Analyzing Memory Usage of Adversarially Resistant Bloom Filters

A. Anas Chentouf, Ningshan Karen Ma, Zi Song Yeoh, Yanyi Zhang

May 15, 2023

Abstract

Bloom filters are widely used data structures in various applications for approximate membership queries (AMQ). However, their usage in security-critical applications is limited due to the potential vulnerabilities to attacks by adversaries who can manipulate the filter to induce false positives. It is also limited by the need to store the representation of the bloom filter and the associated hash functions. Bloom filters are an important tool in cryptography due to their ability to efficiently represent sets and perform set membership tests. By leveraging their properties, it is possible to improve the efficiency, privacy, and security of various cryptographic protocols and applications.

In this project, we analyze the space-efficiency of bloom filters that are resilient against bounded and unbounded adversaries. Building on top work by Naor and Yogev, we show that we can construct adversarially resiliant AMQ filters with near-optimal memory usage under the correct assumptions.

1 Introduction

At the foundation of both the theory and implementation of modern computer science and cryptography is the study of data structures. In particular, the goal is to design efficient data structures that satisfy certain tasks, where efficiency is defined in terms of time and/or space complexity in some model of computation. In a security setting, we are also interested in the robustness of data structure in the face of adversarial agents.

Approximate Membership Query Filters (AMQ-Filter) are space-efficient probabilistic data structures that represent a set of elements S and allow the user to query whether an element x is in S, with some error margin. They approximate a solution to the Exact Membership Query problem. By introducing randomness, we have a tradeoff between complexity and accuracy - we allow some margin of error which corresponds to a rate of false positives, while obtaining a lower space complexity.

The most classic example of an AMQ-Filter is the Bloom filter. Roughly speaking, a Bloom filter is an array A of bits with k hash functions. Each hash function maps an element in S to a position in A. To insert an element, all hash functions are computed on the element and the corresponding bits in the array are set to 1. To check if an element exists, we check if each of the k positions the element would map to are set to 1. By setting the correct parameters, the space complexity is $\log_2(e)n\log_2(\epsilon^{-1}) \approx 1.44n\log_2(\epsilon^{-1})$ bits, where ϵ is the false positive rate. This almost achieves the information-theoretic lower bound, which is $n\log_2(\epsilon^{-1})$ bits. There exist constructions which achieve $(1+o(1))n\log_2(\epsilon^{-1})$ space (e.g. [DP08, Por08, NY13]), but they are typically more complicated. More importantly, none of these are resilient against adversarial queries.

There are several different notions of being resilient against adversarial queries that have been discussed (see [NO22] for a collection of some of these notions). For example, an adversary might be able to see if an element is a false positive after making their query, and make future queries based on the result. The standard analysis of bloom filters do not account for these adaptive adversaries, and the first work of this form is [NY14]. There exist constructions that take $O(n\log(\epsilon^{-1}))$ bits of space, but the constant factor is not explicitly computed (unlike the vanilla bloom filter which we know uses $\approx 1.44n\log_2(\epsilon^{-1})$ bits). The goal of this project is to analyze and optimize the space required to construct adversarial-resilient AMQ-filters.

1.1 Summary of Results

In this paper, we mainly focus on AMQ-filters that are secure under the security game defined in [NY14]. The security game is the following (see Figure 1):

- The adversary first picks a set $S = \{s_1, s_2, \ldots, s_n\}$, which will be inserted into the filter.
- The adversary makes up to t queries, $x_1, x_2, \ldots, x_t$ (after seeing the results of previous queries). Their goal is to find a false positive that has never been queried before.
- The adversary outputs $x \notin \{x_1, x_2, \ldots, x_t\}$ and receive final result from the bloom filter system.

We say that a filter is (n, t, ϵ)-resilient if no adversary can win the above game with more than ϵ probability.

We consider two types of adversaries: bounded-computation adversaries and unbounded-computation adversaries. Our goal is to determine the minimum memory usage of a (n, t, ϵ)-resilient filter.

For the bounded case, [NY14] showed that any AMQ-filter for the non-adversarial case can be converted into a (n, t, ϵ)-resilient filter by adding only λ bits of memory (where λ is the security parameter). Combining with the optimal construction for the non-adversarial case in [DP08], we

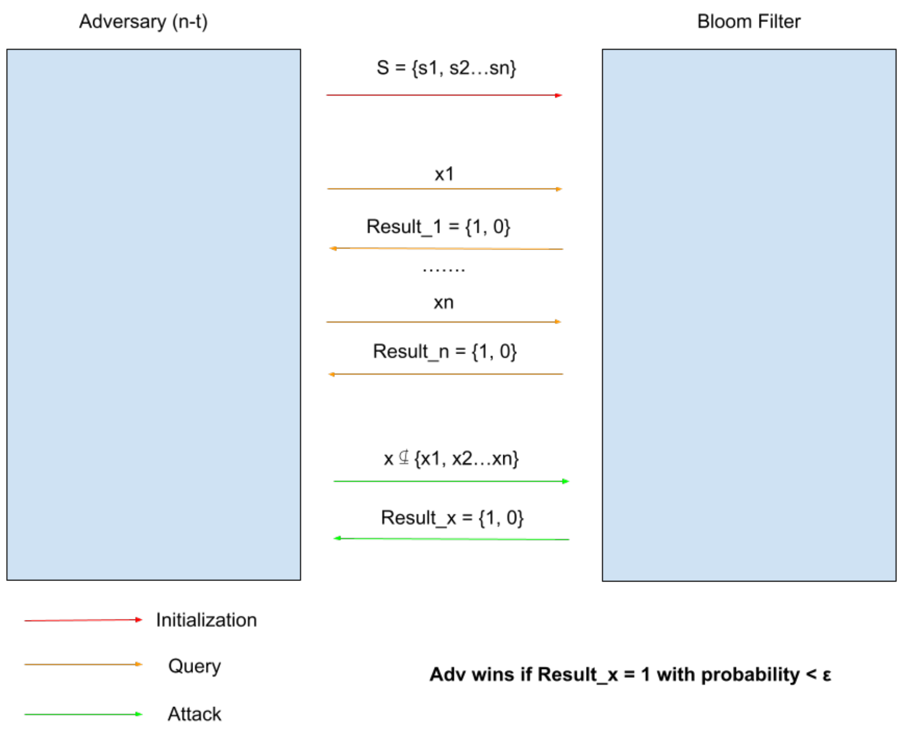

Figure 1: Formal Security Game Flow

show that we can achieve $(1+o(1))n\log\frac{1}{\epsilon}$ bits of memory usage, which is optimal up to lower order terms (since the entropy lower bound is $n\log\frac{1}{\epsilon}$).

For the unbounded case, the answer heavily depends on t. In particular, [NY14] showed that if $t=\omega(n)$, we cannot have a (n,t,ϵ)-resilient AMQ-filter with $O(n\log\frac{1}{\epsilon})$ bits of memory. Hence, we consider the regime where $t=O(n)$. [NY14] showed a construction using $O(n\log\frac{1}{\epsilon}+t)$ bits of memory, which is asymptotically optimal. However, their construction uses at least $2n\log\frac{1}{\epsilon}$ bits of memory (even when $t=0$), which is a 2-factor away from the optimal bound. We modify their construction to achieve a $(1+o(1))n\log\frac{1}{\epsilon}+\log_2(e)n$ memory (n,t,ϵ)-resilient filter assuming $t=o(n)$, which is close to optimal (in particular, it only differs from the theoretical lower bound by lower order terms as $\epsilon\to 0$).

1.2 Related Work

The first work to consider adaptive adversaries for correctness of AMQ-filters is by Naor and Yogev [NY14]. In their model, the adversary is allowed to choose the set S and allowed to make queries to the bloom filter, and return a guess for a false positive that has not been queried before. Following [NY14], [CPS19] analyzed probabilistic data structures which include several variants of bloom filters. Their notion of adversarial correctness is roughly about analyzing the probability of getting some number of false positives in a sequence of adaptive queries. There is also work which discusses adversaries that can perform repeated queries: if the filter is static and the adversary can repeat the same false positive many times, they can dramatically increase the average false positive rate over many queries (see [BFCG+18]). Unfortunately, in this setting, they showed that $\Omega(n\log n)$ bits of memory is required assuming that the universe is large enough. [NO22] attempted to unify these notions and study the relationship between different types of adaptive adversaries. They also constructed filters that satisfy their strongest definition of adversarial correctness and take $O(n \log_2(\epsilon^{-1}))$ space. However, the constant factors are not computed explicitly.

1.3 Practical Applications of AMQ-Filters

Approximate Membership Query (AMQ) data structures like Bloom filters are commonly used in scenarios where space efficiency is crucial and probabilistic answers are acceptable. Recalling that hash tables can be used as an Exact Membership Query (EMQ) data structure, we have that any use of hash tables could, theoretically, be replaced by that of bloom filters. In particular, they are used in instances when we can afford some margin of error. Here are some main applications of Bloom filters:

1. Private information retrieval Transfer: Bloom filters can be used in cryptographic applications such as private information retrieval (PIR).In a PIR protocol, a client wishes to retrieve a record from a database held by a server, without revealing which record is being requested. One way to achieve this is for the client to send a query that is constructed using a Bloom filter. The server can then use this Bloom filter to search its database for records that match the query, without learning which specific record is being requested.

2. Caching: Bloom filters can be used to speed up access to frequently accessed data by caching elements in memory. For instance, a web browser may use a Bloom filter to check whether a URL has been visited previously, allowing it to quickly determine whether to fetch the page from the server or retrieve it from the cache.

3. Network packet filtering: Bloom filters can be used to filter network packets, reducing the amount of data that needs to be inspected by more expensive deep packet inspection mechanisms. For example, an Internet Service Provider (ISP) may use a Bloom filter to identify packets that are known to contain spam, viruses or other malware.

4. Secure Multiparty Computation (SMC): In an SMC protocol, several parties wish to compute a function over their private inputs without revealing their inputs to each other. Bloom filters can be used in SMC protocols to perform set intersection operations without revealing the actual elements in the sets. Specifically, each party can construct a Bloom filter representing their private set, and then exchange these Bloom filters with the other parties. The parties can then compute the intersection of the sets by performing a bitwise AND operation on the Bloom filters, without revealing the actual elements in the sets.

2 Notation and Definition

Throughout this paper, all logarithms are in base 2 unless explicitly specified otherwise.

Firstly, we define the main object of interest in this paper, the AMQ-filter.

Definition 2.1. *We say that a data structure* $\mathbf{D}$ *is a static AMQ-Filter with false positive rate* ϵ *if it supports the following operations:*

- init*(S): Called exactly once at the beginning of the algorithm. Adds all elements of the set S into the data structure. This part of the algorithm can be randomized.*
- query*(x): Returns either* 0 *or* 1*.*
  - *If x was inserted in the* init *operation, the query must return* 1*.*

  - *Otherwise, it must return* 0 *with probability at least* $1 - \epsilon$*. The probability is taken over the randomness of the* init *operation.*

We can similarly define a dynamic AMQ-Filter, where insertions can be made at any time.

Definition 2.2. *We say that a data structure* $\mathbf{D}$ *is a dynamic AMQ-Filter with false positive rate* ϵ *if it supports the following operations:*

- insert*(*x*): Adds* x *to the data structure, which forces future queries of* x *to return* 1 *(see below).*
- query*(*x*): Returns either* 0 *or* 1*.*
  - *If* x *was previously inserted by the insert operation, the query must return* 1*.*
  - *Otherwise, it must return* 0 *with probability at least* $1 - \epsilon$*. The probability is taken over the randomness used in the algorithm.*

To model adaptive adversaries, we consider the following adversary game:

Definition 2.3 ((n,t)-Adversary Game)**.** *We define the* (n,t)*-adversary game as follows:*

- *The adversary first picks a set* $S = \{s_1, s_2, \ldots, s_n\}$*, which will be inserted into the filter.*
- *The algorithm (possibly randomized) constructs the AMQ-Filter. The internal randomness of the construction algorithm is not known to the adversary.*
- *The adversary makes up to* t *queries,* $x_1, x_2, \ldots, x_t$ *(where they are allowed to select the queries based on the outputs they see from the filter).*
- *The adversary outputs* $x \notin \{x_1, x_2, \ldots, x_t\}$*. If* x *is a false positive, the adversary wins.*

Our goal is to analyze AMQ-filters for which no adversary can win the game described above with more than ϵ probability while using as little memory as possible. More formally,

Definition 2.4 ((n,t,ϵ)-resilient)**.** *We say that an AMQ-filter is* (n,t,ϵ)*-resilient if no adversary can win the* (n,t)*-adversary game with more than* ϵ *probability.*

When we are dealing with bounded-time adversaries, we can sometimes drop the restriction on the number of queries. More precisely,

Definition 2.5 ((n,ϵ)-resilient)**.** *We say that an AMQ-filter is* (n,ϵ)*-resilient if it is* (n,t,ϵ)*-resilient for any* $t = O(\mathsf{poly}(n))$*.*

3 Bounds on Memory Usage

Here, we investigate the lower and upper bounds on the memory usage of adversarially resilient AMQ-filters.

3.1 Non-adversarial Setting

As a warmup, we survey the basic bounds on memory usage of AMQ-filters in the classical non-adversarial setting.

Firstly, there is a well-known entropy lower bound on the memory required:

Lemma 3.1. *[CFG+78] Suppose the universe has size* $u \gg n$*. Let* $\epsilon > 0$*. Any AMQ-Filter with false positive rate at most* ϵ *requires at least* $n \log \frac{1}{\epsilon}$ *bits of memory.*

With this, we can define the minimal error of an AMQ-filter.

Definition 3.2 (Minimal error)**.** *Let* $\mathbf{D}$ *be an AMQ-filter using* m *bits of memory and stores* n *elements. Then,* $\epsilon_0 := 2^{-\frac{m}{n}}$ *is defined as the minimal error of* $\mathbf{D}$*.*

Note that Lemma 3.1 implies that any AMQ-filter using a memory of m bits must have false positive rate at least ϵ_0.

Traditional Bloom filters achieves a memory usage of $\log(e) \log \frac{1}{\epsilon} \approx 1.44n \log \frac{1}{\epsilon}$, which is close to the optimal space usage. A lot of work has been done in closing the gap between these bounds. In particular, the following question is interesting: *What is the minimum constant* c *such that an AMQ-filter using* $m \leq cn \log \frac{1}{\epsilon}$ *bits exists?*

For dynamic AMQ-Filters, it is known that the entropy lower bound is **not** achievable, due to a result by [LP10].

Lemma 3.3. *[LP10] Suppose the universe has size* $u \gg n$*. Let* $\epsilon > 0$ *be a fixed constant. Then, there exists a constant* $C(\epsilon) > 1$ *such that any dynamic AMQ-Filter with false positive rate at most* ϵ *requires at least* $C(\epsilon)n \log \frac{1}{\epsilon}$ *bits of memory.*

However, for static AMQ-Filters, the entropy bound of $c = 1$ is achievable (up to lower order terms):

Lemma 3.4. *[DP08, Por08] Given any* $\epsilon > 0$*, there exists a static AMQ-Filter that uses only* $(1 + o(1))n \log \frac{1}{\epsilon}$ *bits of memory.*

Proof Sketch for Weaker Version. The following proof imposes an additive $n \log(e)$ factor, which can be removed with some work.

Given the set S, we can find a minimal perfect hash function h for S with representation using $n \log(e) + o(n)$ bits ([BPZ07]). Let g be another hash function that maps elements of U into $\log \frac{1}{\epsilon}$-bit integers. Store an array A of n elements, each of length $\log \frac{1}{\epsilon}$ bits, where $A[h(x)] := g(x)$ for all $x \in S$. $y \notin S$ is a false positive if and only if $g(y) = A[h(y)]$. □

While Lemma 3.4 is essentially optimal, it does not consider bloom filters that are secure under adversarial queries (as defined in Definition 2.3). Our goal is to determine if such a result is possible for (n, ϵ)-resilient AMQ-filters.

3.2 Adversarial Setting

Now, we discuss the bounds on memory usage of AMQ-filters under the adversarial setting described in Definition 2.3. We will focus on static AMQ-filters, i.e. filters where all insertions happen once simultaneously before queries.

3.2.1 Bounded-time Adversaries

Firstly, suppose the adversary only has bounded computational power. More precisely, given a security parameter λ, we assume that the adversary can only run in $\mathsf{poly}(\lambda)$ time. We also make no specific restrictions on t, the number of queries the adversary is allowed to make in Definition 2.3. Note that we automatically have $t = O(\mathsf{poly}(\lambda))$ since a bounded time adversary cannot make more queries than its runtime.

For technical reasons, we focus on AMQ-filters on a sufficiently large universe and uses less memory than required to store all elements exactly (for which the problem is trivial). More formally,

Definition 3.5 (Non-trivial filters)**.** *[NY14] Let* $\mathbf{D}$ *be an AMQ-filter on a universe of size* u *that uses* m *bits of memory and has minimal error* ϵ_0*. We say that* $\mathbf{D}$ *is nontrivial if* $u = \omega\left(\frac{m}{\epsilon_0^2}\right)$ *and there exists a constant* $c > 0$ *such that* $\epsilon_0 > n^{-c}$*.*

It turns out that (non-trivial) (n, ϵ)-resilient AMQ-filters can only exist if one-way functions exist. In fact, here is a stronger claim:

Lemma 3.6. *[NY14] Let $\mathbf{D}$ be any non-trivial AMQ-filter on n elements using m bits of memory and let $\epsilon_0 = 2^{-\frac{m}{n}}$ be the minimal error of $\mathbf{D}$. If one-way functions do not exist, then for any $\epsilon < 1$, $\mathbf{D}$ is not (n, t, ϵ)-resilient for $t = O(m/\epsilon_0^2)$.*

Corollary 3.7. *If one-way functions do not exist, then non-trivial (n, ϵ)-resilient AMQ-filters do not exist.*

On the other hand, if one-way functions exist, it turns out that we can construct (n, ϵ)-resilient AMQ-filters from any AMQ-filter with false positive rate ϵ.

Lemma 3.8. *[NY14] Let $\mathbf{D}$ be any AMQ-filter on n elements using m bits of memory with false positive rate ϵ. If one-way functions exist, then there exists a negligible function $\mathsf{negl}(\cdot)$ such that for security parameter λ, there is a $(n, \epsilon + \mathsf{negl}(\lambda))$-resilient AMQ-filter using $m + \lambda$ bits of memory.*

Proof Sketch. If one-way functions exist, then pseudorandom permutations (PRP) exists. We will first apply the PRP to each element, and then insert/query them as usual. It can be shown using a hybrid argument that this is adversarially resilient. □

Combined with Lemma 3.4, we obtain

Corollary 3.9. *Assume that one-way functions exist. Given any $\epsilon > 0$, there exists a static AMQ-Filter that is (n, ϵ)-resilient using only $(1 + o(1))n \log \frac{1}{\epsilon}$ bits.*

3.2.2 Unbounded-time Adversaries

For unbounded adversaries, the problem is more interesting, because the optimum constant factor c is unclear and also depends on the value of t, the number of queries the adversary is allowed to make. In particular, if $t \gg n$, the memory required is $\omega\left(n \log \frac{1}{\epsilon}\right)$. The following lemma from [NY14] gives a lower bound on the memory required based on t:

Lemma 3.10. *[NY14] Let $\mathbf{D}$ be a nontrivial AMQ-filter storing n elements using m bits of memory. Let ϵ_0 be the minimal error of $\mathbf{D}$. Then, for any constant $\epsilon < 1$, there exists a $t = O\left(\frac{m}{\epsilon_0^2}\right)$ for which $\mathbf{D}$ is not (n, t, ϵ)-resilient.*

In particular, to achieve $m = O\left(n \log \frac{1}{\epsilon}\right)$ (which is the regime we focus on here), we cannot have $t = \omega(n)$. Hence, we focus on the regime $t = O(n)$.

[NY14] suggested a construction that uses $O\left(n \log \frac{1}{\epsilon} + t\right)$ bits. However, their construction requires at least $2n \log \frac{1}{\epsilon}$ space. We show that we can use perfect hashing to make the constant factor close to 1 for $t = o(n)$ (and approaches 1 when $\epsilon \to 0$).

Lemma 3.11. *For any $n, t \in \mathbb{N}$ and all $\epsilon \in \left(0, \frac{1}{2}\right)$, there exists a (n, t, ϵ)-resilient AMQ-filter that uses $(1 + o(1))n \log \frac{1}{\epsilon} + n \log_2(e) + O(t)$ bits of memory.*

Before we go into the construction, we need to introduce the notion of almost k-wise independent hash functions (which was also used in [NY14]'s construction). First, we recall the notion of k-wise independent hash functions.

Definition 3.12 (k-wise independent hash family)**.** *A hash family $\mathcal{H} = \{h : U \to V\}$ is said to be k-wise independent if*

$$\Pr_{h \leftarrow \mathcal{H}} \left[h(\vec{x}) = \vec{y} \right] = \frac{1}{|V|^k}$$

for any vectors $\vec{x} \in U^k, \vec{y} \in V^k$, where h is applied component-wise to the vector U^k.

It would be nice if we could use k-wise independent hash families in our construction. Unfortunately, they require $O(k \log |U|)$ bits, which is even more than the memory required to store S exactly. Instead, we need to rely on *adaptively almost-k-wise independent* hash families:

Definition 3.13. *A hash family $\mathcal{H} = \{h : U \to V\}$ is said to be adaptively almost-k-wise independent if for any adaptive distinguisher that issues a sequence of k adaptive queries on a randomly selected hash function $h \leftarrow H$, the advantage of distinguishing between h and a k-wise independent function $U \to V$ is polynomially small in k.*

It turns out that it is possible to construct such hash families using $O(k)$ bits of memory.

Lemma 3.14. *[BHKN21] One can construct an adaptively almost-k-wise independent hash family $\mathcal{H}$ such that any $h \in \mathcal{H}$ satisfies the following two conditions:*

1. *h can be evaluated in constant time,*
2. *h can be stored using $O(k \log |V|)$ bits.*

Now, we can describe the construction for the unbounded case.

Proof Sketch of Lemma 3.11. Firstly, we construct a minimal perfect hash function h for S with representation using $n \log(e) + o(n)$ bits ([BPZ07]). We will use an array A of size n, where each element is $(1 + o(1)) \log \frac{1}{\epsilon}$ bits long. The idea is to store a signature of each element x in position $h(x)$ of the array.

To construct the signature, we first take $l = (1+o(1)) \log \frac{1}{\epsilon}$ one-bit functions $g_1, g_2, \ldots, g_l$ where each g_i is selected independently from an adaptively almost-k-wise independent hash family $\mathcal{H}$ with $|V| = \{0, 1\}$ and $k = 2t/ \log \frac{1}{\epsilon}$. Let g be the concatenation of $g_1, g_2, \ldots, g_l$. The signature of each element $x \in \mathcal{U}$ is $g(x) = g_1(x) g_2(x) \ldots g_l(x)$. Hence, $A[h(x)] = g(x)$ for all $x \in S$. The total memory usage to represent the g_is is $O(kl) = O(t)$ by Lemma 3.14. Hence, the total memory usage is $(1 + o(1)) n \log \frac{1}{\epsilon} + n \log_2(e) + O(t)$ bits.

When answering queries, on each query y we would compare $g(y)$ to the value of $A[h(y)]$. However, we utilize the following trick (also used by [NY14]) to decrease the independence necessary for the hash family $\mathcal{H}$: we check whether $g(y)$ equals $A[h(y)]$ by comparing bit by bit (halting whenever we discover two corresponding bits that aren't equal). Moreover, we ensure that we are using each hash function g_i almost equally many times by marking the last bit that was compared, and for the next comparison we start from the next bit in cyclic order.

It remains to show that this construction is (n, t, ϵ)-resilient. The analysis is essentially as the one described in [NY14]. The main idea is that if $x \neq y$, on average we expect to make 2 bit comparisons before concluding whether $g(x) \neq g(y)$. Hence, over all queries, we expect to make $O(t)$ bit comparisons across the l one-bit functions (with high probability via Chernoff bound). Since we distribute the comparisons equally over those functions, each function is used in at most $t/l \leq k$ comparisons, and so almost-k-wise independence is sufficient. □

Note that Lemma 3.11 matches the entropy lower bound up to sublinear terms as $\epsilon \to 0$ and $t = o(n)$.

4 Conclusion

We showed that under the security game defined in [NY14], we can construct near-optimal adversarial-resilient AMQ-filters under the right settings. There are some more open questions that can be asked:

- Can we obtain tight bounds for memory usage of adversarial resilient AMQ-filters against unbounded adversaries where $t = \Theta(n)$? Our bounds only work for $t = o(n)$, and it is likely that the optimal bound for $t = \Theta(n)$ would depend on the constant factor in $t = \Theta(n)$.
- What bounds can we obtain for adversarial resilient dynamic AMQ-filters (i.e. insertions are allowed after queries). This version is harder because (among other reasons) it is known that $(1 + o(1))n \log \frac{1}{\epsilon}$ bits of memory is **not** sufficient even for the non-adversarial case. The strategy of using perfect hashing also no longer works since the set S is not known in advance.
- We can also consider other versions of the security game. We felt that this version of the security game is pretty natural, but there are other stronger variants (e.g. [NO22]). Can we achieve similar optimal bounds for these variants?

Acknowledgements

The authors would like to thank Professor Henry Corrigan-Gibbs and Professor Yael Kalai, as well as teaching assistants Alexandra Henzinger and Kyle Hogan for their comments on the project and help and support throughout the semester.

References

[BFCG$^+$18] Michael A. Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. Bloom filters, adaptivity, and the dictionary problem. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 182–193, 2018.

[BHKN21] Itay Berman, Iftach Haitner, Ilan Komargodski, and Moni Naor. Hardness-preserving reductions via cuckoo hashing, 2021.

[BHP22] Ioana O. Bercea, Jakob Bæk Tejs Houen, and Rasmus Pagh. Daisy bloom filters, 2022.

[BM03] Andrei Broder and Michael Mitzenmacher. Survey: Network applications of bloom filters: A survey. *Internet Mathematics*, 1, 11 2003.

[BPZ07] Fabiano C. Botelho, Rasmus Pagh, and Nivio Ziviani. Simple and space-efficient minimal perfect hash functions. In *Proceedings of the 10th International Conference on Algorithms and Data Structures*, WADS'07, page 139–150, Berlin, Heidelberg, 2007. Springer-Verlag.

[CFG$^+$78] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, page 59–65, New York, NY, USA, 1978. Association for Computing Machinery.

[CPS19] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic data structures in adversarial environments. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1317–1334, New York, NY, USA, 2019. Association for Computing Machinery.

[DP08] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership, 2008.

[DW21] Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor, 2021.

[FPUV22] Mia Filic, Kenneth G. Paterson, Anupama Unnikrishnan, and Fernando Virdia. Adversarial correctness and privacy for probabilistic data structures. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1037–1050, New York, NY, USA, 2022. Association for Computing Machinery.

[GKL15] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. The power of evil choices in bloom filters. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 101–112, 2015.

[GL20] Thomas Mueller Graf and Daniel Lemire. Xor filters. *ACM Journal of Experimental Algorithmics*, 25:1–16, mar 2020.

[HHCG$^+$22] Alexandra Henzinger, Matthew M. Hong, Henry Corrigan-Gibbs, Sarah Meiklejohn, and Vinod Vaikuntanathan. One server for the price of two: Simple and fast single-server private information retrieval. Cryptology ePrint Archive, Paper 2022/949, 2022.

[HT01] Torben Hagerup and Torsten Tholey. Efficient minimal perfect hashing in nearly minimal space. volume 2010, pages 317–326, 02 2001.

[LP10] Shachar Lovett and Ely Porat. A lower bound for dynamic approximate membership data structures. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, page 797–804, USA, 2010. IEEE Computer Society.

[NO22] Moni Naor and Noa Oved. Bet-or-pass: Adversarially robust bloom filters. Cryptology ePrint Archive, Paper 2022/1292, 2022.

[NY13] Moni Naor and Eylon Yogev. Sliding bloom filters, 2013.

[NY14] Moni Naor and Eylon Yogev. Bloom filters in adversarial environments, 2014.

[Por08] Ely Porat. An optimal bloom filter replacement based on matrix solving, 2008.