$ in the dictionary directly (as the reduction of Carter et al. suggests) which would resolve to storing $g(x)$ at either $T_1[h_1(g(x))]$ or $T_2[h_2(g(x))]$ we store $g(x)$ at either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. That is, we use the full description of x to decide where x is stored but eventually store only a hash of x (namely, $g(x)$). Since each element is compared only with two cells, this lets us improve the analysis of the reduction which reduce the size of V to $O\left(\frac{1}{\varepsilon}\right)$ (instead of $\frac{n}{\varepsilon}$).

To initialize the hash function g, instead of using a universal hash function we use a very high independence function (which in turn is also constructed based on cuckoo hashing) based on the work of Pagh and Pagh [23] and Dietzfelbinger and Woelfel [11]. They show how to construct a family G of hash functions such that on any given set of k inputs it behaves like a truly random function with high probability. Furthermore, a function in G can be evaluated in constant time (in the RAM model), and its description can be stored using roughly $O(k \log |V|)$ bits (where V is the range of the function).

Note that the guarantee of the function acting random holds only for sets S of size k that are *chosen in advance*. In our case the set is not chosen in advance but rather chosen adaptively and adversarially. However, Berman et al. [3] showed that the same construction of Pagh and Pagh actually holds *even when the set of queries is chosen adaptively.*

At this point, one solution would be to use the family of functions G setting $k = t + n$, with the analysis of Berman et al. as the hash function g and the structure of the Cuckoo hashing dictionary. To get an error of ε, we set $|V| = O\left(\log \frac{1}{\varepsilon}\right)$ and get an adversarial resilient Bloom filter that is resilient for t queries and uses $O\left(n \log \frac{1}{\varepsilon} + t \log \frac{1}{\varepsilon}\right)$ bits of memory. However, our goal is to get a memory size of $O\left(n \log \frac{1}{\varepsilon} + t\right)$.

To reduce the memory of the Bloom filter even further, we use the family G a bit differently. Let $\ell = O\left(\log \frac{1}{\varepsilon}\right)$, and set $k = O\left(t/\ell\right)$. We define the function g to be a concatenation of ℓ independent instances g_i of functions from G, each outputting a single bit ($V = \{0, 1\}$). Using the analysis of Berman et al. we get that each of them behaves like a truly random function for any sequence of k adaptively chosen elements. Consider an adversary performing t queries. To see how this composition of hash functions helps reduce the independence needed, consider the comparisons performed in a query between $g(x)$ and some value y being performed bit by bit. Only if the first pair of bits are equal we continue to compare the next pair. The next query continues from the last pair compared, in a cyclic order. For any set of k elements, the probability of the two bits to be equal is $1/2$. Thus, with high probability, only a constant number of bits will be compared during a single query. That is, in each query only a constant number of functions g_i will be involved and "pay" in their independence, where the rest remain untouched. Altogether, we get that although there are t queries performed, we have ℓ different functions and each function g_i is involved in at most $O(t/\ell) = k$ queries (with high probability). Thus, the view of each function remains random on these elements. This results in an adversarial resilient Bloom filter that is resilient for t queries and uses only $O(n \log \frac{1}{\varepsilon} + k \log \frac{1}{\varepsilon}) = O(n \log \frac{1}{\varepsilon} + t)$ bits of memory.

4 Preliminaries

We start with some general notation. We denote by $[n]$ the set of numbers $\{1, 2, \ldots, n\}$. We denote by $\mathsf{neg} : \mathbb{N} \to \mathbb{R}$ a function such that for every positive integer c there exists an integer N_c such that for all $n > N_c$, $\mathsf{neg}(n) < 1/n^c$. Finally, throughout this paper we denote by log the base 2 logarithm.

Definition 8 (One-Way Functions). *A function f is said to be* one-way *if:*

1. *There exists a polynomial-time algorithm A such that $A(x) = f(x)$ for every $x \in \{0,1\}^*$.*
2. *For every probabilistic polynomial-time algorithm A' and large enough n,*

$$\Pr[A'(1^n, f(x)) \in f^{-1}(f(x))] < \mathsf{neg}(n),$$

where the probability is taken uniformly over $x \in \{0,1\}^n$ and the internal randomness of A'.

Definition 9 (Universal Hash Family). *A family of functions $\mathcal{H} = \{h : U \to [m]\}$ is called universal if for any $x_1 \neq x_2$: $\Pr_{h \in \mathcal{H}}[h(x_1) = h(x_2)] \leq \frac{1}{m}$.*

5 Adversarial Resilient Bloom Filters and One-Way Functions

In this section we show that adversarial resilient Bloom filters are (existentially) equivalent to one-way functions (see Definition 8). We begin by showing that if one-way functions do not exist, then any Bloom filter can be 'attacked' by an efficient algorithm in a strong sense:

Theorem 4. *Let $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ be any non-trivial Bloom filter of n elements that uses m bits of memory and let ε_0 be the minimal error of $\mathbf{B}$. If one-way function do not exist, then for any constant $\varepsilon < 1$, $\mathbf{B}$ is not (n, t, ε)-adversarial resilient for $t = O\left(m/\varepsilon_0^2\right)$.*

We give two different proofs; The first is self contained (e.g. we do not even have to use the Impagliazzo-Luby [15] technique of finding a random inverse), but, deals only with Bloom filters with steady representations. The second handles Bloom filters with unsteady representations, and uses the framework of adaptively changing distributions of [20].

5.1 A Proof for Bloom Filters with Steady Representations

Overview: We prove Theorem 4 for the case of steady representation (see Definition 1). Actually, for the steady case the theorem holds even for $t = O(m/\varepsilon_0)$.

Assume that there are no one-way functions. We want to construct an adversary that can attack the Bloom filter. We define a function f to be a function that gets a set S, random bits r, and elements $x_1, \ldots, x_t$, computes $M = \mathbf{B}_1(S; r)$

and outputs these elements along with their evaluation on $\mathbf{B}_2(M, \cdot)$ (i.e. for each element x_i the value $\mathbf{B}_2(M, x_i)$). Since f is not one-way, there is an efficient algorithm that can invert it with high probability[2]. That is, the algorithm is given a random set of elements labeled whether they are (false) positives or not and it outputs a set S' and bits r'. For $M' = \mathbf{B}_1(S'; r')$ the function $\mathbf{B}_2(M', \cdot)$ is consistent with $\mathbf{B}_2(M, \cdot)$ for all the elements $x_1, \ldots, x_t$. For a large enough set of queries we show that $\mathbf{B}_2(M', \cdot)$ is actually a good approximation of $\mathbf{B}_2(M, \cdot)$ as a boolean function. We use $\mathbf{B}_2(M', \cdot)$ to find an input x^* such that $\mathbf{B}_2(M', x^*) = 1$ and show that $\mathbf{B}_2(M, x^*) = 1$ as well with high probability. This contradicts $\mathbf{B}$ being adversarial-resilient and proves that f is a (weak) one-way function. See the full paper for more details [22].

5.2 Handling Unsteady Bloom Filters

We describe the proof of the general statement of Theorem 4, i.e., handle Bloom filters with an unsteady-representation as well. A Bloom filter with an unsteady representation (see Definition 3) has a *randomized* query algorithm and may change the underlying representation after each query. We want to show that if one-way functions do not exist then we can construct an adversary, Attack, that 'attacks' this Bloom filter. The proof of this case is more involved and we show a simpler version that has an additional assumption (for the full proof see [22]).

Hard-core positives. Let $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ be an (n, ε)-Bloom filter with an unsteady representation that uses m bits of memory (see Definition 3). Let M and M' be two representations of a set S generated by $\mathbf{B}_1$. In the previous proof in Section 5.1, given a representation M we considered $\mathbf{B}_2(M, \cdot)$ as a boolean function. We defined the function $\mu(M)$ to measure the number of positives in $\mathbf{B}_2(M, \cdot)$ and we defined the error between two representations $\mathsf{err}(M, M')$ to measure the fraction of inputs that the two boolean functions agree on. These definitions make sense only when $\mathbf{B}_2$ is deterministic and does not change the representation. However, in the case of Bloom filters with unsteady representations we need to modify the definitions to have new meanings.

Given a representation M consider the query interface $Q(\cdot)$ initialized with M. For an element x, the probability of x being a false positive is $\Pr[Q(x) = 1] = \Pr[\mathbf{B}_2(M, x) = 1]$. Recall that after querying $Q(\cdot)$, the interface updates its representation and the probability of x being a false positive might change (it could be higher or lower). We say that x is a 'hard-core positive' if after any arbitrary sequence of queries we have that $\Pr[Q(x) = 1] = 1$. That is, the query interface will always response with a 'Yes' on x even after any sequence of queries. Then, we define $\mu(M)$ to be the set of hard-core positive elements in U. Note that over the time, the size of $\mu(M)$ might grow, but it can never become smaller. The following claim proves that for almost all sets S the number of *hard-core* positives is large (see [22] for the proof).

[2] The algorithm can invert the function for infinitely many input sizes. Thus, the adversary we construct will succeed in its attack on the same (infinitely many) input sizes.

Claim. For any Bloom filter with minimal error ε_0 it holds that:

$$\Pr_S\left[\exists r : \mu\left(M_r^S\right) \leq \frac{\varepsilon_0}{8}\right] \leq 2^{-n}$$

where the probability is taken a random set S of size n from the universe U.

The distribution D_M. As we can not talk about the *function* $\mathbf{B}_2(M, \cdot)$ (as in the steady case) we use terms of *distributions*. For any representation M define the distribution D_M: Sample k elements at random $x_1, \ldots, x_k$ (k will be determined later), and output $(x_1, \ldots, x_k, Q(x_1), \ldots, Q(x_k))$. Note that the underlying representation M changes after each query. Formally, the algorithm for D_M is:

1. Sample $x_1, \ldots, x_k \in U$ uniformly at random.
2. For $i = 1, \ldots, k$: compute $y_i = Q(x_i)$.
3. Output $(x_1, \ldots, x_k, y_1, \ldots, y_k)$.

Let M_0 be a representation of a random set S generated by $\mathbf{B}_1$, and let ε_0 be the minimal error of $\mathbf{B}$. Assume that one-way functions do not exists. Our goal is to construct an algorithm Attack that will 'attack' $\mathbf{B}$, that is, it is given access to $Q(\cdot)$ initialized with M_0 (M_0 is secret and not known to Attack) it must find an non-set element x^* such that $\Pr[Q(x) = 1] \geq 2/3$.

Consider the distribution D_{M_0}, and notice that given access to $Q(\cdot)$ we can perform a single sample from D_{M_0}. Let M_1 be the random variable of the resulting representation after the sample. Then, we can sample from the distribution D_{M_1}, and then D_{M_2} and so on. We describe a simplified version of the proof where we assume that M_0 is known to the adversary. This version seems to captures the main ideas.

Attacking when M_0 is known. Suppose that after activating D_{M_0} for r rounds we are given the initial state M_0 (of course, in the actual execution M_0 is secret and later we show how to overcome this assumption). Let $p_1, \ldots, p_r$ be the outputs of the rounds (that is, $p_i = (x_1, \ldots, x_k, y_1, \ldots, y_k)$). For a specific output p_i we say that x_j was labeled '1' if $y_j = 1$.

Denote by $D_{M_0}(p_0, \ldots, p_r)$ the distribution over the $(r+1)^{\text{th}}$ activation of D_{M_0} conditioned on the first r activations resulting in the states $p_0, \ldots, p_r$. Computational issues aside, the distribution $D_{M_0}(p_0, \ldots, p_r)$ can be sampled by enumerating all random strings such that when applied to D_{M_0} yield the output $p_0, \ldots, p_r$, sampling one of them, and outputting the representations generated by the random string chosen. Moreover, define $D_{M_0}(p_0, \ldots, p_r; x_1, \ldots, x_k)$ to be the distribution $D_{M_0}(p_0, \ldots, p_r)$ conditioned on that the elements chosen in the sample are $x_1, \ldots, x_k$. We also define $D(p_0, \ldots, p_r)$ to be the same distribution as $D_{M_0}(p_0, \ldots, p_r)$ only where the representation M_0 is also chosen at random (according to $\mathbf{B}_1(S)$).

We define an (inefficient) adversary Attack (see Figure 1) that (given M_0) can attack the Bloom filter, that is, find an element x^* that was not queried before and is a false positive with high probability.

Set $k = 160/\varepsilon_0$ and $\ell = 100k$. Then we get the following claims.

```
The Algorithm Attack

Given: The representation $M_0$.

Input: $1^\lambda$.

1. Sample $x_1, \ldots, x_k \in U$ at random.
2. For $i \in [\ell]$ sample $D_{M_0}(p_0, \ldots, p_r; x_1, \ldots, x_k)$ to get $y_{i1}, \ldots, y_{ik}$.
3. If there exists an index $j \in [k]$ such that for all $i \in [\ell]$ it holds that $y_{ij} = 1$:
   (a) Set $x^* = x_j$.
   (b) Query $Q(x_1), \ldots, Q(x_{j-1})$.
4. Otherwise set $x^*$ to be an arbitrary element in $U$.
5. Output $x^*$.
```

Fig. 1. The description of the algorithm Attack.

Claim. There is a common x_j: With probability 99/100 there exist a $1 \leq j \leq k$ such that for all $i \in [\ell]$ it holds that $y_{ij} = 1$, where the probability is over the random choice of S and $x_1, \ldots, x_k$.

Proof. Let M_r be the resulting representation of the r^{th} activation of $D_{M_0}(p_0, \ldots, p_r; x_1, \ldots, x_k)$. We have seen that with probability $1-2^{-n}$ over the choice of S for any M_0 we have that the set of hard-core positives satisfy $|\mu(M_0)| \geq \varepsilon_0/16$. By the definition of the hard-core positives, the set $\mu(M_0)$ may only grow after each query. Thus, for each sample from $D_{M_0}(p_0, \ldots, p_r; x_1, \ldots, x_k)$ we have that $\mu(M_0) \subseteq \mu(M_r)$. If $x_j \in \mu(M_0)$ then $x_j \in \mu(M_r)$ and thus $y_{ij} = 1$ for all $i \in [\ell]$. The probability that all elements $x_1, \ldots, x_k$ are sampled outside the set $\mu(M_0)$ is at most $(1 - \varepsilon_0/16)^k \leq e^{-10}$ (over the random choices of the elements). All together we get that probability of choosing a 'good' S and a 'good' sequence $x_1, \ldots, x_t$ is at least $1 - 2^{-n} + e^{-10} \geq 99/100$.

Claim. Let M_r be the underlying representation of the interface $Q(\cdot)$ at the time right after sampling $p_0, \ldots, p_r$. Then, with probability at least 98/100 the algorithm Attack outputs an element x^* such that $Q(x^*) = 1$, where the probability is taken over the randomness of Attack, the sampling of $p_0, \ldots, p_r$, and $\mathbf{B}$.

Proof. Consider the distribution $D_{M_0}(p_0, \ldots, p_r; x_1, \ldots, x_k)$ to work as follows: First a representation M is sampled conditioned on starting from M_0 and outputting the states $p_0, \ldots, p_r$ and then we compute $y_j = \mathbf{B}_2(M, x_j)$. Let $M'_1, \ldots, M'_\ell$ be the representations chosen during the run of Attack. Note that M_r is chosen from the same distribution that $M'_1, \ldots, M'_\ell$ are sampled from. Thus, we can think of M_r of being picked after the choice of $x_1, \ldots, x_k$. That is, we sample $M'_1, \ldots, M'_{\ell+1}$, and choose one of them at random to be M_r, and the rest are relabeled as $M'_1, \ldots, M'_\ell$. Now, for any x_j, the probability that for all i, M'_i will answer '1' on x_j but M_r will answer '0' on x_j is at most $1/\ell$. Thus, the probability that there exist any such x_j is at most $\frac{k}{\ell} = \frac{k}{100k} = 1/100$. Altogether, the probability that A find such an x_j that is always labeled '1' and that M_r answers '1' on it, is at least $99/100 - 1/100 = 98/100$.

We are left to show how to construct the algorithm Attack so that it will run in polynomial-time and perform the same tasks *without* knowing M_0. One difficulty (which was discussed in Section 3), is that the number of samples r must be chosen as a function of the samples and cannot be fixed in advance. Algorithms for such tasks were studied in the framework Naor and Rothblum [20] on adaptively changing distributions. The full proof is given at [22].

5.3 A Construction Using Pseudorandom Permutations

We have seen that Bloom filters that are adversarial resilient require using one-way functions. To complete the equivalence, we show that pseudorandom permutations and functions can be used to construct adversarial resilient Bloom filters. Actually, we show that any Bloom filter can be efficiently transformed to be adversarial resilient with essentially the same amount of memory. The idea is simple and can work in general for other data structures as well: On any input x we compute a pseudo-random permutation of x and send it to the original Bloom filter. The full proof is given at [22].

Theorem 5. *Let* $\mathbf{B}$ *be an* (n, ε)*-Bloom filter using* m *bits of memory. If pseudorandom permutations exist, then for any security parameter* λ *there exists an* $(n, \varepsilon + \mathsf{neg}(\lambda))$*-strongly resilient Bloom filter with memory* $m' = m + \lambda$*.*

6 Computationally Unbounded Adversary

In this section, we extend the discussion of adversarial resilient Bloom filters to ones against computationally *unbounded* adversaries. First, notice that the attack of Theorem 4 holds in this case as well, since an unbounded adversary can invert any function (with probability 1). Formally, we get the following:

Corollary 1. *Let* $\mathbf{B} = (\mathbf{B}_1, \mathbf{B}_2)$ *be any non-trivial Bloom filter of* n *elements that uses* m *bits of memory and let* ε_0 *be the minimal error of* $\mathbf{B}$*. Then for any constant* $\varepsilon < 1$*,* $\mathbf{B}$ *is not* (n, t, ε)*-adversarial resilient against unbounded adversaries for* $t = O\left(\frac{m}{\varepsilon_0^2}\right)$*.*

As we saw, any (n, ε)-Bloom filter must use at least $n \log \frac{1}{\varepsilon}$ bits of memory. We show how to construct Bloom Filters that are resilient against *unbounded* adversaries for t of queries while using only $O\left(n \log \frac{1}{\varepsilon} + t\right)$ bits of memory (for a discussion on the optimality of the number of queries t see the full paper [22]).

Theorem 6. *For any* $n, t \in \mathbb{N}$*, and* $\varepsilon > 0$ *there exists an* (n, t, ε)*-resilient Bloom filter (against unbounded adversaries) that uses* $O(n \log \frac{1}{\varepsilon} + t)$ *bits of memory.*

Our construction uses two main ingredients: Cuckoo hashing and a very high independence hash family G. We begin by describing these ingredients.

The Hash Function Family G. Pagh and Pagh [23] and Dietzfelbinger and Woelfel [11] (see also Aumuller et al. [2]) showed how to construct a family G of hash functions $g : U \to V$ so that on any set of k inputs it behaves like a truly random function with high probability $(1 - 1/\mathsf{poly}(k))$. Furthermore, g can be evaluated in constant time (in the RAM model), and its description can be stored using $(1+\alpha)k \log |V| + O(k)$ bits (where here α is an arbitrarily small constant).

Note that the guarantee of g acting as a random function holds for any set S that is *chosen in advance.* In our case the set is not chosen in advance but chosen adaptively and adversarially. However, Berman et al. [3] showed that the same line of constructions, starting with Pagh and Pagh, actually holds *even when the set of queries is chosen adaptively.* That is, for any distinguisher that can adaptively choose k inputs, the advantage of distinguishing a function $g \in_R G$ from a truly random function is polynomially small[3].

Set $\ell = 4 \log \frac{1}{\varepsilon}$. Our function g will be composed of the concatenation of ℓ one bit functions $g_1, g_2, \ldots g_\ell$ where each g_i is selected independently from a family G where $V = \{0,1\}$ and $k = 2t/\log \frac{1}{\varepsilon}$. For a random $g_i \in_R G$:

- There is a constant c (which we can choose) so that for any adaptive distinguisher that issues a sequence of k adaptive queries g_i the advantage of distinguishing between g_i and an exact k-wise independent function $U \to V$ is bounded by $\frac{1}{k^c}$.
- g_i can be represented using $(1+\alpha)k\ell = O(t)$ bits.
- g_i can be evaluated in constant time.

Thus, the representation of g requires $O(t)$ bits. The evaluation of g at a given point x takes $O(\ell) = O\left(\log \frac{1}{\varepsilon}\right)$ time.

Cuckoo Hashing. Cuckoo hashing is a data structure for dictionaries introduced by Pagh and Rodler [26]. It consists of two tables T_1 and T_2, each containing r cells where r is slightly larger than n (that is, $r = (1+\alpha)n$ for some small constant α) and two hash functions $h_1, h_2 : U \to [r]$. The elements are stored in the two tables so that an element x resides at either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. Thus, the lookup procedure consists of one memory accesses to each table plus computing the hash functions. (This description ignores insertions.)

We assume that $n > \log u$ (we can actually let n go as low as $O(\log \log u)$ using almost pair-wise independent hashing). Our construction of an adversarial resilient Bloom filter is:

Setup. The input is a set S of size n. Sample a function g by sampling ℓ functions $g_i \in_R G$ and initialize a Cuckoo hashing dictionary D of size n (with $\alpha = 0.1$) as described above. That is, D has two tables T_1 and T_2 each of size $1.1n$, two hash functions h_1 and h_2, and each element x will reside at either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. Insert the elements of S into D. Then, go over the two tables T_1 and

[3] Any exactly k-wise independent function is also good against k adaptive queries, but this is not necessarily the case for *almost* k-wise

T_2 and at each cell replace each x with $g(x)$. That is, now for each $x \in S$ we have that $g(x)$ resides at either $T_1[h_1(x)]$ or $T_2[h_2(x)]$. Put $\perp$ in the empty locations. The final memory of the Bloom Filter is the memory of D and the representation of g. The dictionary D consists of $O(n)$ cells, each of size $|g(x)| = O(\log \frac{1}{\varepsilon})$ bits and therefore D and g together can be represented by $O(n \log \frac{1}{\varepsilon} + t)$ bits.

Lookup. On input x we answer whether 'Yes' if either $T_1[h_1(x)] = g(x)$ or $T_2[h_2(x)] = g(x)$. The full proof of this construction is given at [22].

References

1. Arbitman, Y., Naor, M., Segev, G.: Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In: FOCS. pp. 787–796. IEEE Computer Society (2010)
2. Aumüller, M., Dietzfelbinger, M., Woelfel, P.: Explicit and efficient hash families suffice for cuckoo hashing with a stash. Algorithmica 70(3), 428–456 (2014), http://dx.doi.org/10.1007/s00453-013-9840-x
3. Berman, I., Haitner, I., Komargodski, I., Naor, M.: Hardness preserving reductions via cuckoo hashing. In: TCC. pp. 40–59 (2013)
4. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13, 422–426 (1970)
5. Blum, A., Furst, M.L., Kearns, M.J., Lipton, R.J.: Cryptographic primitives based on hard learning problems. In: Advances in Cryptology, CRYPTO. Lecture Notes in Computer Science, vol. 773, pp. 278–291. Springer (1993)
6. Broder, A., Mitzenmacher, M.: Network applications of Bloom filters: A survey. In: Internet Mathematics. pp. 636–646 (2002)
7. Carter, L., Floyd, R., Gill, J., Markowsky, G., Wegman, M.: Exact and approximate membership testers. In: Proceedings of the tenth annual ACM symposium on Theory of computing. pp. 59–65. STOC '78, ACM, New York, NY, USA (1978), http://doi.acm.org/10.1145/800133.804332
8. Chazelle, B., Kilian, J., Rubinfeld, R., Tal, A.: The bloomier filter: an efficient data structure for static support lookup tables. In: SODA. pp. 30–39. SIAM (2004)
9. Deng, F., Rafiei, D.: Approximately detecting duplicates for streaming data using stable Bloom filters. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. pp. 25–36. ACM (2006)
10. Dharmapurikar, S., Krishnamurthy, P., Sproull, T.S., Lockwood, J.W.: Deep packet inspection using parallel Bloom filters. IEEE Micro 24(1), 52–61 (2004)
11. Dietzfelbinger, M., Woelfel, P.: Almost random graphs with simple hash functions. In: Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA. pp. 629–638. ACM (2003)
12. Dwork, C., McSherry, F., Nissim, K., Smith, A.: Calibrating noise to sensitivity in private data analysis. In: Theory of Cryptography, Third Theory of Cryptography Conference, TCC. vol. 3876, pp. 265–284. Springer (2006)
13. Fan, L., Cao, P., Almeida, J.M., Broder, A.Z.: Summary cache: a scalable wide-area web cache sharing protocol. IEEE/ACM Trans. Netw. 8(3), 281–293 (2000)
14. Hardt, M., Woodruff, D.P.: How robust are linear sketches to adaptive inputs? In: STOC. pp. 121–130. ACM (2013)

15. Impagliazzo, R., Luby, M.: One-way functions are essential for complexity based cryptography (extended abstract). In: FOCS. pp. 230–235. IEEE Computer Society (1989)
16. Lipton, R.J., Naughton, J.F.: Clocked adversaries for hashing. Algorithmica 9(3), 239–252 (1993)
17. Manber, U., Wu, S.: An algorithm for approximate membership checking with application to password security. Inf. Process. Lett. 50(4), 191–197 (1994)
18. Mironov, I., Naor, M., Segev, G.: Sketching in adversarial environments. SIAM J. Comput. 40(6), 1845–1870 (2011), http://dx.doi.org/10.1137/080733772
19. Mullin, J.K.: Optimal semijoins for distributed database systems. Software Engineering, IEEE Transactions on 16(5), 558–560 (1990)
20. Naor, M., Rothblum, G.N.: Learning to impersonate. In: ICML. ACM International Conference Proceeding Series, vol. 148, pp. 649–656. ACM (2006)
21. Naor, M., Yogev, E.: Sliding Bloom filters. In: Algorithms and Computation - 24th International Symposium, ISAAC. vol. 8283, pp. 513–523. Springer (2013)
22. Naor, M., Yogev, E.: Bloom filters in adversarial environments. CoRR abs/1412.8356 (2014), http://arxiv.org/abs/1412.8356
23. Pagh, A., Pagh, R.: Uniform hashing in constant time and optimal space. SIAM J. Comput. 38(1), 85–96 (2008)
24. Pagh, A., Pagh, R., Rao, S.S.: An optimal Bloom filter replacement. In: SODA. pp. 823–829. SIAM (2005)
25. Pagh, R.: Cuckoo hashing. In: Encyclopedia of Algorithms. Springer (2008)
26. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms 51(2), 122–144 (2004)
27. Putze, F., Sanders, P., Singler, J.: Cache-, hash-, and space-efficient bloom filters. ACM Journal of Experimental Algorithmics 14 (2009)
28. Tarkoma, S., Rothenberg, C.E., Lagerspetz, E.: Theory and practice of bloom filters for distributed systems. IEEE Communications Surveys and Tutorials 14(1), 131–155 (2012)
29. Valiant, L.G.: A theory of the learnable. Commun. ACM 27(11), 1134–1142 (1984)
30. Yan, J., Cho, P.L.: Enhancing collaborative spam detection with bloom filters. In: ACSAC. pp. 414–428. IEEE Computer Society (2006)
31. Zhang, L., Guan, Y.: Detecting click fraud in pay-per-click streams of online advertising networks. In: ICDCS. pp. 77–84. IEEE Computer Society (2008)
32. Zhu, Y., Jiang, H., Wang, J.: Hierarchical Bloom filter arrays (hba): a novel, scalable metadata management system for large cluster-based storage. In: CLUSTER. pp. 165–174. IEEE Computer Society (2004)