The proceedings version of this paper appears at CCS '19. This is the full version.

Probabilistic Data Structures in Adversarial Environments

David Clayton, Christopher Patton, and Thomas Shrimpton

Florida Institute for Cybersecurity Research
Computer and Information Science and Engineering
University of Florida

{davidclayton,cjpatton,teshrim}@ufl.edu

Abstract

Probabilistic data structures use space-efficient representations of data in order to (approximately) respond to queries about the data. Traditionally, these structures are accompanied by probabilistic bounds on query-response errors. These bounds implicitly assume benign attack models, in which the data and the queries are chosen non-adaptively, and independent of the randomness used to construct the representation. Yet probabilistic data structures are increasingly used in settings where these assumptions may be violated.

This work provides a provable-security treatment of probabilistic data structures in adversarial environments. We give a syntax that captures a wide variety of in-use structures, and our security notions support derivation of error bounds in the presence of powerful attacks.

We use our formalisms to analyze Bloom filters, counting (Bloom) filters and count-min sketch data structures. For the traditional version of these, our security findings are largely negative; however, we show that simple embellishments (e.g., using salts or secret keys) yields structures that provide provable security, and with little overhead.

Contents

1 Introduction **3**
1.1 Related work . . . 6

2 Syntax **7**
2.1 Preliminaries . . . 7
2.2 Data structures . . . 8

3 Notions of Adversarial Correctness **8**

4 Bloom Filters **10**
4.1 Insecurity of unsalted BFs . . . 11
4.2 Salted BFs in the (im)mutable setting . . . 13
4.3 Keyed BFs . . . 19
4.4 ℓ-thresholded BFs . . . 22
4.5 Discussion . . . 24

5 Counting Filters **25**
5.1 Insecurity of public counting filters . . . 26
5.2 Security of private, ℓ-thresholded counting filters . . . 27

6 Count-Min Sketches **27**
6.1 Insecurity of public sketches . . . 28
6.2 Private, ℓ-thresholded sketches . . . 28
6.3 Discussion . . . 30

A Proof of Theorem 5 **33**

1 Introduction

Probabilistic data structures, which use space-efficient representations of data to provide (approximately correct) answers to queries about the data, find myriad uses in modern communication, storage, and computational systems. The Bloom filter [4], for example, is ubiquitous in distributed computing, including web caches (e.g., Squid) and hash tables (e.g., BigTable and Hadoop), resource and packet routing, and network measurement. (We refer the reader to the surveys [5, 28] for a comprehensive list of applications.)

The traditional approach to analyzing the correctness of a data structure is to assume that all inputs, and all queries, are independent of any internal randomness used to construct it. But as highlighted by Naor and Yogev (CRYPTO '15 [24]), there are important use-cases in which the inputs and queries may be chosen *adversarially* and *adaptively*, based on partial information and prior observations about the data structure. Attacks of this sort can be used to disrupt or reduce the availability of real systems [9, 18, 22].

Naor and Yogev (hereafter NY) formalized a notion of adversarial correctness for Bloom-filter-like structures. Recall that a Bloom filter encodes a set $\mathcal{S}$ into a length-m array of bits (initially all zeros), where m is much less than the number of bits needed to store $\mathcal{S}$ in full. Elements $x \in \mathcal{S}$ are encoded by computing multiple hash values $h_1(x), h_2(x), \ldots, h_k(x) \in [m]$, then setting the indicated array positions to 1. This bit-array representation of $\mathcal{S}$ allows for set membership queries, i.e., "is $x \in \mathcal{S}$?", by hashing x and responding positively iff all of the indicated positions hold a 1-bit. False-negative respones are not possible, but false-positive responses are. Classical results relate $|\mathcal{S}|, m, k$ to the probability of false-positive query responses [5, 20], where the probability is over the sampling of the hash functions. (These are usually modeled as independent random functions.) Crucially, these results assume that $\mathcal{S}$ and the $h_1, \ldots, h_k$ are independent of each other. Said another way, even if $\mathcal{S}$ is adversarially chosen, this choice cannot depend on particular hash functions that are used to produce the Bloom filter and compute the query responses. The conceptual innovation of NY was to remove this assumption an explore the consequences upon the probability of Bloom filter query-response errors. In particular, NY allowed the adversary to specify a (fixed) set $\mathcal{S}$ that may depend on the hash functions, and then attempt to induce errors via set-membership queries.

We expand upon NY in several ways, providing syntax and security notions that allow analysis of a large class of data structures (not only Bloom filters), in settings where the data may not be a set and may change over time, and where the structure's representation of the data may (or may not) be publicly visible.

Beyond sets and Bloom filters. Our first significant extension of NY is that our attack model allows the adversary to adaptively *update* the collection $\mathcal{S}$ during its attack. This captures settings in which the data to be represented may change over time, e.g., streaming data applications. Many data structures are designed for such settings — the counting filter [15], count-min sketch [8], cuckoo filter [14], and stable Bloom filter [11], to name a few — by providing updatable, or *mutable*, representations. Our syntactic formalization of data structures captures this reality.

Next, while the Bloom filter was designed to represent data collections $\mathcal{S}$ that are sets, a data stream (for example) is more accurately modeled as a multiset. Here one is often interested in information about frequency, e.g., "how many times does x appear in $\mathcal{S}$?" Thus, in addition to admiting mutable representations, our formalization of data structures allows for rich query spaces. Specifically, we define a data structure to be a triple of algorithms $(\textsc{Rep}, \textsc{Qry}, \textsc{Up})$ denoting the *representation*, *query-evaluation*, and *update* algorithms, respectively. Associated to the data structure is a set of supported query *functions* $\mathcal{Q}$, and a set $\mathcal{U}$ of allowed update functions. For reasons we will elucidate in a moment, all three algorithms take a key K as input, and both Rep and Up may be randomized.

The combination of mutability and rich query spaces has significant implications for security. Consider the counting Bloom filter structure [15] (we refer to this simply as a "counting filter" in the remainder). It is similar to a Bloom filter, but instead of a bit array, a counting filter represents an updatable multiset $\mathcal{S}$ as an array of m integers; these serve as counters. To add x to $\mathcal{S}$, hash values $h_1(x), \ldots, h_k(x) \in [m]$ are computed, and the indicated counters are incremented. Decrementing the counters implements *deletion* of an occurrence of x from $\mathcal{S}$. Like a Bloom filter, a counting filter provide approximately correct answers to set-membership queries,[1] where a query about x results in a positive response iff all of the hash-indicated counters are at least one. Unlike a Bloom filter, this structure admits both false-positive *and* false-negative responses. In particular, if the representation is updated by "removing" an element y that does not appear

[1] Indeed, they were initially introduced to support deletions from a *set*, without having to rebuild the representation, as one would for a Bloom filter.

in the underlying $\mathcal{S}$, one or more of the counters associated to x may be decremented, potentially causing x to become a false negative.

Both the Bloom and counting filters have binary query responses, making the notion of response error easy to define: the response is either correct or incorrect. But practically important structures, like the count-min sketch, admit frequency-of-element queries, which have integer responses. What constitutes an error is less clear for such queries. Even in the traditional analyses (i.e., non-adaptive attacks) one is guaranteed only that responses will be "close" to correct, with probability "close" to one. We therefore parameterize our security experiments with a specifiable *error function* δ. If the correct response to an adversarial query is a and the data structure responds with a', the experiments award the adversary with error weight $\delta(a, a') \geq 0$. Our experiments are additionally parameterized by an *error capacity* $r \geq 0$, and the adversary is considered to "win" if the total cost of the errors it induces is greater than this value. As it turns out, even calculating this total cost is not straightforward in our setting: one must determine whether or not the cost of a given error should be carried across (adaptive, adversarial) updates to $\mathcal{S}$ and its representation.

Public vs. Private Representations. We define two experiments, one in which representations are shown to the adversary, and one in which they are not. In the ERR-Pub game, the adversary is given a representation-oracle **Rep** that, on input a collection $\mathcal{S}$, returns the resulting representation $\text{Rep}_K(\mathcal{S})$. Note that the key K (which may be the empty string, to capture unkeyed structures) is fixed across all calls; however, per-representation randomness (e.g. salts) may be encoded by the representation. The adversary is permitted to (adaptively) update any established representation via an update-oracle **Up** and, at any time, it may query a representation via a query-oracle **Qry**. The adversary is given credit (determined by δ) for each **Qry**-query that results in an error. The ERR-Priv game is defined in much the same way, except that representations are not shown to the adversary unless it explicitly asks for them to be revealed.

There are many applications in which the adversary would not have unfettered access to the structure [18], and for which the assumption of a private data structure is most fitting. However, we do not want to rule out the possibility that the adversary may eventually learn the contents of some data structures, which are likely to have looser access controls than long-term private keys or similar critical data. When possible, we would also like to account for cases where the adversary may be able to freely view the data structures as well, such as in the cases where Bloom filters are used for distributed computations. This corresponds to the ERR-Pub case, though this is a much stronger notion which we will find is not always achievable.

Case studies and our findings. We exercise our syntax and notions by analyzing three important, real-world data structures: Bloom filters [4] (Section 4), counting filters [15] (Section 5), and count min-sketches [8] (Section 6). Our studies examine the basic versions of each, as well as variants that may take a key or a per-representation random salt, and variants that incorporate alternate measures of when the structure is full. Each of the basic structures supports different queries and update operations; taken together, they provide interesting coverage of the structure/attack-model landscape.

We find that *none of the (basic) structures meets either of our security notions.* In particular, if the data being represented, the updates, and the queries all may depend on the choice of hash function, then each of these structures is susceptible to a class of attacks we call *target-set coverage attacks* (described in Section 4.1). These are closely related to *pollution attacks* against standard Bloom filters [18], which we will discuss in some detail.

On the positive side, we show how these structures can be modified in ways that are conceptually straightforward and intuitive in order to prove security. Our results are summarized in Figure 1.

Bloom filters, our in-depth study. Due to their wide-spread and varied use (and following NY), we begin with a deep look at Bloom filters. It is well-known that standard Bloom filters do not perform well in adversarial settings [24, 18]; we first corroborate these findings via an explicit ERR-Priv attack (Section 4.1). We then consider the security of several variants of the basic Bloom filter for which we can derive correctness (i.e., security) bounds. The first idea is to generate a short, random *salt*, which we prepend to the input of the hash. Thus, instead of computing $h_i(x)$ for each $1 \leq i \leq k$ we compute $h_i(Z \,\|\, x)$, where Z is a short (say, 128-bit) string chosen by the representation algorithm. This leads to our first positive result, for this *salted* Bloom filter, in the public-representation setting when attacks treat representations as immutable (i.e., updates are forbidden); this is Theorem 1. Following the traditional approach [5], we model the hash functions as random oracles (ROM) [1]. Our security argument must account for any hash-exploiting

Structure	Results
Bloom filter (Fig. 5, Fig. 11)	Basic structure insecure. *Immutable case:* structure can be secured with per-representation salt. *Mutable case:* structure additionally require a secret key or keeping representations private, and benefit from thresholding (defining 'fullness' by Hamming weight rather than number of elements).
Counting filter (Fig. 14)	Basic structure insecure. Security can be achieved by combining a per-representation salt, thresholding, and private representations.
Count-min sketch (Fig. 15)	Basic structure insecure. Security can be achieved by combining a per-representation salt, thresholding, and private representations.

Figure 1: A high-level summary of our results.

precomputation performed by the adversary via the random oracle. This leads to fairly weak bounds, which means that larger filters must be used to achieve a reasonable correctness upper bound (Figure 7). On the other hand, we find far better bounds, even in the mutable setting, if the representation is kept private (Theorem 2).

We derive a similarly good bound for *keyed* Bloom filters, which use a secretly-keyed pseudorandom function (PRF) instead of a hash function (in addition to salts). This result is in the mutable *and* public-representation setting (Theorem 3), the strongest attack model we formalize.

Normally, Bloom filters are considered to be "full" when some pre-determined set size, or *capacity*, is reached. Indeed, Bloom filter parameters are generally chosen as a function of this maximum capacity [20]. We explore an alternative definition of fullness, whereby the filter is deemed full once the Hamming weight of the filter (i.e., the number of 1s) crosses a pre-determined *threshold*. While the two definitions are more or less interchangeable in the non-adaptive, traditional setting, we show that this alternative definition has substantial analytical value in adversarial environments. In Theorem 4, we reconsider the security of salted BFs in the mutable, private setting, and exhibit substantially tighter bounds. In particular, we find that as long as salts are reasonably large, we can use a 900-byte filter to store 100 objects, while incurring a less than 10% chance of a single false positive over the course of 2^{32} queries, and a less than one-in-a-million chance of 5 or more false positives (see Figure 13). This holds even if the adversary is allowed to completely control the filter's construction.

Counting filters. Following the deep dive into Bloom filters in Section 4, we then consider counting filters, which allow for both insertion and deletion operations while maintaining a compact representation by using counters in place of single bits. Besides this, the construction is identical to that of a standard Bloom filter. Despite the similarities, we find that counting filters are not secure in the public-representation setting, even if we add a salt or use a PRF in place of the hash function. The fact that the adversary can see exactly which filters are incremented or decremented with each update, along with the fact that updates can be trivially reversed (deletion undoes insertion and vice versa) allows the adversary to mount attacks by trial and error even if it lacks the ability to predict in advance where an element will be sent by the hash functions. However, we are able to derive a good correctness in the mutable/private setting (Theorem 5), using a per-representation salt and a notion of "fullness" similar to threshold Bloom filters.

Count min-sketches. Finally, we also consider the case of the count min-sketch (CMS). These structures provide a compact representation of a multiset rather than a set, allowing queries for approximate frequency of an element in the multiset. While a count-min sketch hashes in much the same way as a Bloom or counting filter, it uses a 2D array of non-negative integer counters rather than a linear array, taking the minimum counter value over all arrays to answer queries. We find that due to their structural similarities and the similar update operations allowed for the structures, CMS and counting filters exhibit similar security properties. Again we see that these structures are not secure in the public-representation case, but find a bound in the private-representation case when salts and thresholds are used (Theorem 6).

Structure	Data Objects	Supported Queries	Supported Updates	Parameters
Bloom filter (Fig. 5)	Sets, $\mathcal{S} \subseteq \{0,1\}^*$	$\mathsf{qry}_x(\mathcal{S}) = [x \in \mathcal{S}]$	$\mathsf{up}_x(\mathcal{S}) = \mathcal{S} \cup \{x\}$	n, max $\lvert\mathcal{S}\rvert$ k, # hash functions m, array size (bits)
ℓ-thresholded Bloom filter (Fig. 11)	Sets, $\mathcal{S} \subseteq \{0,1\}^*$	$\mathsf{qry}_x(\mathcal{S}) = [x \in \mathcal{S}]$	$\mathsf{up}_x(\mathcal{S}) = \mathcal{S} \cup \{x\}$	ℓ, max # 1s in array k, # hash functions m, array size (bits)
count-min sketch (Fig. 15)	Multisets, $\mathcal{S} \in \mathrm{Func}(\{0,1\}^*, \mathbb{N})$	$\mathsf{qry}_x(\mathcal{S}) = \mathcal{S}(x)$	$\mathsf{up}_{x,0}(\mathcal{S})(x) = \mathcal{S}(x) + 1$ $\mathsf{up}_{x,1}(\mathcal{S})(x) = \mathcal{S}(x) - 1$ $\mathsf{up}_{x,b}(\mathcal{S})(y) = \mathcal{S}(y)$ for $x \neq y$	ℓ, max # nonzero counters k, # hash functions and arrays m, # counters per array
counting filter (Fig. 14)	Multisets, $\mathcal{S} \in \mathrm{Func}(\{0,1\}^*, \mathbb{N})$	$\mathsf{qry}_x(\mathcal{S}) = [\mathcal{S}(x) > 0]$	$\mathsf{up}_{x,1}(\mathcal{S})(x) = \mathcal{S}(x) + 1$ $\mathsf{up}_{x,-1}(\mathcal{S})(x) = \mathcal{S}(x) - 1$ $\mathsf{up}_{x,b}(\mathcal{S})(y) = \mathcal{S}(y)$ for $x \neq y$	ℓ, max # non-zero counters k, # hash functions m, # counters per array

Figure 2: The data structures that we consider. Each data structure yields a space-efficient representation of its input data object and, in the presence of non-adaptive attacks, provides approximately correct responses to the supported queries. For counting filters and count-min sketches, typical implementations prevent updates that would cause $\mathcal{S}(x) - 1 < 0$. Count-min sketch supports additional queries (e.g. range queries) that we do not consider.

Recommendations. We find that the keeping the data structure private is often essential to guaranteeing security. In many settings this is not an issue, for example if the structure is only directly accessed by some specific trusted source. For counting filters and count min-sketches, the presence of both insertion and deletion operations gives an adaptive adversary enough power that security is difficult to provide. However, *with per-representation salts and the thresholding procedure we describe, security can be guaranteed in the private setting.*

For Bloom filters, the situation is somewhat better. Even in the public setting, *a salt alone can suffice to provide security for Bloom filters*, though this may also require a substantial increase in the size of the filter. For security purposes, a private key is also useful, forcing the adversary's attacks to be 'online' rather than relying on offline hash computations. However, using a private key may not be possible in all applications, since anyone making a query to the filter must have access to the key.

Future work. The focus of this work is the data structures themselves. Even so, we were only able to consider a handful of (important, real-world) examples. We hope that future work will apply our formalisms to other probabilistic data structures, such as the previously-mentioned cuckoo filter and stable Bloom filter.

Going into a different direction, future work should also address how adversarial correctness impacts high-level protocols that use these probabilistic data structures. A good example is content-distribution networks [6], where many servers propagate representations of their local cache to their neighbors. (In Section 4 we will touch briefly on the real-world attacks that are possible in this setting.) The Bloom filter family alone has a wide range of practical applications, for example in large database query processing [5], routing algorithms for peer-to-peer networks [26], protocols for establishing linkages between medical-record databases [27], fair routing of TCP packets [16], Bitcoin wallet synchronization [19], and deep packet inspection [28]. Recently, Bloom filters were proposed as a means of efficient certificate-revocation list (CRL) distribution [21], a crucial component of public-key infrastructures. Analyzing higher-level primitives or protocols will require establishing appropriate syntax and security notions for those, too; hence we leave this for future work.

Another interesting direction is to consider what information data structures leak via their public representations. A large variety of data structures with interesting privacy properties have been proposed. For example, variants of Bloom filters that ensure privacy of the *query* have been studied [3, 25]. These prior works leave open the security of more conventional data structures, like those studied in this paper.

1.1 Related work

Comparison with Naor-Yogev. As previously noted, NY [24] were the first to formalize adversarial correctness of Bloom filters. Our work extends theirs significantly in several directions. First, we consider abstract data structures, rather than only set-membership structures. Even with respect to the specific case of correctness for set-membership structures, our work offers several advantages as compared to the

NY treatment. One, our syntax distinguishes between the (secret) key and the public portion of a data structure, an important distinction that is missing in their work. Two, the NY definition of correctness allows the adversary to make several queries, some of which may produce incorrect results; the attacker then succeeds if it outputs a *fresh* query that causes an error. This separation seems arbitrary, and we propose instead a parameterized definition in which the attacker succeeds if it can cause a certain number of (distinct) errors during its entire execution. Three, Naor and Yogev analyze the correctness of a new Bloom filter variant of their own design. In contrast, we are mainly interested in analyzing existing, real-world constructions to understand their security.

Other related works. There is a long tradition in computer science of designing structures that concisely (but probabilistically) represent data so as to support some set of queries [7, 8, 12, 13, 17, 23]. Each of these structures has its own interesting security characterisitcs.

Perhaps the earliest published attack on the correctness of a data structure was due to Lipton and Naughton [22], who showed that timing analysis of record insertion in a hash table allows an adversary to adaptively choose elements so as to increase look-up time, effectively degrading a service's performance. Crosby and Wallach [9] exploited hash collisions to increase the average URL load time in Squid, a web proxy used for caching content in order to reduce network bandwidth. More recently, Gerbet et al. [18] described *pollution attacks* on Bloom filters, whereby an adversary inserts a number of adaptively-chosen elements with the goal of forcing a high false-positive rate. Although some of their attacks exploit weak (i.e., non-cryptographic) hash functions (as do [9]), their methodology is effective even for good choices of hash functions. They suggest revised parameter choices for Bloom filters (i.e., filter length and number of hashes) in order to cope with their attacks, as well as the use of keyed hash functions. With our more general attack model, however, we will see that a secret key alone does not guarantee correctness.

2 Syntax

2.1 Preliminaries

Let $x \twoheadleftarrow \mathcal{X}$ denote sampling x from a set $\mathcal{X}$ according to the distribution associated with $\mathcal{X}$; if $\mathcal{X}$ is finite and the distribution is unspecified, then it is uniform. Let $[i..j]$ denote the set of integers $\{i, \ldots, j\}$; if $i > j$, then define $[i..j] = \emptyset$. For all m let $[m] = [1..m]$.

Bitstring operations. Let $\{0,1\}^*$ denote the set of bitstrings and let ε denote the empty string. Let $X \,\|\, Y$ denote the concatenation of bitstrings X and Y. For all $m \geq 0$ define B_m as the following function: for all $\boldsymbol{x} \in [m]^*$ let $B_m(\boldsymbol{x}) = X_1 X_2 \cdots X_m \in \{0,1\}^m$, where $X_v = 1$ if and only if $\boldsymbol{x}_i = v$ for some $i \in [|\boldsymbol{x}|]$. We call $B_m(\boldsymbol{x})$ the *bitmap* of $\boldsymbol{x}$. Let X and Y be equal-length bitstrings. We write $X \vee Y$ for their bitwise-OR, $X \wedge Y$ for their bitwise-AND, and $X \oplus Y$ for their bitwise-XOR. Let $\neg X = 1^{|X|} \oplus X$ (bitwise-NOT), and let $w(X)$ denote the Hamming weight of (i.e., the number of 1s in) X. For an array $\boldsymbol{M}$ of integers, we analogously define $w'(\boldsymbol{M})$ to be the number of *nonzero* integers in the array. We also let zeroes(m) denote the length m vector of zeros.

Let $\mathrm{Func}(\mathcal{X}, \mathcal{Y})$ denote the set of functions $f : \mathcal{X} \to \mathcal{Y}$. For every function $f : \mathcal{X} \to \mathcal{Y}$, define $\textsc{id}^f : \{\varepsilon\} \times \mathcal{X} \to \mathcal{Y}$ so that $\textsc{id}^f(\varepsilon, x) = f(x)$ for all x in the domain of f. This allows us to use unkeyed hash functions H in situations where, syntactically, a keyed function (e.g., a pseudorandom function) is called for.

Adversaries. Adversaries are randomized algorithms that expect access to one or more oracles defined by the experiment in which it is executed. We say that an adversary is t-time if it halts in t time steps (with respect to some model of computation, which we leave implicit) regardless of its random coins or the responses to its oracle queries. By convention, the adversary's runtime includes the time required to evaluate its oracle queries.

Pseudorandom functions. For sets $\mathcal{X}$ and $\mathcal{Y}$ and a keyspace $\mathcal{K}$, we define a pseudorandom function to be a function $F : \mathcal{K} \times \mathcal{X} \to \mathcal{Y}$. The intent is for the outputs of the function to appear random for a uniformly randomly chosen key, which is formally captured by the game described in Figure 3. We define the advantage of an adversary A to be $\mathbf{Adv}^{\mathrm{prf}}_F(A) = \Pr\left[\mathbf{Exp}^{\mathrm{prf}}_F(A) = 1\right]$, and the function $\mathbf{Adv}^{\mathrm{prf}}_F(t, q)$ to be the maximum advantage of any t-time adversary making q queries to $\mathbf{F}$.

$\underline{\mathbf{Exp}^{\mathrm{prf}}_{F}(A)}$	$\underline{\text{oracle } \mathbf{F}(x)}$:
$b \twoheadleftarrow \{0,1\}$; $K \twoheadleftarrow \mathcal{K}$	if $b = 1$ then return $F_K(x)$
$b' \twoheadleftarrow A^{\mathbf{F}}$	if $T[x] \neq \perp$ then return $T[x]$
return $[b = b']$	$T[x] \twoheadleftarrow \mathcal{Y}$; return $T[x]$

Figure 3: The PRF experiment used to define the pseudorandomness of function F with key space $\mathcal{K}$.

2.2 Data structures

Fix non-empty sets $\mathcal{D}, \mathcal{R}, \mathcal{K}$ of *data objects*, *responses* and *keys*, respectively. Let $\mathcal{Q} \subseteq \mathrm{Func}(\mathcal{D}, \mathcal{R})$ be a set of allowed *queries*, and let $\mathcal{U} \subseteq \mathrm{Func}(\mathcal{D}, \mathcal{D})$ be a set of allowed data-object *updates*. A *data structure* is a tuple $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$, where:

- $\textsc{Rep}\colon \mathcal{K} \times \mathcal{D} \to \{0,1\}^* \cup \{\perp\}$ is a randomized *representation algorithm*, taking as input a key $K \in \mathcal{K}$ and data object $\mathcal{S} \in \mathcal{D}$, and outputting the representation $\mathsf{repr} \in \{0,1\}^*$ of D, or $\perp$ in the case of a failure. We write this as $\mathsf{repr} \twoheadleftarrow \textsc{Rep}_K(\mathcal{S})$.
- $\textsc{Qry}\colon \mathcal{K} \times \{0,1\}^* \times \mathcal{Q} \to \mathcal{R}$ is a deterministic *query-evaluation algorithm*, taking as input $K \in \mathcal{K}$, $\mathsf{repr} \in \{0,1\}^*$, and $\mathsf{qry} \in \mathcal{Q}$, and outputting an answer $a \in \mathcal{R}$. We write this as $a \leftarrow \textsc{Qry}_K(\mathsf{repr}, \mathsf{qry})$.
- $\textsc{Up}\colon \mathcal{K} \times \{0,1\}^* \times \mathcal{U} \to \{0,1\}^* \cup \{\perp\}$ is a randomized *update algorithm*, taking as input $K \in \mathcal{K}$, $\mathsf{repr} \in \{0,1\}^*$, and $\mathsf{up} \in \mathcal{U}$, and outputting an updated representation repr', or $\perp$ in the case of a failure. We write this as $\mathsf{repr}' \twoheadleftarrow \textsc{Up}_K(\mathsf{repr}, \mathsf{up})$.

Allowing each of the algorithms to take a key K lets us separate (in our security notions) any secret randomness used across data structure operations, from per-operation randomness (e.g., generation of a salt). Note that our syntax admits the common case of *unkeyed* data structures, by setting $\mathcal{K} = \{\varepsilon\}$.

We formalize Rep as randomized to admit defenses against offline attacks and, as we will see, per-representation randomness will play an important role in achieving our notion of correctness in the presence of adaptive adversaries. Both Rep and the Up algorithm can be viewed (informally) as mapping data objects to representations — explicitly so in the case of Rep, and implicitly in the case of Up — so we allow Up to make per-call random choices, too. Many common data structures do not have randomized representation updates, but some do, e.g. the Cuckoo filter [14] and the stable Bloom filter [11].

Note that Up takes a function operating on data objects as an argument, even though Up itself operates on *representations* of data objects. This is intentional, to match the way these data structures generally operate. In a data structure representing a set or multiset, we often think of performing operations such as 'insert x' or 'delete y'. When the set or multiset is not being stored, but instead modeled via a representation, the representation must transform these operations into operations on the actual data structure it is using for storage. A Bloom filter, for example, will handle an 'insert x' query by hashing x and setting the resulting bits in the filter to 1. In this way, the abstract insertion function up_x, operating on sets, is handled by Up as a concrete action of setting certain bits in the filter. Side-effects of Up, or cases where the algorithm's behavior does not perfectly match the intended update up, are a potential source of errors that an adversary can exploit.

We also note that the query algorithm Qry is deterministic. This reflects the overwhelming behavior of data structures in practice, in particular those with space-efficient representations. It also allows us to focus on correctness errors caused by the actions of an adaptive adversary, without attending to those caused by randomized query responses. Randomized query responses may be of interest from a data privacy perspective, but our focus is on correctness.

3 Notions of Adversarial Correctness

Let $\Pi = (\textsc{Rep}, \textsc{Up}, \textsc{Qry})$ be a data structure with response space $\mathcal{R}$. We define two adversarial notions of correctness involving Π, an *error function* $\delta : \mathcal{R}^2 \to \mathbb{R}$, and an *error capacity* r. The values $\delta(x, y)$ of the error function represent the "badness" of getting an erroneous result of x from Qry when y should actually have been returned. In general we require $\delta(x, y) \geq 0$ and $\delta(x, x) = 0$ for all x and y, but otherwise place

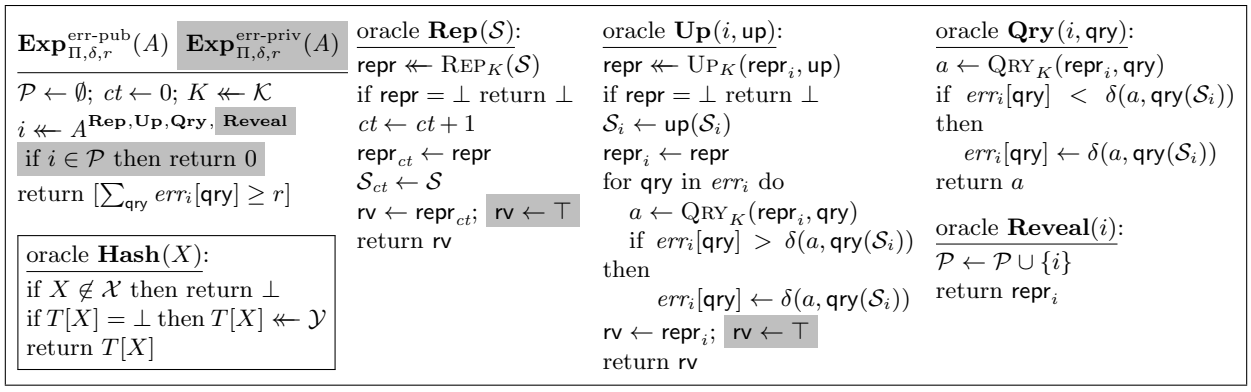

Figure 4: Two notions of adversarial correctness. The ERR-Pub notion captures correctness when the representation is always known to the adversary, while the ERR-Priv notion captures correctness when the representation is secret. When modeling a function $H : \mathcal{X} \to \mathcal{Y}$ as a random oracle, the **Hash** oracle is given to A, Rep, Up and Qry.

no restrictions on what the error function might look like. For example, in the case of Bloom filters we use a very simply error function: $\delta(x, y) = 1$ for any $x \neq y$.

The two correctness notions are given by the experiments in Figure 4. One corresponds to cases where the representations of the true data are public (ERR-Pub) and the other to where they are private (ERR-Priv). We will describe the former and then give a brief explanation of how the latter differs, as the two are closely related to each other.

Both experiments aim to capture the total *weight* of the errors caused by the adversary's queries. However, because we consider mutable data objects and representations, we only give the adversary credit for **Qry** calls that produce errors in the "current" data objects $\mathcal{S}_i$ and their representations repr_i. Because we consider mutable data objects and representations, the notion of "current" is defined by calls to the **Rep** and **Up** oracles. In the case of Bloom filters, for example, we want to keep track of all the false positives which have been found so far, except for those false positives which have since been turned into true positives.

To track errors, both experiments maintain an array $err_i[]$ for every data object $\mathcal{S}_i$ that has been defined. Initially, $err_i[]$ is implicitly assigned the value of $\perp$ at every index. (We will silently adopt the same convention for all uninitialized arrays.) For purposes of value comparison, we adopt the convention that $\perp < n$ for all $n \in \mathbb{R}$. Now, the array err_i is indexed by query functions qry, and the value of $err_i[\mathsf{qry}]$ is the weight of the error caused by qry, with respect to the *current* data object $\mathcal{S}_i$ and *current* representation repr_i (of $\mathcal{S}_i$). The value of $err_i[\mathsf{qry}]$ is updated within the **Qry**- and **Up**-oracles, but observe that $err_i[\mathsf{qry}] = \perp$ until (i, qry) is queried to the **Qry**-oracle. Intuitively, a representation repr_i of data object $\mathcal{S}_i$ cannot surface errors until it is queried.

When $\mathbf{Qry}(i, \mathsf{qry})$ executes, the value in $err_i[\mathsf{qry}]$ is overwritten iff the error caused by qry is larger than the existing value of $err_i[\mathsf{qry}]$. The first time (i, qry) is queried to **Qry** this is guaranteed, since the minimum possible value output by δ is 0. After this, the adversary gets credit only for making a worse error than the one already found. This prevents the adversary from trivially winning by repeatedly sending the same error-producing query to **Qry**. In our Bloom filter example, if **Qry** finds that qry is a new false positive, it will set $err_i[\mathsf{qry}]$ to 1, showing that an additional error has been produced.

When a query $\mathbf{Up}(i, \mathsf{up})$ is made, the oracle first updates the data object $\mathcal{S}_i$ and its corresponding representation. Now, for each defined value $err_i[\mathsf{qry}]$, we re-evaluate the error that *would* be caused by the previously asked qry, with respect to the newly updated $\mathcal{S}_i$ and repr_i. If the existing value of $err_i[\mathsf{qry}]$ is larger than the error that qry would cause (again, w.r.t. the newly updated $\mathcal{S}_i$ and repr_i), then we overwrite $err_i[\mathsf{qry}]$ with the smaller value. Doing so ensures that the array err_i does not overcredit the attacker for errors against the current data object and representation. For example, if x was previously found to be a false positive for a Bloom filter, and the adversary then inserts x into the data structure, we set $err_i[\mathsf{qry}_x]$ to 0. Since x is now a true positive rather than a false positive, it should no longer be counted as an error.

ERR-Priv differs from ERR-Pub only in that the **Rep** and **Up** oracles do not reveal the representation to the adversary. This models the case where the data structure is stored privately, where the adversary can ask queries but not see the full representation. To model the possibility that information about a representation is

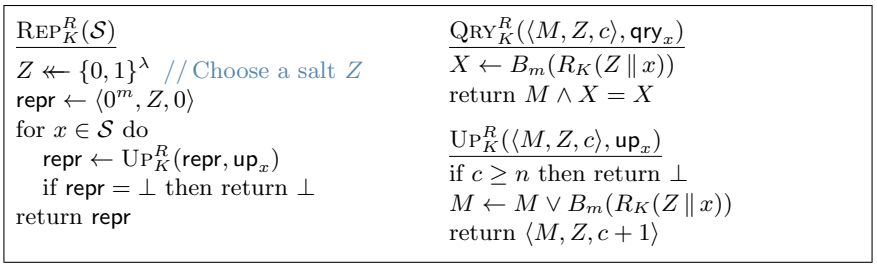

Figure 5: Keyed structure $\text{BLOOM}[R, n, \lambda] = (\text{REP}^R, \text{QRY}^R, \text{UP}^R)$ is used to define Bloom filter variants used to represent sets of at most n elements. The parameters are a function $R : \mathcal{K} \times \{0,1\}^* \to [m]^k$ and integers $n, \lambda \geq 0$. A concrete scheme is given by a particular choice of parameters. The function B_m is defined in Section 2.1.

eventually leaked, we also give the adversary a **Reveal** oracle that reveals a given representation. However, to prevent this from being trivially equivalent to the public-representation case we do not allow the adversary to win by finding errors in a representation which has been revealed using **Reveal**. Since the ERR-Pub adversary gets access to the same information that ERR-Priv does without the need for **Reveal** calls, ERR-Pub security is a stronger notion than ERR-Priv security.

We define the advantage of an ERR-Pub-adversary A as

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(A) = \Pr\left[\mathbf{Exp}^{\text{err-pub}}_{\Pi,\delta,r}(A) = 1\right]$$

and write $\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(t, q_R, q_T, q_U, q_H)$ as the maximum advantage of any ERR-Pub-adversary running in t time steps and making q_R calls to **Rep**, q_T calls to **Qry**, q_U calls to **Up**, and q_H calls to **Hash** in the ROM. We define ERR-Priv advantage in kind, except that we add an extra parameter q_V representing the number of calls to **Reveal**. We sometimes use ERR-Pub1 or ERR-Priv1 to refer to the restriction of the ERR-Pub or ERR-Priv games to the case of $q_R = 1$; for these we remove the q_R parameter from the advantage function.

4 Bloom Filters

In this section we consider two classes of Bloom filters, each employing a different strategy to determine when the filter reaches full capacity. The first class is specified in Figure 5. This class of n-capped filters captures the classical setting in which the filter is used to represent some fixed number of elements $n \geq 0$. Our construction $\text{BLOOM}[R, n, \lambda] = (\text{REP}^R, \text{QRY}^R, \text{UP}^R)$ has two additional parameters besides the cap: a function $R : \mathcal{K} \times \{0,1\}^* \to [m]^k$ and the *salt length* $\lambda \geq 0$. Let $H : \{0,1\}^* \to [m]^k$ be a hash function and let $\ell, n, \lambda \geq 0$ be integers. The standard Bloom filter is the structure $\text{BF}[H, n] = \text{BLOOM}[\text{ID}^H, n, 0]$, which we will term the *basic* Bloom filter. It has no key (the key space of ID^H is $\{\varepsilon\}$, see Section 2.1) and does not use a salt. The *salted* Bloom filter $\text{SBF}[H, n, \lambda] = \text{BLOOM}[\text{ID}^H, n, \lambda]$ is the same except that it allows a non-empty salt. Finally, we consider a salted variant that uses a PRF instead of a hash function. The *keyed* Bloom filter $\text{KBF}[F, n, \lambda]$ is the structure $\text{BLOOM}[F, n, \lambda]$, where $F : \mathcal{K} \times \{0,1\}^* \to [m]^k$ is a PRF. Note that the basic and salted BFs have key space $\{\varepsilon\}$ and the keyed BF has key space $\mathcal{K}$.

In this section, we will show that the basic Bloom filter construction $\text{BF}[H, n]$ is insecure in our setting. This is because it allows the adversary to make an offline attack that has a high probability of success while using a minimal number of queries. In the immutable setting, where the adversary is constrained to never use the **Up** oracle, i.e. $q_U = 0$, it suffices to use the SBF construction in order to provide a security guarantee in either the public-representation or private-representation settings. However, in the case where we allow $q_U > 0$ so that the adversary can make updates, we will find that SBF is only secure in the ERR-Priv setting. To provide ERR-Pub security when updates are needed, KBF must be used instead.

At the end of the section, we discuss the second class of filters that we call the *ℓ-thresholded* filters. Instead of rejecting updates after a pre-determined number of elements are added to the set, a thresholded filter is deemed full once at least $\ell \geq 0$ bits of the filter are set. In the usual, non-adaptive setting, this implementation behaves very similarly to the standard "n-capped" Bloom filter, but we find that a filter

threshold allows us to obtain better bounds. We will demonstrate this for salted BFs in the ERR-Priv setting.

NON-ADAPTIVE FALSE-POSITIVE PROBABILITY. Let $\rho : \{0,1\}^* \to [m]^k$ be a function, $\lambda \geq 0$ be an integer, and define $\text{BLOOM}[\text{ID}^\rho, n, \lambda] = (\text{REP}^\rho, \text{QRY}^\rho, \text{UP}^\rho)$ as in Figure 5. (Note the mild abuse of notation by which we write "ρ" instead of "ID^ρ".) Let $\mathcal{S} \subseteq \{0,1\}^*$ be a set of length n. We define the non-adaptive, false positive probability for Bloom filters as

$$P_{k,m}(n) = \Pr\left[\rho \twoheadleftarrow \text{Func}(\{0,1\}^*, [m]^k); \mathsf{repr} \twoheadleftarrow \text{REP}^\rho(\mathcal{S}); \\ x \twoheadleftarrow \{0,1\}^* \setminus \mathcal{S} : \text{QRY}^\rho(\mathsf{repr}, \mathsf{qry}_x) = 1 \mid \mathsf{repr} \neq \perp\right]. \tag{1}$$

That is, $P_{k,m}(n)$ is the probability that some x is a false positive for the representation of some $\mathcal{S}$ for which $|\mathcal{X}| = n$ and $x \notin \mathcal{S}$, when a random function is used for hashing. Because of the randomization provided by ρ, this probability is independent of $\mathcal{S}$ and x. Finding a tight, concrete upper bound for $P_{k,m}(n)$ has proven challenging, but we do understand its asymptotic behavior. Kirsch and Mitzenmacher [20] prove that, for certain choices of k and m as functions of n, it holds that $P_{k,m}(n) = \lim_{n\to\infty}(1 - e^{-kn/m})^k$. Moreover, they demonstrate via simulation that this is a very good approximation of the false positive probability. In lieu of a concrete upper bound, we will refer to $P_{k,m}(n)$ as defined in Equation (1) in the remainder.

ERROR FUNCTION FOR SET-MEMBERSHIP QUERIES. Throughout this section we will use the error function δ defined as

$$\delta(x, y) = \begin{cases} 0 & \text{if } x = y \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

This simply indicates whether the query result matched the correct response.

4.1 Insecurity of unsalted BFs

The performance of basic Bloom filters is well-understood, assuming the choice of set $\mathcal{S}$ being represented is independent of the choice of hash function. When this assumption is violated, however, their performance can be substantially degraded [18]. Here we show that, even when we (optimistically) model the hash function as a random oracle, basic BFs cannot achieve security in our setting. The basic Bloom filter has no salt and no secret key. Let $H : \{0,1\}^* \to [m]^k$ be a function, fix $n \geq 0$, and let $\Pi = \text{BF}[H, n] = (\text{REP}^H, \text{QRY}^H, \text{UP}^H)$ as defined above. With no per-representation randomness and no secret key to be concealed from the adversary, there is no difference between ERR-Pub and ERR-Priv security, as the adversary can easily compute the representation of any set for itself. This ability of the adversary to reconstruct the set without making queries allows for various attacks that badly harm the accuracy of the filter.

POLLUTION ATTACKS. Gerbet et al. [18] provide the following example of an attack setting and a potential attack against Bloom filters. Suppose the adversary is interacting with a system representing a dataset with Π and that it is able to choose some fraction of the input data. For example, consider a web crawler which performs a "crawl" of webpages [10], following the links on each page it visits in order to index, archive, or otherwise analyze websites. In order to keep track of the set of webpages which have already been visited during a crawl, some crawlers use a Bloom filter which is updated to include each new page the crawler visits. Suppose the adversary controls at least one such webpage along the crawl's path and wishes to deny the spider access to a different webpage, the "target webpage". The adversary can choose the links present on its own webpage, which will cause the spider to visit the chosen webpages and set the corresponding bits of its Bloom filter to 1. If those links are chosen in such a way that they produce a false positive for the target webpage, the spider will then erroneously believe it has already visited the target webpage. The target webpage will therefore never be visited during the spider's crawl.

In cases where the adversary is able to control at least some of the filter inputs, Gerbet et al. describe an attack where the adversary chooses a set of inputs that maximizes the number of 1s in the filter. This strategy is especially effective when the structure of the hash function is known to the adversary. In particular, as long as the choice of hash function and any associated parameters are public, the adversary can compute the hash function on its own in order to determine which choices maximize the number of bits set to 1, or which choices will set certain target bits to 1 in order to cause specific false positives. They show that with

$m = 3200$ and $k = 4$, the adversary can double the false positive rate if they control 200 out of a total of $n = 600$ insertions, under the assumption that H is known to and computable by the attacker.

Gerbet et al. suggest various ways to mitigate pollution attacks, such as choosing the parameters k, m, n so that even if a pollution attack occurs, the false positive rate is kept below some threshold of acceptability. This strategy is potentially viable, but may significantly increase the amount of memory required to store the data structure. The bounds we provide show how the parameters of a filter can be tweaked to keep the error rate low not just in the presence of this specific type of attack, but in the presence of any adversary covered by our more general attack model; doing so, however, will require altering the structure.

Gerbet et al. also discuss the possibility of using a secretly-keyed hash function. In the attack model they consider, where representations are kept private indefinitely, this suffices to prevent the pollution attack they describe. However, under the more general attack models where the representation may eventually be recovered (in the private-representation setting via **Reveal**) or is public, simply using a PRF *without per-representation randomness* does not suffice for security in our setting.

Target-set coverage attacks. Of course, exhibiting a high false positive rate is not the only way a Bloom filter might fail to be correct. In particular, it would be undesirable if the filter were consistently incorrect on a *particular set of inputs*. Rather than pollute the filter, the adversary's goal might be to craft a set of legitimate looking inputs that cover some disjoint target set of inputs. This type of attack is nicely captured by our adversarial model. In a *target-set coverage attack*, the adversary is given a small target set $\mathcal{T} \subseteq \{0,1\}^*$ and searches for a cover set $\mathcal{R} \subseteq \{0,1\}^*$ such that $\textsc{Qry}^H(\textsc{Rep}^H(\mathcal{R}), \mathsf{qry}_x) = 1$ for each $x \in \mathcal{T}$. Once a suitable cover set is found, the adversary queries $\mathbf{Rep}(\mathcal{R})$. Then for each $x \in \mathcal{T}$, it asks $\mathbf{Qry}(\mathsf{qry}_x)$, achieving a score of $r = |\mathcal{T}|$.

This ERR-Priv1 attack succeeds with probability 1 assuming a covering set can be found. If $|\mathcal{T}| \leq |\mathcal{R}|$, then such a set exists; but finding it may be computationally infeasible, depending on the size of the cover set, the size of the target set, and the parameters of the Bloom filter. Fix integer $n \geq 0$, let $H : \{0,1\}^* \to [m]^k$ be a function, and let $\Pi = \mathrm{BF}[H, n, 0]$ as specified in Figure 5. The possibility of pre-computing the structure in the ERR-Priv1 experiment yields the following attack. Given a set $\mathcal{T} \subseteq \{0,1\}^*$ of target queries, the adversary searches for a set $\mathcal{S} \subseteq \{0,1\}^*$ such that $\textsc{Qry}^H(\textsc{Rep}^H(\mathcal{S}), x) = 1$ for all $x \in \mathcal{T}$. We call such a set a *cover* set. Once a suitable $\mathcal{S}$ is found, the adversary queries $\mathbf{Rep}(\mathcal{S})$ followed by $\mathbf{Qry}(x)$ for each $x \in \mathcal{T}$. Assuming $|\mathcal{T}| \geq r$, where r is the error capacity, the attack succeeds with probability 1. Next we describe a heuristic strategy for finding a covering set and evaluate its performance in terms of success rate and computational cost. We will (conservatively) model H as a random oracle.

We first choose a set $\mathcal{S}$ of $s > n$ *potential test queries*. Our goal is to find a subset of $\mathcal{S}$ that covers $\mathcal{T}$ but contains at most n elements. For each test query x, we compute $X = B_m(\mathbf{Hash}(x))$. If we have the resources to compute $X_1 \vee \cdots \vee X_n$ for each of the $\binom{s}{n}$ size-n subsets of targets, we will eventually find a suitable covering set if such a set exists. The query complexity of this approach is modest; we need to make $s + r$ queries to **Hash** (s for the test set, r for the target set), one query to **Rep**, and r queries to **Qry**. However, checking $\binom{s}{n}$ sets would be infeasible, even for modest choices of k, m, and n. Observe, however, that there is a lot of sub-structure to exploit in the search. In particular, there is a simple heuristic strategy for finding a satisfying set (if it exists) in which we need only check about $O(kr)$ sets on average.

Let $\mathcal{S} = \{x_1, \ldots, x_s\}$ be the set of potential test queries, and let $\mathcal{T} = \{x_{s+1}, \ldots, x_{s+r}\}$ be the set of target queries. We construct a tree whose vertices are labeled with subsets of $[s]$ as follows. Let $\emptyset$ be the root. For each node I and each $w \in [s] \setminus I$, if $|I| < n$, then let $I \cup \{w\}$ be a child of I. The new attack works as follows: traverse the tree depth-first, beginning at $\emptyset$, until a vertex I is reached such that each element of $\mathcal{T}$ is a false positive for the representation of $\mathcal{S}_I = \{x_i : i \in I\} \subseteq \mathcal{S}$. Then for every child J of I, the elements of $\mathcal{T}$ are false positives for $\mathcal{S}_J$. The adversary may choose any one of these as its query to **Rep**.

The tree has $\binom{s}{n}$ leaves, and it will traverse the entire tree if there is no solution. Hence, the worst-case runtime is the same as before. However, we can reduce the search space dramatically in two ways. First, if there is no solution, then often this can be determined without traversing the tree. Let $M_{\mathcal{S}}$ and $M_{\mathcal{T}}$ denote the filters corresponding to the set of test and target queries, respectively. If for some $i \in [m]$, the i-th bit of $M_{\mathcal{T}}$ is set, but the i-th bit of $M_{\mathcal{S}}$ is not set, then it is immediate that there is no covering set. The second is a greedy heuristic for eliminating branches of the search tree. The idea is that we only take a branch if it results in covering an additional bit in $M_{\mathcal{T}}$. More precisely, for each child J of I, we do as follows: if there is a bit i such that the i-th bit of $M_{\mathcal{S}_J}$ and $M_{\mathcal{T}}$ are set, but the i-th bit of $M_{\mathcal{S}_I}$ is *not set*, then the branch J is taken; otherwise it is not. This optimization results in a heuristic attack, since the search may miss the

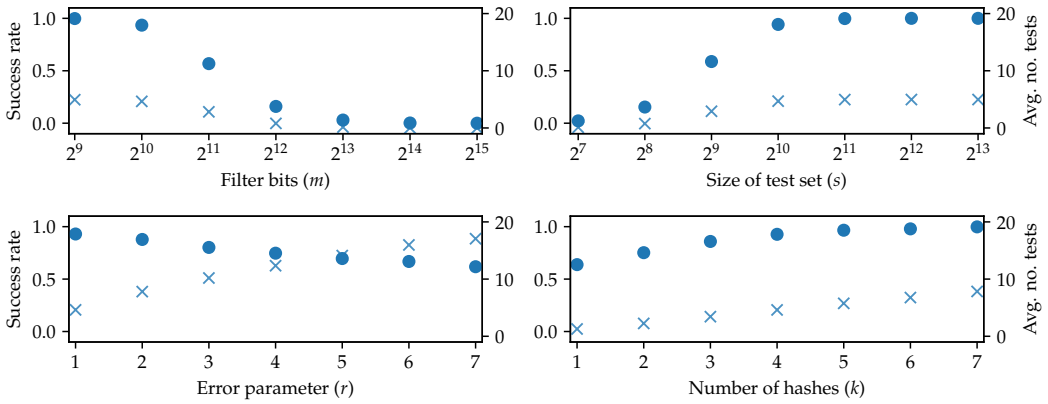

Figure 6: Success rate and average-case time complexity for the optimized attack on classic Bloom filter. Each plot shows the success rate (●), as well as the average number of tested sets (×), for 1000 executions of the attack on simulated inputs. Default parameters are $k = 4$, $m = 2^{10}$, $n = 100$, $r = 1$, and $s = 2^{10}$. In each plot, one of these parameters is varied.

optimal solution.

We implemented this attack and evaluated its performance. Figure 6 shows the success rate and average number of sets checked for a number of simulations and various parameters. (For each simulation we start with a "fresh" choice of random oracle.) Unsurprisingly, the success rate decreases as we increase the number of filter bits (top-left of Figure 6); however, with $k = 4$, $m = 1024$, $n = 100$, and $r = 1$, a test set of just 512 elements suffices for a success rate of nearly 60% (top-right). It is also worth noting that increasing the error parameter only slightly decreases the success rate (bottom-left). These results show that even for very pessimistic parameter choices, the basic Bloom filter is not secure in the ERR-Priv1 sense. Finally, we find that the average number of sets that were tested is about $O(kr)$ within all of the parameter regimes we studied.

4.2 Salted BFs in the (im)mutable setting

Here we consider the correctness of Bloom filters when the hashed input is prepended with a salt. Fix $H : \{0,1\}^* \to [m]^k$ and $n, \lambda \geq 0$ and let $\Pi = \mathrm{SBF}[H, n, \lambda]$.

If the adversary can update the representation via **Up**, then it can perform an ERR-Pub1 attack against Π that is closely related to the attacks in the previous section. The adversary calls $\mathbf{Rep}(\emptyset)$, getting an empty filter and the salt in response. It may then use the salt to construct representations on its own just as described in the target-set coverage attack. The works because the adversary can test for errors on its own because it knows the salt. In practice, an adversary may not be able to perform this exact attack, since even in the streaming setting it is possible that the salt is not immediately revealed to the adversary. However, as soon as the adversary does learn the salt, it can immediately launch a target-set coverage attack against the filter, without having to make any queries directly to the filter.

Without the ability to insert elements even after the salt has been seen, the above attack fails. Indeed, when we restrict ourselves to the immutable setting, we can prove the following.

Theorem 1 (Immutable ERR-Pub security of salted BFs)**.** *Let $p = P_{k,m}(n)$. For all integers $q_R, q_T, q_H, r, t \geq 0$ it holds that*

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(t, q_R, q_T, 0, q_H) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + \left(\frac{pq}{r}\right)^r e^{r-pq}\right] ,$$

where H is modeled as a random oracle, $q = q_T + q_H$, and $r > pq$.

We consider only the case of $r > pq$ because pq is the expected number of false positives obtained by an adversary that simply uses its knowledge of the salt (after the representation is created) to guess as many random elements as possible. Because this simple adversary can get pq successes on average, we can only hope to provide good security bounds against arbitrary adversaries in the case that $r > pq$.

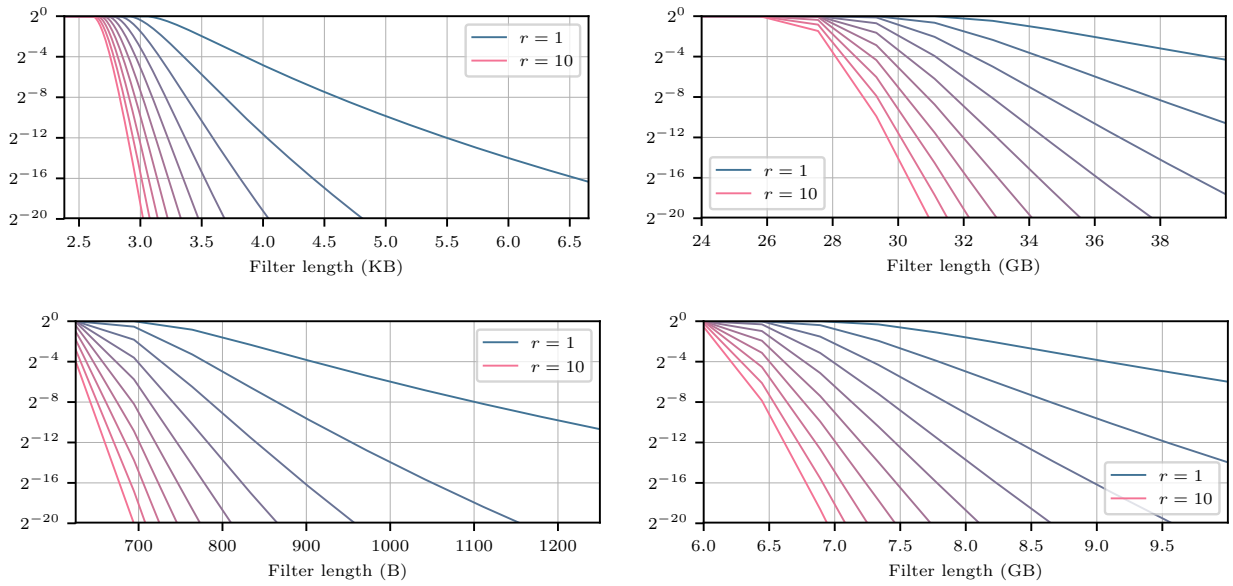

Figure 7: The value of $\zeta_{k,m,n}(q,r)$ (Equation (3)) for: $k = 16$; $q = 2^{64}$ (top) or 2^{32} (bottom); $n = 100$ (left) or 10^9 (right); varying values of r (one line per r-value); and varying values of the filter length m (the x-axis). Note the log-2 scale on the y-axis.

Before giving the proof, let us take a moment to unpack the result a bit. The bound can be broken down into three main components. The factor of q_R means that the bound we can prove is weakened somewhat when a number of representations are observed by the adversary. (In Section 4.4, we will show that we can do better by thresholding rather than capping.) The $q_H/2^\lambda$ term corresponds to the probability of the adversary guessing the salt before the representation is constructed, but this will be very small as long as λ is chosen to be sufficiently large (say, $\lambda = 128$). The final, messier term comes from applying a Chernoff bound to the non-adaptive adversary's probability of succeeding in the experiment given $q = q_H + q_T$ guesses. By way of clarifying the performance of our bound, we have plotted the last component for various parameters of interest. Let

$$\zeta_{k,m,n}(q,r) = \left(\frac{p^* q}{r}\right)^r e^{r-p^*q} \tag{3}$$

where $p^* = (1 - e^{-kn/m})^k$, the approximation of the non-adaptive false positive probability given by Kirsch and Mitzenmacher [20]. Figure 7 shows values of $\zeta_{k,m,n}(q,r)$ for varying m. What these plots show is that, for a given error capacity r, once a certain lower bound on the filter size is reached, the ζ term decreases quite quickly. Moreover, the rate at which ζ decreases scales nicely with the error capacity. For example, if one is willing to tolerate up to $r = 10$ false positives for a filter representing $n = 100$ elements, then picking a filter length of 3 kilobytes is sufficient to ensure that observing 10 false positives occurs with probability less than 2^{-17}, even when the adversary can make $q = 2^{64}$ **Hash** or **Qry** queries.

We concede that requiring a 3KB for a filter a set of 100 elements may be prohibitive in some applications. We would require a substantially smaller filter for smaller q, but unfortunately, a query complexity of $q = 2^{64}$ in the ERR-Pub setting is quite realistic, since the attack can be carried out offline. In the ERR-Priv setting, or in the ERR-Pub setting when we use a PRF instead of a hash function, the adversary's attack is largely *online*, rendering the q term overly conservative. In these settings, a significantly smaller filter will do. For example, if we assume that an adversary making an online attack will make no more than $q = 2^{32}$ online queries, we get the results seen in the lower plots of Figure 7. Beyond this, if a larger error rate is acceptable, the filters can again be made substantially smaller.

Proof of Theorem 1. We will use the following lemma for keyless structures.

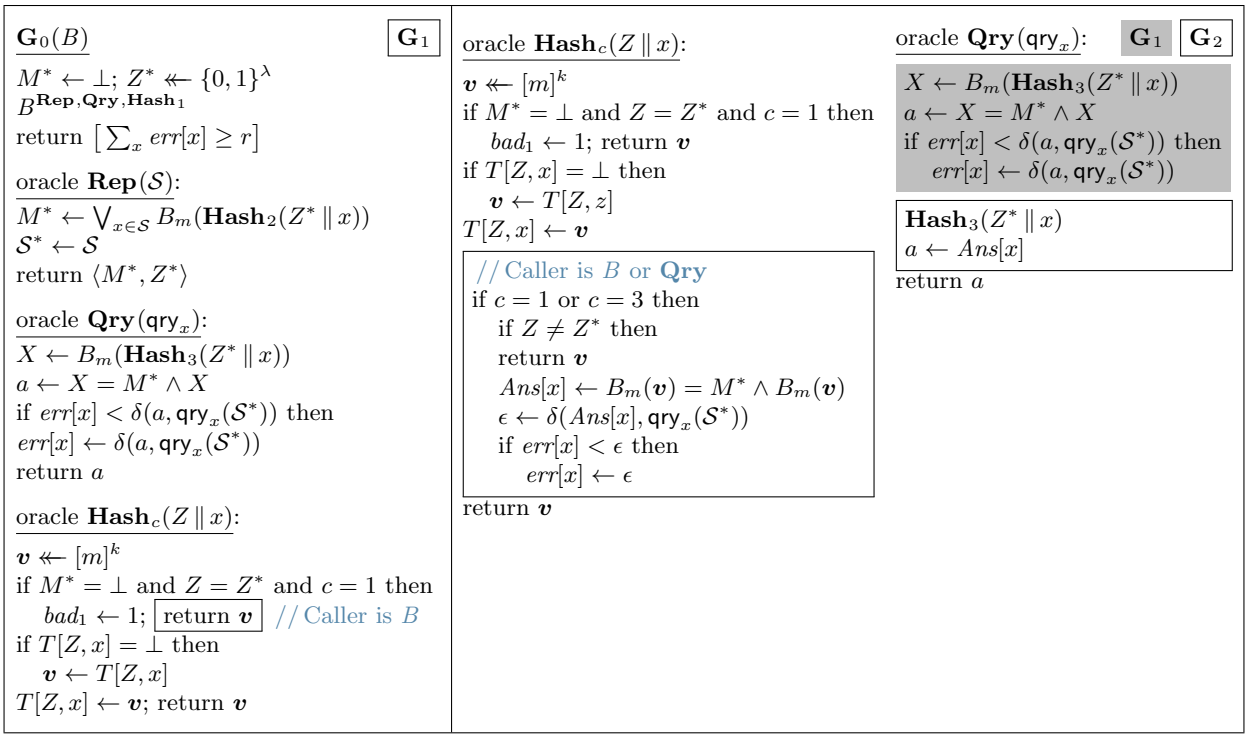

Figure 8: Games 0, 1, and 2 for proof of Theorem 1.

Lemma 1. *For every $q_R, q_T, q_U, q_H, r, t \geq 0$ and keyless structure Γ it holds that*

$$\mathbf{Adv}^{\text{err-pub}}_{\Gamma,\delta,r}(t, q_R, q_T, q_U, q_H) \leq q_R \cdot \mathbf{Adv}^{\text{err-pub1}}_{\Gamma,\delta,r}(O(t), q_T, q_U, q_H)\,,$$

Proof. The proof is by a fairly straightforward hybrid argument. Let B be an adversary for the ERR-Pub case. Because Γ is keyless, note that an ERR-Pub1 adversary A can perfectly simulate the behavior of the oracles in the experiment. This allows A to simulate B as follows. At the start of the game, A picks i from $[q_R]$ uniformly at random. Then A simulates B while also simulating all oracle queries except those related to the i-th representation. When the i-th call to **Rep** is made, A makes its single **Rep** call. Any **Up** and **Qry** calls B makes for the i-th representation are forwarded by A to its own oracles, whereas all other oracle calls are simulated. Once B halts, A also halts and returns 1. Since the simulation is perfect, A is guaranteed to win as long as two conditions occur: A's choice of i matches the output of B, and B itself would win the experiment. Since A has a $1/q_R$ chance of randomly picking the correct value for i, we have the result that $\mathbf{Adv}^{\text{err-pub}}_{\Gamma,\delta,r}(B) \leq q_R \cdot \mathbf{Adv}^{\text{err-pub1}}_{\Gamma,\delta,r}(A)$. ■

Let A be an ERR-Pub adversary making 1 query to **Rep**, q_T queries to **Qry**, 0 queries to **Up**, and q_H queries to the random oracle **Hash**. We make the following assumptions, all of which are without loss of generality. First, all of A's **Qry** queries follow its **Rep** query. Second, we assume that $x \notin \mathcal{S}$ for all queries qry_x to **Qry**, where $\mathcal{S}$ was the input to A's **Rep** query. This is without loss because Bloom filters admit false positives, but not false negatives. Third, we we assume that $|\mathcal{S}| \leq n$; this is without loss because otherwise **Rep** outputs $\perp$ and A gets no advantage. Fourth, we assume that all of A's **Hash** queries are of the form $Z \,\|\, x$, where $|Z| = \lambda$.

We begin with a game-playing argument [2], then obtain the final bound via application of Lemma 1. The high-level goal is to rewrite the game so that the probability that one of A's queries runs up the score is precisely the non-adaptive false positive probability. In other words, our goal is to transition into a setting in which the Bloom filter output by **Rep** is independent of the outcome of A's other queries.

Consider the game $\mathbf{G}_0(B)$ defined in Figure 8. It is similar to the ERR-Pub experiment when executed with A, Π, δ, and r, but the pseudocode has been simplified to clarify our argument. Indeed, it is not difficult

to see that for every A there exists an adversary B such that

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(A) \le \Pr[\,\mathbf{G}_0(B) = 1\,] \tag{4}$$

and B has the same resources as A. Adversary B executes A, forwarding A's oracle queries to its own oracles in the natural way.

Observe that in game $\mathbf{G}_0$ the salt used for the representation of $\mathcal{S}^*$ is generated prior to executing B. Game $\mathbf{G}_1$ is identical to $\mathbf{G}_0$ until the flag bad_1 gets set by oracle **Hash**. This occurs if B asks $\mathbf{Hash}_1(Z^* \,\|\, x)$, where Z^* is the salt generated at the beginning of the game, and it has not yet called its **Qry** oracle (i.e., $M^* = \perp$). By the Fundamental Lemma of Game Playing [2] it follows that

$$\begin{aligned} \Pr[\,\mathbf{G}_0(B) = 1\,] &\le \Pr[\,\mathbf{G}_1(B) = 1\,] + \Pr[\,\mathbf{G}_1(B) \,\mathbf{sets}\, bad_1\,] &(5)\\ &\le \Pr[\,\mathbf{G}_1(B) = 1\,] + q_H/2^\lambda\,. &(6)\end{aligned}$$

Note that in $\mathbf{G}_1$, the value of M^* is independent of B's $\mathbf{Hash}_1$ queries. In particular, the probability that some bit of M^* is set is independent of random coins of B.

In game $\mathbf{G}_2$ the **Hash** and **Qry** oracles have been rewritten so that the winning condition is computed by **Hash** instead of **Qry**. The former oracle maintains a set Ans such that $Ans[x] = \text{QRY}^{\mathbf{Hash}_3}(M^*, \mathsf{qry}_x)$ for each query $Z^* \,\|\, x$; on input of qry_x, oracle **Qry** simply runs $\mathbf{Hash}_3(Z^* \,\|\, x)$ and returns $Ans[x]$. We are effectively giving the adversary credit for RO queries that result in false positives for the representation of $\mathcal{S}^*$, but which it does not explicitly ask of **Qry**. Because B's advantage in the new game is at least as much as it gets in the old one, we have that.

$$\Pr[\,\mathbf{G}_1(B) = 1\,] \le \Pr[\,\mathbf{G}_2(B) = 1\,]\,. \tag{7}$$

We now consider $\Pr[\,\mathbf{G}_2(B) = 1\,]$. Let $\mathcal{X}$ be the set $\{x \in \{0,1\}^* : Ans[x] \neq \perp\}$ and $\mathcal{T} = \{x \in \mathcal{X} : Ans[x] = 1\}$, where Ans is at is defined when B halts. We will call $\mathcal{X}$ the set of attempts and $\mathcal{T}$ the set of false positives. Note that $\mathcal{X} \cap \mathcal{S}^* = \emptyset$ by assumption, and $|\mathcal{X}| \le q_H + q_T$ by definition. Hence, the probability that $\mathbf{G}_2(B) = 1$ is equal to the probability that $|\mathcal{T}| \ge r$.

For each $x \in \mathcal{X}$, let $T(x)$ denote the event that $x \in \mathcal{T}$. In the random oracle model for H, the set of random random variables $T(x)$ for each $x \in X$ are independently and identically distributed. Hence, the probability that B succeeds is binomially distributed:

$$\Pr[\,\mathbf{G}_2(B) = 1\,] = \Pr[\,|\mathcal{T}| \ge r\,] = \sum_{i=r}^{q} \binom{q}{i} p^i (1-p)^{q-i}\,, \tag{8}$$

where $q \le q_H + q_T$ and $p = \Pr[T(x) = 1]$. Here we can apply a Chernoff bound which states that, for any $\delta > 0$,

$$\Pr[\,X \ge (1+\delta)\mu\,] < \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu\,. \tag{9}$$

We set $\delta = r\mu^{-1} - 1$ and note that $\mu = pq$. This yields

$$\Pr[\,|\mathcal{T}| \ge r\,] < \left(\frac{e^{r\mu^{-1}-1}}{(r\mu^{-1})^{r\mu^{-1}}}\right)^\mu = \left(\frac{e^{r-\mu}}{(r\mu^{-1})^r}\right) = e^{r-pq}\left(\frac{pq}{r}\right)^r \tag{10}$$

and so

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(A) < \frac{q_H}{2^\lambda} + \left(\frac{pq}{r}\right)^r e^{r-pq}\,. \tag{11}$$

Applying Lemma 1 to move from the single-representation case to the general case, we get our final bound of

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(A) \le q_R \cdot \left[\frac{q_H}{2^\lambda} + \left(\frac{pq}{r}\right)^r e^{r-pq}\right]\,. \tag{12}$$

□

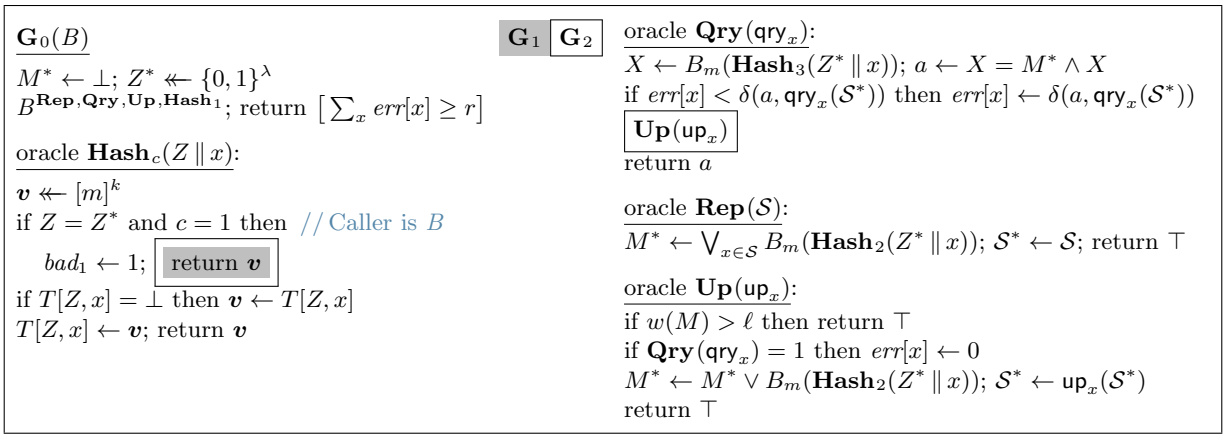

Figure 9: Games 0, 1, and 2 for proof of Theorem 2.

Security in the mutable setting. Recall that the ERR-Pub1 attack against mutable salted filters exploited the fact that the adversary learned the salt as soon as the filter was created, and that from this it could compute the hash function on its own. Even if the filter is mutable, we can prevent this attack from working as long as we require that the filter under attack be kept secret from adversaries. In fact, we can attain the following ERR-Priv bound for Π.

Theorem 2 (ERR-Priv security of salted BFs)**.** *Let $p' = P_{k,m}(n+r)$. For all integers $q_R, q_T, q_U q_H, q_V, r, t \geq 0$, if $r > p' q_T$, then it holds that*

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(t, q_R, q_T, q_U, q_H, q_V) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + \left(\frac{p' q_T}{r}\right)^r e^{r - p' q_T}\right],$$

where H is modeled as a random oracle.

The proof follows a similar structure to that of Theorem 1. The main differences come from arguing that without a "lucky" guess of the salt, the adversary cannot use **Hash** to find false positives, and from having to show that the adversary's access to **Up** does not substantially change the security bound that can be derived. The first of these is straightforward given the private-representation setting, but the second requires investigating how much of an advantage the **Up** oracle can give, then moving to games where this advantage is taken into account.

Proof of Theorem 2. Just as in the proof of Theorem 1, we will assume the adversary just makes a single call to **Rep** and use Lemma 1 to complete the bound. Let A be an ERR-Priv adversary making exactly 1 call to **Rep**, q_T calls to **Qry**, q_U calls to **Up**, and q_H calls to **Hash**. Because A creates only a single representation, it will necessarily lose if it calls **Reveal** on that representation. We may therefore assume without loss of generality that A makes no calls to **Reveal**, and because of this we omit **Reveal** from each of the games.

In addition to the assumptions of Theorem 1, we assume without loss of generality that the adversary never uses **Up** to insert an element into $\mathcal{S}$ which is already present in the set, and never uses **Up** to insert an element x where $\mathbf{Qry}(\mathsf{qry}_x)$ has already been called and has returned a positive result. Since these insertions do not change the filter, the adversary would gain no advantage from performing these updates. Furthermore, we assume without loss of generality that an adversary halts as soon as it determines it has accumulated enough errors to win the experiment.

We begin with a game $\mathbf{G}_0(B)$ (Figure 9) similar to the first game in the proof of Theorem 1, except that it also defines an **Up** oracle. Again, we observe that for every A there exists a B such that

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(A) \leq \Pr[\,\mathbf{G}_0(B) = 1\,] \tag{13}$$

and B has the same query resources as A.

Since we are seeking a stronger bound, we now wish to isolate the possibility that the adversary *ever* guesses the salt, as opposed to just guessing the salt before calling **Rep**. This is no longer a trivial task for the adversary because the representations are private, and so **Rep** does not directly reveal the salt. We therefore set the bad_1 flag whenever the adversary manages to guess the salt, without the requirement that $M^* = \perp$. However, since the adversary is still limited to a total of q_H **Hash** queries, regardless of when the queries are made, we can follow nearly the same argument as in the previous proof to get the bound

$$\Pr[\,\mathbf{G}_0(B) = 1\,] \leq \Pr[\,\mathbf{G}_1(B) = 1\,] + q_H/2^{\lambda}\,. \tag{14}$$

In $\mathbf{G}_1$, $\mathbf{Hash}_1$ queries are always independent of $\mathbf{Hash}_2$ and $\mathbf{Hash}_3$ queries. In particular, it is irrelevant whether the adversary guesses the salt. We still cannot move to the binomial distribution for non-adaptive queries, however, since $\mathbf{Hash}_2$ and $\mathbf{Hash}_3$ queries are not necessarily independent of each other. By one of our starting assumptions, the same input is never provided twice to $\mathbf{Hash}_2$ because the adversary never tries to insert an element which is already in $\mathcal{S}$. We also want to show that an adversary never queries the same element twice. To do this, let B be an adversary for $\mathbf{G}_1$. We construct an adversary C that achieves at least the same advantage without making repeated queries. This C simulates B, maintaining a list of queries that have been made so far during the game. Any **Rep**, **Up**, or $\mathbf{Hash}_1$ queries from B are forwarded to C's oracles without performing additional computations. When B makes a $\mathbf{Qry}(\mathsf{qry}_x)$ call, C checks whether x has already been queried. If so, C selects as y the lexicographically first string such that $y \notin \mathcal{S}$ and such that y has not been previously queried, and calls $\mathbf{Qry}(\mathsf{qry}_y)$ instead of $\mathbf{Qry}(\mathsf{qry}_x)$.

Recall that B makes no queries for elements of $\mathcal{S}$, so any repeated queries must have returned either false positives or true negatives the first time they were queried. If B makes a $\mathbf{Qry}(\mathsf{qry}_x)$ where x was previously found to be a false positive, the total number of errors cannot possibly increase since B has already gotten credit for this error. On the other hand, if B calls $\mathbf{Qry}(\mathsf{qry}_x)$ for an x that was previously found to be a true negative, it is possible that x has since become a false positive due to **Up** calls that have occurred since. However, since $\mathbf{Hash}_3$ calls are independent of $\mathbf{Hash}_2$ calls with different inputs, it is just as likely that those intervening updates have made y a false positive. Therefore, regardless of what type of queries B makes, C makes queries that are at least as likely to produce false positives and is therefore at least as likely to win, i.e. $\Pr[\,\mathbf{G}_1(B) = 1\,] \leq \Pr[\,\mathbf{G}_1(C) = 1\,]$. Since C only changes the inputs to some of B's oracle queries, but does not change whether or not a query is made, B and C have identical query resources.

By the reduction from B to C, we are now dealing with an adversary where the hash queries are all independent except for **Qry** and **Up** calls to the same element. We will now further reduce from C to D, where D immediately follows any $\mathbf{Qry}(\mathsf{qry}_x)$ that finds a true negative with a call to $\mathbf{Up}(\mathsf{up}_x)$ to insert that element. We have D simulate C while maintaining a count of updates that have been performed so far. Any **Rep** and $\mathbf{Hash}_1$ calls from C are forwarded to D's oracles without performing any additional computation. Any $\mathbf{Qry}(\mathsf{qry}_x)$ call is also forwarded, but if the oracle reveals the element is a true negative then D immediately calls $\mathbf{Up}(\mathsf{up}_x)$ unless q_U updates have already been performed. Finally, if C makes an **Up** call then D forwards the call unless it has already made q_U updates, in which case it just returns $\top$ to C.

By our earlier assumptions, there are only two types of update C may make:

1. Inserting an element which is not already in $\mathcal{S}$ and has previously been tested with **Qry**, returning a negative result.
2. Inserting an element which is not already in $\mathcal{S}$ and has not previously been tested with **Qry**.

Since calls to $\mathbf{Hash}_3$ with different choices of x are independent of each other, and since $\mathbf{Hash}_3$ uses random sampling, the effects of type 1 updates on the representation are identically distributed. Similarly, since calls to $\mathbf{Hash}_2$ produce independent random results, the effects of type 2 updates on the representation are also identically distributed. However, the effects of the two types of update are *not* identically distributed compared to each other. In particular, making a type 1 update ensures that at least one new bit in the filter will be set to 1, since the distribution of $B_m(\mathbf{Hash}_2(Z^* \,\|\, x))$ is conditioned on not producing a false positive. On the other hand, making a type 2 update provides no guarantee about how many bits in the filter might be set to 1. Type 1 updates are therefore always preferable for an adversary attempting to produce false positives.

Note that, at any point during the experiment, D has always made at least as many updates as C has. Furthermore, all of the updates made by D but not by C are type 1 updates, which are maximally effective at increasing the error rate. Therefore any **Qry** calls made by C have at least the same probability of causing an error when made by D, and so $\Pr[\,\mathbf{G}_1(C) = 1\,] \leq \Pr[\,\mathbf{G}_1(D) = 1\,]$. Since D is capped at making q_U total updates and its handling of other oracles is identical, its query resources are still the same as C.

For $\mathbf{G}_2$, then, we enforce this update-after-query behavior, changing $\mathbf{Qry}(\mathsf{qry}_x)$ to automatically insert x into $\mathcal{S}$ after computing the correct response to the query. For any D for $\mathbf{G}_1$ we can construct E for $\mathbf{G}_2$ that simulates D to attain the same advantage, forwarding oracle queries in the natural way except that any call of the form $\mathbf{Up}(\mathsf{up}_x)$ are ignored if $\mathbf{Qry}(\mathsf{qry}_x)$ has been called previously. Ignoring these **Up** calls does not negatively affect the adversary because in $\mathbf{G}_2$ the original $\mathbf{Qry}(\mathsf{qry}_x)$ call already automatically inserts x into the set. Then E wins whenever D does, and $\Pr[\,\mathbf{G}_1(D) = 1\,] \leq \Pr[\,\mathbf{G}_2(E) = 1\,]$. Since E performs at most as many oracle queries as D, its query resources are the same.

However, the parameters of the games played by E and D are slightly different. In particular, D (and, by extension, E) may find up to r false positives before halting. When D finds these false positives they are by assumption not ever inserted into $\mathcal{S}$, while in the case of E the **Qry** oracle automatically inserts them into the set as soon as they are found. While inserting a false positive does not affect the filter itself in any way, it does increment the number of elements in the underlying set. Therefore if adversaries D in $\mathbf{G}_1$ are limited to representing a set of size n, we restrict adversaries E in $\mathbf{G}_2$ to representing sets of up to size $n+r$.

We now therefore only consider the case of a **Rep** call followed by the $\mathbf{G}_2$ version of **Qry** calls. Let $\mathcal{X}$ be the set of all queries qry_x which are sent to **Qry** over the course of the experiment. We necessarily have $|\mathcal{X}| \leq q_T$, and each $\mathsf{qry}_x \in \mathcal{X}$ has some probability of causing an error. Since $\mathcal{S}^*$ never grows to contain more than $n + r$ elements regardless of what **Rep** or **Up** calls are made, and because the results of **Qry** calls are independent of any prior oracle queries, the false positive probability for each such qry_x is bounded above by p', the false-positive probability of a Bloom filter containing $n + r$ elements. So we have, analogously to the proof of Theorem 1,

$$\Pr[\,\mathbf{G}_2(E) = 1\,] \leq \sum_{i=r}^{q_T} \binom{q_T}{i} p'^i (1-p')^{q_T - i}\,, \tag{15}$$

where q_T replaces q and the larger p' replaces p. Applying the same Chernoff bound reduces this to

$$\Pr[\,\mathbf{G}_2(E) = 1\,] \leq e^{r - p'q_T} \left(\frac{p'q_T}{r}\right)^r. \tag{16}$$

Again we apply Lemma 1 to get a final bound of

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(A) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + \left(\frac{p'q_T}{r}\right)^r e^{r - p'q_T}\right]. \tag{17}$$

□

4.3 Keyed BFs

Salted BFs are ERR-Priv secure in general, and are ERR-Pub secure in the immutable setting, but are not ERR-Pub secure when the adversary has access to an **Up** oracle. Our argument for the ERR-Priv security of salted Bloom filters is made possible by virtue of the structure under attack not being revealed to the adversary. While this is realistic in many applications, it may be desirable for the Bloom filter to be public *and* updatable. Here we show that building a Bloom filter from a PRF suffices for security in this setting. Let $F : \mathcal{K} \times \{0,1\}^* \to [m]^k$ be a function, fix integers $n, \lambda \geq 0$, and let $\Pi = \text{KBF}[F, n, \lambda]$.

Theorem 3 (ERR-Pub security of keyed BFs)**.** *Let $p' = P_{k,m}(n + r)$. For integers $q_R, q_T, q_U, q_H, r, t \geq 0$ such that $r > p'q_T$, it holds that*

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(t, q_R, q_T, q_U, q_H) \leq \mathbf{Adv}^{\text{prf}}_F(O(t), nq_R + q_T + q_U) + \frac{q_R^2}{2^\lambda} + \left(\frac{p'q_Rq_T}{r}\right)^r e^{r - p'q_Rq_T}. \tag{18}$$

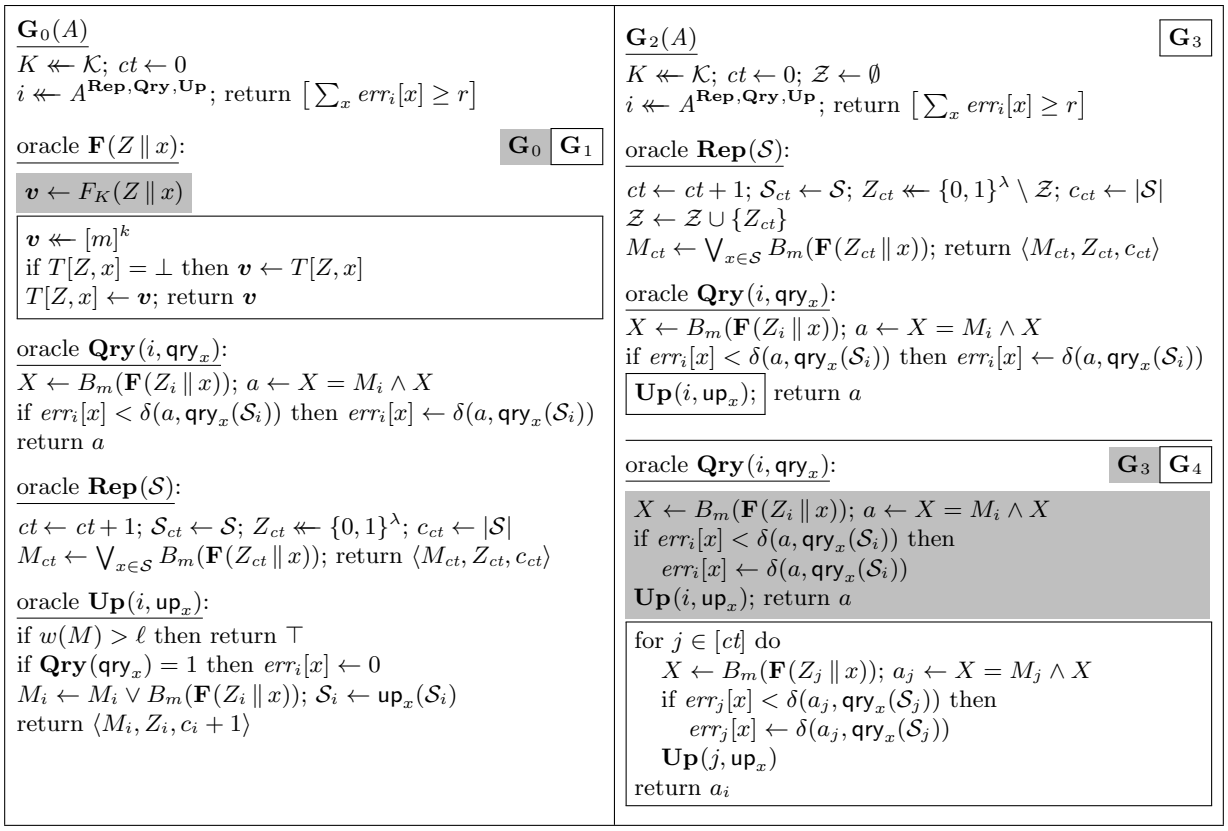

Figure 10: Games 0–4 for proof of Theorem 3.

As usual, our strategy will be to move to the non-adaptive setting via a sequence of game transitions, but the details of how we get there differ from the case of keyless Bloom filters. In particular, since we are using a PRF, the initial parts of the proof deal with the adversary potentially being able to break the PRF and with the possibility of the salts repeating rather than with the adversary being able to guess the salt.

Proof of Theorem 3. We start with a game $\mathbf{G}_0$, shown in Figure 10, which is essentially the same as the standard ERR-Pub experiment on a Bloom filter, given the assumption (without loss of generality) that the adversary never attempts to construct a representation for a set with more than n elements. Unlike in the previous two proofs, we cannot use Lemma 1 because an adversary cannot simulate the oracles without knowing the private key. We use an alternate approach to gradually reduce to the standard binomial bound deriving from the non-adaptive false positive probabilities. The first thing we want to do is to bound the probability that the adversary can break the PRF.

The number of times the PRF is evaluated on distinct inputs is bounded by the number of queries available to the adversary. In particular, **Qry** and **Up** each call the PRF once, while **Rep** may call the PRF up to n times. Thus, when executed with A, game $\mathbf{G}_0$ makes at most $Q = q_T + q_U + nq_R$ queries to $\mathbf{F}$. In $\mathbf{G}_1$, we have a game which is identical to $\mathbf{G}_0$ except that it uses random sampling in place of the PRF. If A cannot distinguish the PRF from a random function then these games are indistinguishable from the adversary's perspective. We exhibit a $O(t)$-time, PRF-adversary D making at most Q queries to its oracle such that

$$\mathbf{Adv}^{\mathrm{prf}}_F(D) = \Pr[\,\mathbf{G}_0(A) = 1\,] - \Pr[\,\mathbf{G}_1(A) = 1\,]\ . \tag{19}$$

Adversary $D^{\mathbf{F}}$ works by executing A in game $\mathbf{G}_1$, except that whenever the game calls *its* $\mathbf{F}$, adversary D uses its own oracle to compute the response. Finally, when A halts, if the winning condition in $\mathbf{G}_1$ is satisfied, then D outputs 1 as its guess; otherwise it outputs 0. Then conditioning on the outcome of the coin flip b

in D's game, we have that

$$\begin{aligned}
\mathbf{Adv}^{\mathrm{prf}}_{F}(D) &= 2\Pr\left[\mathbf{Exp}^{\mathrm{prf}}_{F}(D)=1\right]-1 && (20)\\
&= 2\left(1/2\Pr\left[\mathbf{Exp}^{\mathrm{prf}}_{F}(D)=1 \mid b=1\right]+1/2\Pr\left[\mathbf{Exp}^{\mathrm{prf}}_{F}(D)=1 \mid b=0\right]\right)-1 && (21)\\
&= \Pr\left[\mathbf{Exp}^{\mathrm{prf}}_{F}(D)=1 \mid b=1\right]+\Pr\left[\mathbf{Exp}^{\mathrm{prf}}_{F}(D)=1 \mid b=0\right]-1 && (22)\\
&= \Pr[\,\mathbf{G}_0(A)=1\,]-\Pr[\,\mathbf{G}_1(A)=1\,]\,. && (23)
\end{aligned}$$

Next, our goal is to argue, in a similar manner as to the previous theorems, that all of the oracle calls are independent. In order to guarantee this we must deal with the possibility of a salt collision between different representations. In $\mathbf{G}_2(A)$ we require that all salts be distinct between representations. By the birthday bound, collisions between randomly-generated salts occur with frequency at most $q_R^2/2^\lambda$, so $\Pr[\,\mathbf{G}_1(A)=1\,] \leq q_R^2/2^\lambda + \Pr[\,\mathbf{G}_2(A)=1\,]$.

With guaranteed-unique salts, the result of each **Rep**, **Up**, and **Qry** call for a given representation is independent of the calls for all other representations. By an almost identical argument to the one in the proof of Theorem 2, we can reduce from any A to an adversary B which follows any **Qry** call that finds a true negative with an **Up** call to insert that element, and therefore move to $\mathbf{G}_3(B)$, which as in the Theorem 2 proof performs an update after each query is made, with the guarantee that $\Pr[\,\mathbf{G}_1(A)=1\,] \leq \Pr[\,\mathbf{G}_2(B)=1\,]$.

Finally, we must deal with the possibility that the adversary chooses which representations to target with **Up** and **Qry** calls based on the result of **Rep**, since some representations may be more full than others. In game $\mathbf{G}_4$, we allow the adversary credit if a call to **Qry** produces an error in any of the representations that have been constructed. Furthermore, the updates made by **Qry** apply to all representations that are not already full. Since all **Up** calls are identically and independently distributed, and having more elements in a filter cannot decrease the false positive rate, the fact that some representations may become full more quickly than they otherwise would have can only help the adversary. Similarly, having **Qry** count errors across all representations never harms the adversary, and so the adversary's advantage may only increase when moving to $\mathbf{G}_4$, i.e. $\Pr[\,\mathbf{G}_3(B)=1\,] \leq \Pr[\,\mathbf{G}_4(B)=1\,]$.

We are now in a situation where we can apply the standard, non-adaptive error bound. Let $\mathcal{X}$ be the set of all queries qry_x made by the adversary over the course of the game. As in the previous proof, we have $|\mathcal{X}| \leq q_T$. However, qry_x may now cause a false positive in any of the representations. The probability of causing a false positive in a specific representation is still given by the non-adaptive false positive probability p' for a Bloom filter containing $n+r$ elements. Since the representations are independent of each other, the probability of a false positive occurring in any of up to q_R representations is at most $p'q_R$. We can therefore bound the adversary's as before. In particular,

$$\Pr[\,\mathbf{G}_4(B)=1\,] \leq e^{r-p'q_Rq_T}\left(\frac{p'q_Rq_T}{r}\right)^r. \tag{24}$$

So, substituting this bound back into the earlier advantage inequalities, we find the final bound of

$$\mathbf{Adv}^{\text{err-pub}}_{\Pi,\delta,r}(A) \leq \mathbf{Adv}^{\mathrm{prf}}_{F}(D) \quad + \frac{q_R^2}{2^\lambda} + \left(\frac{p'q_Rq_T}{r}\right)^r e^{r-p'q_Rq_T}\,.$$

□

The fact that both a key and a salt are used in the KBF construction is critical. In particular, without the per-representation randomness given by the salt, we would not be able to argue that **Up** and **Qry** calls are independent across representations. On the contrary, seeing the representation of a singleton set $\{x\}$ would immediately allow the adversary to test whether x was a member in every other representation that had been constructed, simply by testing whether every bit set to 1 in the representation of $\{x\}$ was also set to 1 in other representations. Even in the ERR-Priv game, using the **Reveal** oracle on some representations leaks information about other representations, and again we cannot use the argument that provides the above bound.

$\underline{\text{REP}_K^R(\mathcal{S})}$
$Z \twoheadleftarrow \{0,1\}^\lambda$ // Choose a salt Z
$\mathsf{repr} \leftarrow \langle 0^m, Z\rangle$
for $x \in \mathcal{S}$ do
  $\mathsf{repr} \leftarrow \text{UP}_K^R(\mathsf{repr}, \mathsf{qry}_x)$
  if $\mathsf{repr} = \perp$ then return $\perp$
return repr

$\underline{\text{QRY}_K^R(\langle M, Z\rangle, \mathsf{qry}_x)}$
$X \leftarrow B_m(R_K(Z \,\|\, x))$
return $M \wedge X = X$

$\underline{\text{UP}_K^R(\langle M, Z\rangle, \mathsf{qry}_x)}$
if $w(M) > \ell$ then return $\perp$
return $\langle M \vee B_m(R_K(Z \,\|\, x)), Z\rangle$

Figure 11: The class of ℓ-thresholded Bloom filters is given by $\text{BLOOM}_{\text{ft}}[R, \ell, \lambda] = (\text{REP}^R, \text{QRY}^R, \text{UP}^R)$. This is a slight variant of n-capping wherein we use the Hamming weight of the filter (w, as defined in Section 2.1) to decide if the filter is full.

We note that Gerbet et al. [18] suggest using keyed hash functions as one possibility for constructing secure filters, which is equivalent in our terminology to using a keyed but unsalted filter. The distinction is that Gerbet et al. assume that representations are kept private indefinitely, an assumption similar to that underlying our ERR-Priv game, but with the stronger restriction that the adversary has no equivalent of a **Reveal** oracle. This makes their notion of security much weaker than ours with respect to keyed structures.

4.4 ℓ-thresholded BFs

So far we have proven bounds for only n-capped BFs. It is important to understand the security of this class of structures because it is representative of how BFs are used in practice. In this section we demonstrate that we can improve security bounds by defining "fullness" in terms of the Hamming weight of the filter, rather than the number of elements it represents. The general form of this alternate construction is formally defined in Figure 11. We can define the more specific constructions $\text{BF}_{\text{ft}}[H, \ell]$, $\text{SBF}_{\text{ft}}[H, \ell, \lambda]$, and $\text{KBF}_{\text{ft}}[H, \ell, \lambda]$ in an exactly the same way as the n-capped variants. Here we only consider case of $\Pi = \text{SBF}_{\text{ft}}[H, \ell, \lambda]$ and compare it to the SBF construction in Section 4.2. The non-adaptive false positive probability is similar is similar to capped filters, since the number of 1 bits in the filter can be closely predicted from the number of elements in a randomly-selected underlying set. Because of this, and because we are able to demonstrate better security bounds for an ℓ-thresholded filter than for a capped filter (Theorem 2), we suggest this as a way of providing strong security guarantees for even smaller filter sizes.

Theorem 4 (ERR-Priv security of thresholded BFs)**.** *Let* $p_\ell = ((\ell+k)/m)^k$*. For integers* $q_R, q_T, q_U, q_H, q_V, r$*,* $t \geq 0$ *such that* $r > p_\ell q_T$*, it holds that*

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(t, , q_R, q_T, q_U, q_H, q_V) \leq \frac{q_R(q_H + q_R)}{2^\lambda} + e^{r - p_\ell q_T} \left(\frac{p_\ell q_T}{r}\right)^r ,$$

where H *is modeled as a random oracle.*

From a technical point of view, the main difference between thresholded and capped filters is that attacks cannot set more than $\ell + k$ bits of the filter to 1, regardless of how the attack is conducted. The thrust of the proof is to conservatively assume the adversary will always be able to produce such a maximally full filter, and then use a standard binomial-distribution-based bound to place a limit on the adversarial advantage even in this worst-case scenario.

Proof of Theorem 4. As before, we assume without loss of generality that there are no insertions of or queries for elements of $\mathcal{S}$, and we start with a game $\mathbf{G}_0$, defined in Figure 12, that is identical to the ERR-Priv game for KBF_{ft}.

To avoid the unfortunate q_R factor in the bound, we do not make use of Lemma 1 in this proof. Because of that, we must find some other way to ensure that **Reveal** is not useful to the adversary. In particular, if there are unique salts across representations, the **Rep**, **Qry**, and **Up** calls for one representation will be independent of those for other representations, since the unique salt is passed as part of the input. Therefore in $\mathbf{G}_1$ we specify that all salts created will be unique, but deny access to **Reveal**. By the birthday bound, the probability of salts repeating in $\mathbf{G}_0$ is no more than $q_R^2/2^\lambda$. If the representations are independent, calling **Reveal** would provide no information about other representations, and would in fact

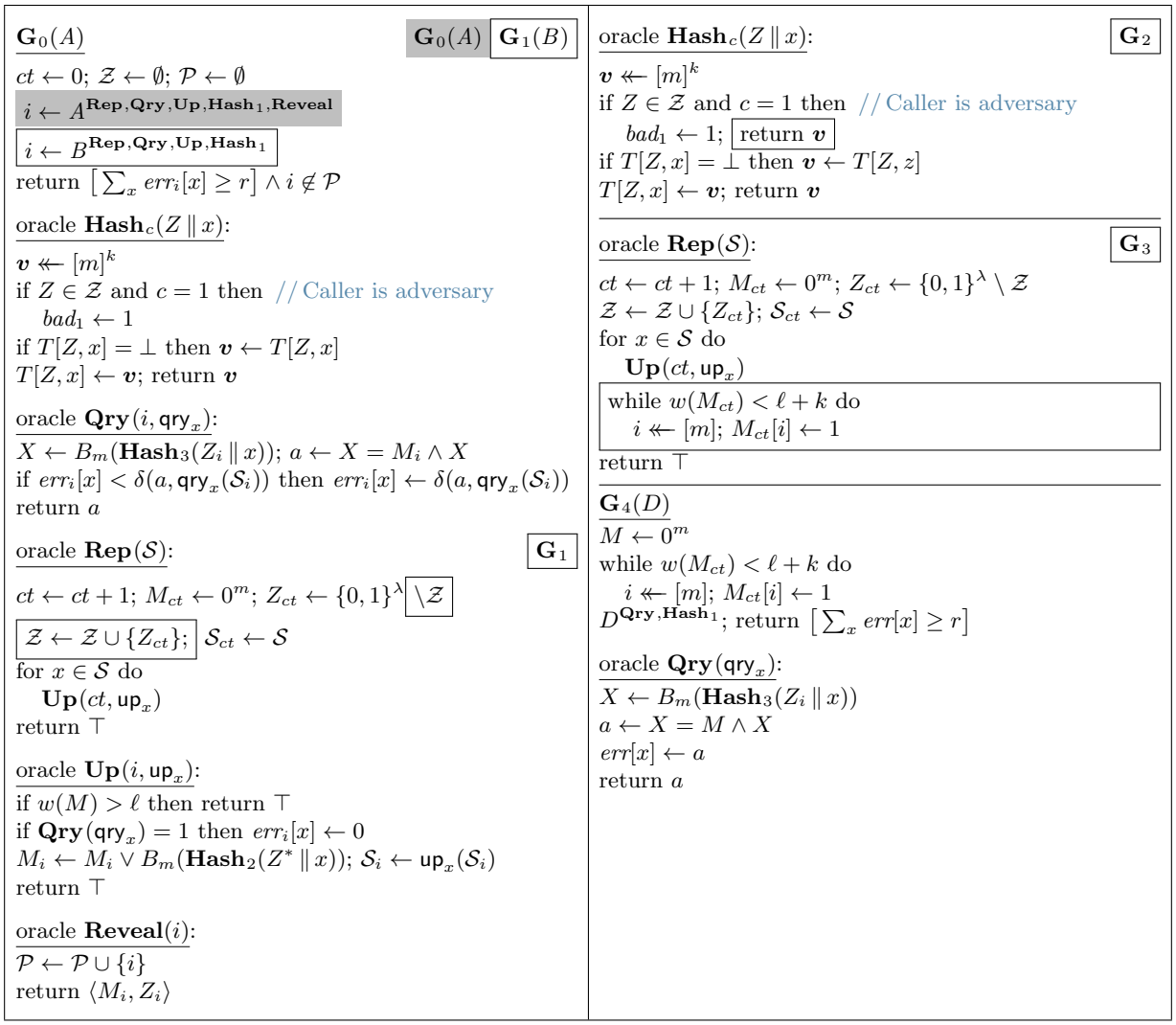

Figure 12: Games 0, 1, and 2 for proof of Theorem 4.

only weaken the adversary by causing some possible outputs $i \in [q_R]$ to be automatic losses. So we have $\Pr[\,\mathbf{G}_0(A) = 1\,] \leq q_R^2/2^\lambda + \Pr[\,\mathbf{G}_1(B) = 1\,]$, where B is an adversary that performs identically to A but is syntactically distinct because it lacks a **Reveal** oracle.

Next, we want to ensure that the adversary's $\mathbf{Hash}_1$ queries are independent of the $\mathbf{Hash}_2$ and $\mathbf{Hash}_3$ queries used for **Rep**, **Qry**, and **Up**. Since the $\mathbf{Hash}_c$ oracles use random sampling to fill a shared table, this occurs if and only if the adversary calls $\mathbf{Hash}_1(Z_i \,\|\, x)$ for some salt Z_i used by one of the representations created by **Rep**. By an argument very similar to that in the previous proofs, the adversary has at most $q_H/2^\lambda$ probability of calling $\mathbf{Hash}_1$ with the salt used by some specific representation. However, since there are now q_R representations, each with a distinct salt, there is at most a $q_R q_H/2^\lambda$ probability of the adversary correctly guessing a salt. In $\mathbf{G}_1(B)$, we set the bad_1 flag if the adversary succeeds in guessing the salt in this manner, but the flag does not affect the game. The case of $\mathbf{G}_2(B)$ is identical until the bad_1 flag is set, which occurs only when a salt is guessed, so we have $\Pr[\,\mathbf{G}_1(B) = 1\,] \leq q_R q_H/2^\lambda + \Pr[\,\mathbf{G}_2(B) = 1\,]$.

Now the adversary derives no advantage from guessing the salts of the representations, in the sense that the outputs of $\mathbf{Hash}_1$ are independent of the results of all calls to **Rep**, **Qry**, and **Up**. The next oracle we want to target is **Up**. Now that we have a filter threshold, we want to argue that the adversary cannot use **Up** to mount an effective attack. In $\mathbf{G}_3(C)$, the **Rep** oracle creates the filter as normal and then randomly sets bits until filter is full (i.e., its Hamming weight is at least ℓ), which is $\ell + k$ (since updates are not

allowed when more than ℓ bits are set, and a single update may set at most k bits to 1). For any B in $\mathbf{G}_2$, we can construct C for $\mathbf{G}_3$ that obtains at least as large an advantage by having C simulate B, forwarding all oracle queries in the natural way except that **Up** queries are ignored, with C simply returning $\top$ to B without performing any additional computations or oracle calls. (Recall that, in the ERR-Priv setting, the adversary expects $\top$ to be the output of **Up**.) Since B does not query elements which are already in $\mathcal{S}$, the outputs of $\mathbf{Hash}_3$ calls are independent of any prior $\mathbf{Hash}_2$ calls. The probability of such an output producing a false positive is strictly a function of the number of bits in the filter which have been set to 1. Since at least as many bits have been set to 1 in $\mathbf{G}_3$ as in $\mathbf{G}_2$, every **Qry** call is at least as likely to produce a false positive. Therefore $\Pr[\,\mathbf{G}_2(B) = 1\,] \leq \Pr[\,\mathbf{G}_3(C) = 1\,]$.

In $\mathbf{G}_3$, **Up** calls do not actually change the representation. Since each representation is created using independent $\mathbf{Hash}_2$ outputs and then filled in a uniform random manner until $\ell + k$ bits are set to 1, and is never modified afterwards, the representations themselves are random bitmaps which are uniformly distributed over the set of m-length bitmaps with $\ell + k$ bits set to 1. This allows us to move to $\mathbf{G}_4(D)$, where the adversary is given a single arbitrary bitmap repr of length m with $\ell + k$ bits set to 1 and makes **Qry** calls exclusively for repr, winning if it produces r errors for that 'representation'. Given an adversary C for $\mathbf{G}_3$, we construct a D for $\mathbf{G}_4$ that simulates C. When C makes a **Rep** call, D immediately returns $\top$ to C, and when C makes a $\mathbf{Qry}(i, \mathsf{qry}_x)$ call, D selects the lexicographically first y that has not yet been queried and calls $\mathbf{Qry}(y)$, returning the result to C. Random oracle calls are forwarded to D's own random oracle. Since representations are uniformly distributed and $\mathbf{Hash}_3$ calls are independent of the $\mathbf{Hash}_2$ calls used to construct the representation, each **Qry** call made by D has the same chance of producing a hit as the **Qry** call made by C has of producing a false positive. The winning conditions differ only in that D wins by accumulating r positive results, whereas C must accumulate r false positives within a single representation, and so D wins if C does. Therefore $\Pr[\,\mathbf{G}_3(C) = 1\,] \leq \Pr[\,\mathbf{G}_4(D) = 1\,]$.

However, since **Qry** calls with distinct inputs have independent outputs, each **Qry** call made by D has the same probability of producing a false positive. In particular, the probability of any one of the k outputs of $\mathbf{Hash}_3$ colliding with a 1 bit is $(\ell + k)/m$, and the probability of all k outputs doing so is then $((\ell + k)/m)^k$. If we let $\mathcal{X}$ be the set of all inputs made to **Qry**, we again have a binomial distribution where q_T queries are made. Letting $p_\ell = ((\ell + k)/m)^k$, we have

$$\Pr[\,\mathbf{G}_4(D) = 1\,] \leq \sum_{i=r}^{q_T} \binom{q_T}{i} p_\ell^i (1 - p_\ell)^{q_T - i}\,, \tag{25}$$

and we can once more apply a Chernoff bound, as long as $p_\ell q_T < r$, to simplify this to

$$\Pr[\,\mathbf{G}_4(D) = 1\,] \leq e^{r - p_\ell q_T} \left(\frac{p_\ell q_T}{r}\right)^r. \tag{26}$$

Substituting this into our earlier inequalities yields

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(A) \leq \frac{q_R(q_H + q_R)}{2^\lambda} + e^{r - p_\ell q_T} \left(\frac{p_\ell q_T}{r}\right)^r. \tag{27}$$

□

4.5 Discussion

The target-set coverage attack shows that the standard Bloom filter construction is weak to adaptive adversaries. Moving to the salted SBF construction mitigates this, but if filters are public they must be both large and immutable (Theorem 1). In the ERR-Priv setting updates do not break security and the minimum size of the filter to guarantee a fixed error rate is considerably reduced (Theorem 2). The guarantee (or, alternatively, filter size) can be further improved, especially if the number of representations constructed is large, by using ℓ-thresholding (Theorem 4). Additionally, if the filters themselves cannot be kept private but a secret key for the hash functions *can* be concealed from adversaries, the KBF construction shows how to provide security in the ERR-Pub setting (Theorem 3).

These requirements are more stringent than the mitigations suggested by Gerbet et al. [18] due to our stronger attack model (where multiple filters can be constructed, and sometimes revealed, to the adversary)

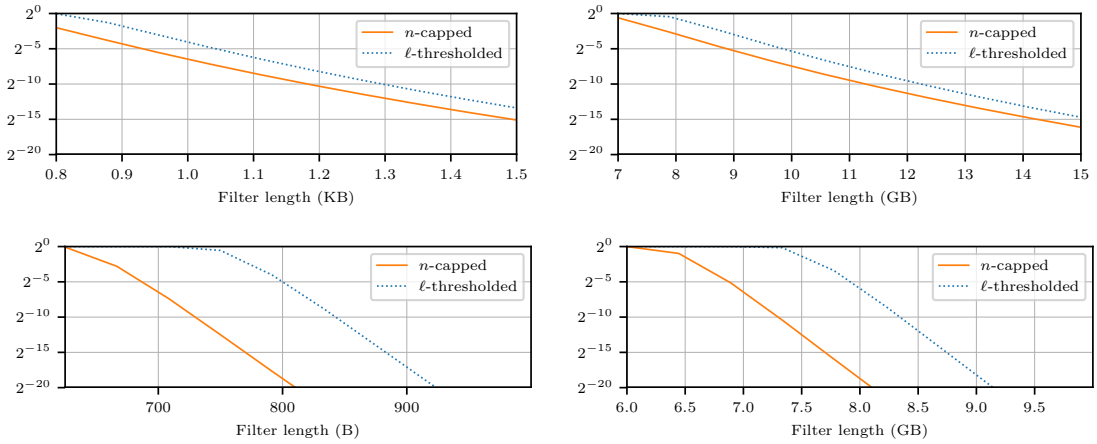

Figure 13: Performance of n-capped versus ℓ-thresholded Bloom filters. The solid orange line shows the value of $\zeta_{k,m,n}(q,r)$ for: $k = 16$; $q = 2^{32}$; $n = 100$ (left) or 10^9 (right); $r = 1$ (top) or 5 (bottom); and varying m (on the x-axis). The dotted blue line shows the dominant term in the bound of Theorem 4 for $\ell = nk$. The bounds are comparable, but thresholding would perform much better than capping for even modest q_R.

and our goal of establishing a general security bound for any adversary rather than mitigating specific attacks. If q_R is small, our ERR-Priv guarantee for SBF and ERR-Pub guarantee for KBF show that filters need not be made much larger than Gerbet et al.'s in order to provide comparable security against more general adversaries. If q_R is large, however, the q_R term in the error bounds means that the filters must be made large to provide good error guarantees. In this scenario, however, the ℓ-thresholding class of filter provides a way to get strong error guarantees without significantly increasing the filter size.

Capping versus thresholding. Figure 13 shows the dominant terms in the ERR-Priv bounds for n-capped and ℓ-threholded salted BFs (Theorem 2 and 4 respectively). This shows us that the bounds are comparable for $\ell = nk$, which is a reasonable choice of ℓ given that a set of size n can set at most nk bits to 1, and this upper bound only occurs in the unlikely circumstance that there are no hash collisions during insertion. When we take into account the factor of q_R present in the n-capped security bound, we conclude that thresholding provides significantly more security if the adversary is allowed even a small number of additional calls to q_R.

5 Counting Filters

Counting filters are a modified version of Bloom filters. Like ordinary Bloom filters, counting filters are only designed to handle set membership queries, but counting filters are designed to allow for deletion as well as insertion [15]. Our construction $\textsc{Count}[R, \ell, \lambda]$ defined in Figure 14 involves an ℓ-thresholded version of this structure. The traditional description of a counting filter involves a parameter n describing the size of the input multiset, whereas our ℓ describes the maximum number of nonzero counters. As in the case of ordinary Bloom filters, this does not significantly change the operation of the filter in a non-adversarial setting, since for random input multisets the number of nonzero counters is closely related to the number of elements in the multiset. In the presence of an adversary, however, we expect ℓ-thresholding to provide better security bounds than the traditional definition would provide by handicapping pollution attacks and similar adversarial strategies.

We will show that, in the ERR-Pub setting, the counting filter is insecure regardless of whether ℓ-thresholding is used or not, and regardless of the details of the behavior of the hash function or PRF used to insert and delete elements. On the other hand, we show that ERR-Priv security is achievable even under the assumption that a salted but unkeyed hash function is used, i.e. $\lambda > 0$ but $\mathcal{K} = \{\emptyset\}$.

Error function for frequency queries. Unlike with a Bloom filter, counting filters must account for

$\underline{\text{Rep}_K^R(\mathcal{S})}$
$Z \twoheadleftarrow \{0,1\}^\lambda$ // Choose a salt Z
$\mathsf{repr} \leftarrow \langle \text{zeroes}(m), Z\rangle$
for $x \in \mathcal{S}$ do
  $\mathsf{repr} \leftarrow \text{Up}_K^R(\mathsf{repr}, \mathsf{up}_{x,1})$
  if $\mathsf{repr} = \perp$ then return $\perp$
return repr

$\underline{\text{Qry}_K^R(\langle \boldsymbol{M}, Z\rangle, x)}$
$\boldsymbol{X} \leftarrow R_K(Z \,\|\, x)$
for $i \in \boldsymbol{X}$ do
  if $\boldsymbol{M}[i] = 0$ then return 0
return 1

$\underline{\text{Up}_K^R(\langle \boldsymbol{M}, Z\rangle, \mathsf{up}_{x,b})}$
if $c \geq n$ then return $\perp$
$\boldsymbol{M}' \leftarrow \boldsymbol{M}$; $\boldsymbol{X} \leftarrow R_K(Z \,\|\, x)$
for i in $\boldsymbol{X}$ do
  $a \leftarrow \boldsymbol{M}'[i]$
  if $a = 0 \wedge b < 0$ then return $\perp$
  $\boldsymbol{M}'[i] \leftarrow \boldsymbol{M}'[i] + b$
$\boldsymbol{M} \leftarrow \boldsymbol{M}'$
return $\langle \boldsymbol{M}, Z\rangle$

Figure 14: Keyed structure $\text{Count}[R, \ell, \lambda]$ given by $(\text{Rep}^R, \text{Qry}^R, \text{Up}^R)$ is used to define the ℓ-thresholded version of a counting filter. The parameters are a function $R : \mathcal{K} \times \{0,1\}^* \to [m]^k$ and integers $\ell, \lambda \geq 0$. A concrete scheme is given by a particular choice of parameters. The function w', used to determine if the filter is full, is defined in Section 2.1.

two different types of errors: false positives and false negatives. To be as general as possible, we define a parametrized error function δ for positive $\delta^+, \delta^- \in \mathbb{R}$ as

$$\delta(x,y) = \begin{cases} 0 & \text{if } x = y \\ \delta^+ & \text{if } x = 1, y = 0 \\ \delta^- & \text{if } x = 0, y = 1 \end{cases} \tag{28}$$

This means that false positives are given a weight of δ^+ while false negatives are given a weight of δ^-, and correct responses are given a weight of 0.

5.1 Insecurity of public counting filters

Unlike in the Bloom filter case, good security bounds cannot be achieved for a counting filter in the ERR-Pub setting, even if salts and/or private keys are used. The insecurity is due to the relative power of the **Up** oracle compared to the Bloom filter, which adds and subtracts numeric values stored in the structure rather than seeing only the effects of a bitwise-OR. Because of this, the adversary can mount an attack similar to the target-set coverage attack for a Bloom filter even if a PRF is used for hashing. First, the adversary calls $\mathbf{Rep}(\emptyset)$ to get an empty representation. The adversary can then call **Up** to insert an element into the set, see exactly what the outputs of each of the hash functions are, and then call **Up** again to delete the element. By doing this repeatedly, an adversary can determine the outputs of the PRF for u different inputs using $2u$ calls to **Up**. The combination of public representation with the insertion and deletion operations effectively provides an oracle for the secretly-keyed PRF. Once a sufficiently large number of PRF outputs has been determined, the adversary can construct the test and target set used for the target-set coverage attack (Section 4.1). The adversary then calls **Up** several more times to insert each element of the test set into the filter, and then each element of the target set will be overestimated.

In actual use, this specific attack may not be feasible for the adversary. However, as long as the filter is public, the adversary can easily determine the exact results of inserting or deleting any element just by seeing which counters are incremented or decremented. For this reason it is not enough that the function used to perform queries and updates is impossible for the adversary to simulate, since the adversary can build a lookup table just by watching the filter as it is updated. Instead, we must require that the filter itself be kept secret from the adversary.

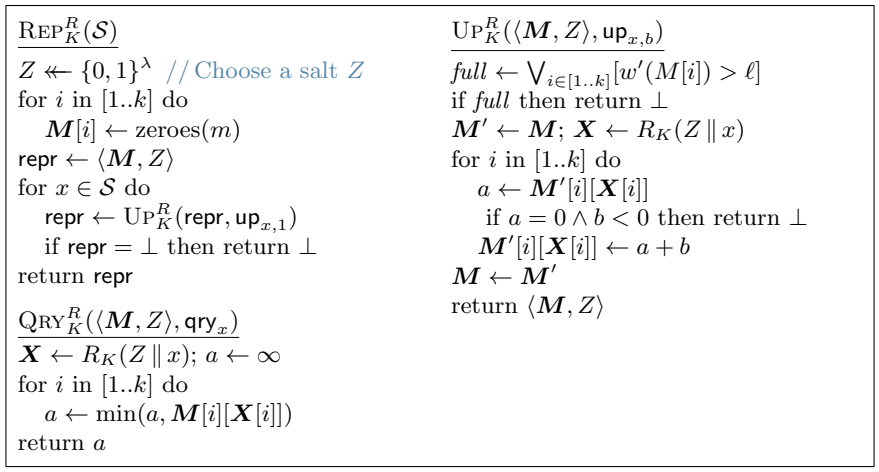

Figure 15: Keyed structure $\text{SKETCH}[R, \ell, \lambda]$ given by $(\text{REP}^R, \text{QRY}^R, \text{UP}^R)$ is used to define count min-sketch variants. The parameters are a function $R : \mathcal{K} \times \{0,1\}^* \to [m]^k$ and integers $\ell, \lambda \geq 0$. A concrete scheme is given by a particular choice of parameters. The function w', used to determine if the sketch is full, is defined in Section 2.1.

5.2 Security of private, ℓ-thresholded counting filters

Theorem 5. *Fix integers $q_R, q_T, q_U, q_H, q_V, r, t \geq 0$, let $p_\ell = ((\ell+1)/m)^k$, and let $r' = \lfloor r/\max(\delta^+, k\delta^-)\rfloor$. For all such $q_R, q_T, q_U, q_H, q_V, r$, and t, if $r' > p_\ell q_T$ then*

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(O(t), q_R, q_T, q_U, q_H, q_V) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + e^{r'-p_\ell q_T}\left(\frac{p_\ell q_T}{r'}\right)^{r'}\right],$$

where H is modeled as a random oracle.

The proof is similar to the case of count min-sketch (addressed in the next section), but has additional difficulties because the deletion operation is somewhat more complex to deal with in the case of a counting filter. In particular, because the filter contains single bits rather than counters, the adversary may be able to "delete" an element incorrectly believed to be in the set in order to induce false negatives. We therefore delay the proof until Appendix A. The proof is in fact quite similar to that of Theorem 6 due to similarity of the two structures, and in particular the fact that the allowed updates are the same.

So we find that, unlike Bloom filters, there is no simple tweak that can be performed to a counting filter to provide good ERR-Pub security bounds. In particular, it does not achieve security even in the immutable setting, and adding a secret key does not help. However, the bound above shows that in settings where filters can be assumed secret it is possible to prove an upper bound on the number of overestimates an adversary can cause. In particular, we recommend the combination of random per-representation salts and ℓ-thresholding in order to mitigate possible attacks in the ERR-Priv setting.

Due to the scaling factor of k that appears in r', false negatives impact the bound more than false positives, which indicates that applications seeking to minimize false negatives will require larger filters than those seeking to minimize false positives. This is distinctly different than in the non-adaptive setting, where false positives are much more common in counting filters than false negatives, and therefore much more relevant in determining the minimum size of the filter.

6 Count-Min Sketches

The count min-sketch (CMS) data structure is designed to concisely estimate the number of times a datum has occurred in a data stream. In other words, it is designed to estimate the frequency of each element of a multiset. The data structure is similar to a Bloom filter, but instead of a length-m array of bits it uses a k-by-m array of counters. It is designed to deal with streams of data in the non-negative turnstile model [8], which means the sketch accommodates both insertions and deletions but does not allow any entries to have

a negative frequency. Despite this, we will see that the two structures are closely related in terms of security properties. We show that ERR-Pub security is similarly impossible, but employing ℓ-thresholding allows for ERR-Priv security with a bound that is close to count min-sketch.

Non-adaptive error bound. The CMS is designed to minimize the number of elements whose frequencies are overestimated, while still allowing for reasonably low memory usage. For a function ρ and integer $\lambda \geq 0$, let $\textsc{Sketch}[\textsc{id}^\rho, n, \lambda] = (\textsc{Rep}^\rho, \textsc{Qry}^\rho, \textsc{Up}^\rho)$ as defined in Figure 15. If $\mathcal{S}$ is a multiset containing a total of n elements (counting duplicates as separate elements), i.e. $\mathcal{S} \in \mathrm{Func}(\{0,1\}^*, \mathbb{N})$ in our syntax, and $x \in \{0,1\}^*$ is any string, possibly but not necessarily a member of $\mathcal{S}$, we define the error probability as

$$P'_{k,m}(n) = \Pr\left[\rho \twoheadleftarrow \mathrm{Func}(\{0,1\}^*, [m]^k); \mathsf{repr} \twoheadleftarrow \textsc{Rep}^\rho(\mathcal{S}) : \textsc{Qry}^\rho(\mathsf{repr}, \mathsf{qry}_x) > \mathsf{qry}_x(\mathcal{S}) + \frac{en}{m} \mid \mathsf{repr} \neq \bot\right]. \tag{29}$$

(Here as above, e denotes the base of the natural logarithm.) Informally, $P'_{k,m}(n)$ is the probability that some x is overestimated by a non-negligible amount in the representation of some $\mathcal{S}$ containing a total of n elements, when a random function is used for hashing. Cormode and Muthukrishnan [8] show that this probability is bounded above by e^{-k}. This structure does not provide a bound for underestimation of frequencies, since it is designed for use cases where overestimates are considered harmful but underestimates are not.

Error function for frequency queries. The count min-sketch is designed for settings where overestimation in particular is undesirable, and so we aim to provide tight bounds on the size of overestimates but makes no guarantees about underestimates. To make the bounds simpler while staying conservative in our assumptions, we will use an error function that counts *any* overestimate as an error, not just overestimates larger than some lower bound of significance. In particular, we define δ as

$$\delta(x,y) = \begin{cases} 1 & \text{if } x > y \\ 0 & \text{otherwise.} \end{cases} \tag{30}$$

Note that other $\delta(x,y)$ may be preferable in some applications. For example, if the degree of error is significant, it may be desirable to use a δ which only counts as an error if x and y differ by more than some threshold value, or to use a function such as $\delta(x,y) = |x-y|$. In this paper we use this error bound because it is in some sense the most conservative, counting any overestimate as an error.

6.1 Insecurity of public sketches

The count min-sketch structure necessarily fails to satisfy ERR-Pub correctness for the same reasons as in the case of a counting filter. In particular, the adversary can call $\mathbf{Rep}(\emptyset)$ to receive an empty representation, insert an element x, observe which counters are incremented by this insertion, and then delete x. (Again, this is possible only because the sketch is public). By doing this repeatedly, the adversary can gain information about which elements overlap with which combinations of other elements, and can therefore mount the same attack described in Section 5.1.

6.2 Private, ℓ-thresholded sketches

Given the success of ℓ-thresholding in the case of Bloom filters, we continue using this tweak in the case of count min-sketches. Between thresholding and the use of a per-representation random salt, we are able to establish an upper bound on the number of overestimates in a count min-sketch. However, the bound is not quite as good as in the case of a salted and thresholded Bloom filter, which is unsurprising given the increased flexibility provided by the update algorithm coupled with the additional information returned by the query evaluation algorithm. Formally, we consider the structure given by $\Pi = \textsc{Sketch}[H, \ell, \lambda]$ for a hash function $H : \{0,1\}^* \to [m]^k$, which we will model as a random oracle.

Theorem 6 (ERR-Priv security of thresholded CMS)**.** *Let $p_\ell = ((\ell+1)/m)^k$. For all $q_R, q_T, q_U, q_H, q_V, r, t \geq 0$ it holds that*

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(O(t), q_R, q_T, q_U, q_H, q_V) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + e^{r' - p_\ell q_T}\left(\frac{p_\ell q_T}{r'}\right)^r\right],$$

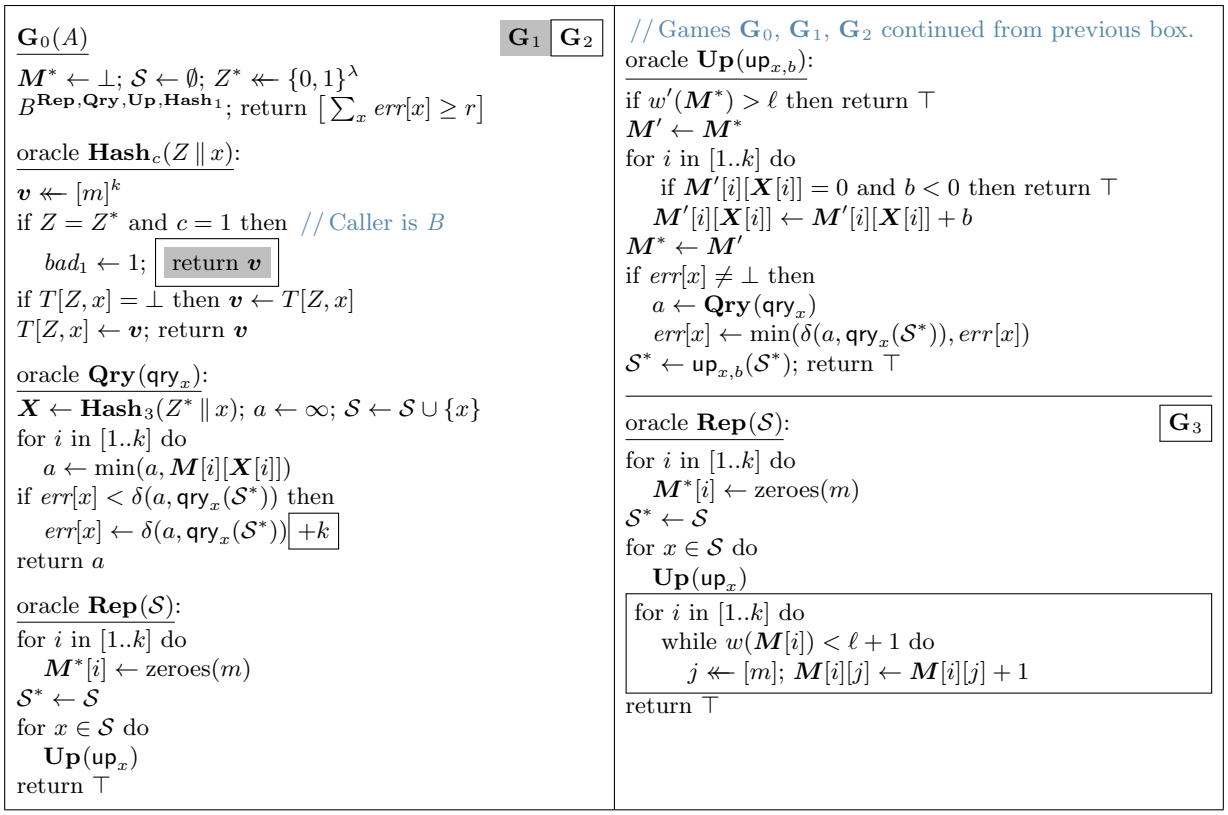

Figure 16: Games 0–3 for proof of Theorem 6.

where H is modeled as a random oracle, $r' = \lfloor r/(k+1) \rfloor$, and $r' > p_\ell q_T$.

The proof uses a game-playing argument in which we gradually whittle away at the flexibility the adversary has in performing repeated insertions, deletions, and queries to the same elements. The r' in place of r in the bound comes from the fact that, if the adversary finds that some x is overestimated, it may be able to produce as many as k additional overestimates by inserting x. We take this into account by automatically giving the adversary credit for all k additional overestimates as soon as it discovers the false positive. After taking this into account, we can reduce to the standard binomial argument in which the adversary seeks to find r' overestimates by making arbitrary queries.

Proof of Theorem 6. We derive a bound in the ERR-Priv1 case and then use Lemma 1 to move from ERR-Priv1 to the more general ERR-Priv case. Because we are in the ERR-Priv1 case, we may assume without loss of generality that the adversary does not call **Reveal**, since revealing the only representation automatically prevents the adversary from winning.

We begin with the game $\mathbf{G}_0$ as shown in Figure 16, which has identical behavior to the ERR-Priv1 experiment for Π. As usual, we have a bad_1 flag that gets set if the adversary ever calls $\mathbf{Hash}_1$ with the actual salt used by the representation. By an almost identical argument, we can move to $\mathbf{G}_1$, where the behavior is different only when the bad_1 flag is set, with a bound of

$$\Pr[\,\mathbf{G}_0(A) = 1\,] \leq q_H/2^\lambda + \Pr[\,\mathbf{G}_1(A) = 1\,]\ . \tag{31}$$

The key differences between this proof and the Bloom filter proof are the more complex response space of **Qry** ($\mathbb{N}$ rather than $\{0,1\}$) and the fact that both elements of $\mathcal{S}$ and non-elements of $\mathcal{S}$ may produce errors.

As a first step in dealing with the **Up** oracle, we want to show that deletion is never helpful for the adversary. So, for any A, we construct an adversary B that simulates A, forwarding all oracle queries in the natural way, except that it ignores any $\mathbf{Up}(\mathsf{up}_{x,-1})$ calls, i.e., any deletions. Because deleting x does

not change whether x is overestimated or not, ignoring deletions does not affect whether later calls of the form $\mathbf{Qry}(\mathsf{qry}_x)$ will produce an error. Furthermore, if $y \neq x$, then the probability of $\mathbf{Qry}(\mathsf{qry}_y)$ causing an error can only increase if x's deletion is ignored, since the deletion of x decreases counter values without decreasing the true frequency of y. Therefore $\Pr[\,\mathbf{G}_1(A) = 1\,] \leq \Pr[\,\mathbf{G}_1(B) = 1\,]$, and we have reduced to the case of an adversary whose **Up** calls only consist of insertions.

Next, we move from B to an adversary C that never inserts an element more than once. Similarly to the previous step, C simulates B, tracking the elements of $\mathcal{S}$ and forwarding B's oracle queries in the natural way, except that any **Up** queries to insert an element already present in $\mathcal{S}$ are ignored. First, inserting x does not change whether x is overestimated or not, so C ignoring the re-insertion does not affect whether later $\mathbf{Qry}(\mathsf{qry}_x)$ calls will produce an error. For $y \neq x$, the fact that B makes no deletions is key. The value of the counters associated with y by the hash functions must be at least equal to the true frequency of y, and $\mathbf{Qry}(\mathsf{qry}_y)$ will find an overestimate if these counters are all strictly greater than the true frequency. Since updates are deterministic, re-inserting x can only increment the same counters that were incremented by the original insertion of x, and so this re-insertion cannot cause y to become overestimated if it was not already. So all **Qry** calls are just as likely to produce an error for C as they are for B, and $\Pr[\,\mathbf{G}_1(B) = 1\,] = \Pr[\,\mathbf{G}_1(C) = 1\,]$.

As a third step, we move from $\mathbf{G}_1$ to a $\mathbf{G}_2$ where the adversary gains $k+1$ 'points' for finding a query which produces an overestimate, but which prevents the adversary from querying elements of $\mathcal{S}$. These extra points are necessary because, unlike in the case of a Bloom filter, inserting an overestimated element x can cause other elements of $\mathcal{S}$ to become overestimated. In particular, if one of the counters incremented by the insertion of x is shared with an element of $\mathcal{S}$ that is not overestimated, that element may become overestimated. However, if that counter is shared with multiple elements of $\mathcal{S}$, that counter is already an overestimate for all of the elements associated with it, and so no more than one overestimate can be caused per counter incremented by the insertion of x. Since inserting x increments k counters, at most k errors can be caused in this way. For any adversary C for $\mathbf{G}_2$, we can construct D for $\mathbf{G}_3$ that simulates C perfectly except that it ignores any oracle calls that would insert these elements. Since D already gets credit equal to the maximum number of errors these insertions could cause in addition to the credit for the original overestimate, D accumulates at least as many errors as C does, and so $\Pr[\,\mathbf{G}_1(C) = 1\,] \leq \Pr[\,\mathbf{G}_2(D) = 1\,]$.

We are now dealing with an adversary that gains points when it finds any overestimate, but which only makes queries to $x \notin \mathcal{S}$. This means that an error is simply a query that returns a value greater than 1. Analogously to the proof of Theorem 4, we now move to a game $\mathbf{G}_3$ where **Rep** randomly fills the sketch to capacity after inserting the elements of $\mathcal{S}$, so that each row has $\ell + 1$ non-zero counters. For any D for $\mathbf{G}_2$ we construct E for $\mathbf{G}_3$ that simulates D, forwarding **Rep**, **Qry**, and $\mathbf{Hash}_1$ calls but ignoring **Up** calls. Analogously to Theorem 4, adversary E achieves at least the same advantage as D by having a maximally full sketch as soon as **Rep** is called, and so $\Pr[\,\mathbf{G}_2(D) = 1\,] \leq \Pr[\,\mathbf{G}_3(E) = 1\,]$.

The probability of E winning can now be given by another binomial bound. The set of nonzero counters in each row is a uniformly random subset of $[m]$ of size $\ell+1$. Since any query returning a nonzero value is a success for the adversary, the probability of any particular **Qry** call causing a collision within a single row i is $(\ell+1)/m$, and the probability of a collision in every row (i.e. an error) is $((\ell+1)/m)^k$. The adversary has a total of q_T attempts, and wins if it accumulates $\lfloor r/(k+1) \rfloor$ successes, since each error gives it $k+1$ points. So, letting $p_\ell = ((\ell+1)/m)^k$ and $r' = \lfloor r/(k+1) \rfloor$, we have

$$\Pr[\,\mathbf{G}_3(E) = 1\,] \leq \sum_{i=r'}^{q_T} \binom{q_T}{i} p_\ell^i (1-p_\ell)^{q_T - i}\,. \tag{32}$$

Applying the usual Chernoff bound and applying Lemma 1 turns this into the final bound of

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(A) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + e^{r' - p_\ell q_T} \left(\frac{p_\ell q_T}{r'}\right)^{r'}\right]. \tag{33}$$

□

6.3 Discussion

The results for count min-sketches are similar to the results for counting filters, as might be expected given the similarities in terms of both the supported updates and the structure of the representations themselves

(any count min-sketch can be transformed into a counting filter by adding all the rows together element-wise.) In particular, we see that count min-sketches which are publicly visible cannot provide good security guarantees. This means that sketches intended for a security-sensitive setting should be kept hidden from potential adversaries. Furthermore, our bound relies on per-representation random salts and ℓ-thresholding, so these changes should also be taken into account when constructing secure count min-sketches. The size increase of the sketches is comparable to the size increase of counting filters, but does not need to take into account multiple types of errors

The bound we achieve is based on the same binomial bound as in the case of Bloom filters, but has a notable difference in the form of r' replacing r. This negatively impacts the amount of space the filter must take up in order to provide low error bounds, but because the scaling factor between r and r' is only $k+1$, the difference should not be unacceptably extreme given reasonable parameter choices. We also note that it is possible this bound can be improved to reduce the impact on sketch size, since the initial factor of q_R does not have an obvious attack associated with it which would make this bound tight. (The same is true, of course, of Bloom and counting filters.)

References

[1] Bellare, M., and Rogaway, P. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security* (New York, NY, USA, 1993), CCS '93, ACM, pp. 62–73.

[2] Bellare, M., and Rogaway, P. The security of triple encryption and a framework for code-based game-playing proofs. In *EUROCRYPT 2006: Proceedings of the 24th Annual International Conference on The Theory and Applications of Cryptographic Techniques* (2006).

[3] Bellovin, S. M., and Cheswick, W. R. Privacy-enhanced searches using encrypted Bloom filters. Cryptology ePrint Archive, Report 2004/022, 2004. `http://eprint.iacr.org/2004/022`.

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

[5] Broder, A., and Mitzenmacher, M. Network applications of Bloom filters: A survey. *Internet Mathematics 1*, 4 (2004).

[6] Byers, J. W., Considine, J., Mitzenmacher, M., and Rost, S. Informed content delivery across adaptive overlay networks. *IEEE/ACM Trans. Netw. 12*, 5 (2004).

[7] Chazelle, B., Kilian, J., Rubinfeld, R., and Tal, A. The Bloomier filter: An efficient data structure for static support lookup tables. In *SODA 2004: Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms* (2004).

[8] Cormode, G., and Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *Journal of Algorithms 55*, 1 (2005).

[9] Crosby, S. A., and Wallach, D. S. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th USENIX Security Symposium* (2003).

[10] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008).

[11] Deng, F., and Rafiei, D. Approximately detecting duplicates for streaming data using stable bloom filters. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data* (2006), ACM, pp. 25–36.

[12] Dietzfelbinger, M., and Pagh, R. Succinct data structures for retrieval and approximate membership (extended abstract). In *ICALP 2008: Proceedings of the 35th International Colloquium on Automata, Languages and Programming* (2008).

[13] Durand, M., and Flajolet, P. Loglog counting of large cardinalities. In *ESA 2003: Proceedings of the 11th Annual European Symposium on Algorithms* (2003).

[14] Fan, B., Andersen, D. G., Kaminsky, M., and Mitzenmacher, M. D. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies* (2014).

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

[16] Feng, W., Kandlur, D. D., Saha, D., and Shin, K. G. Stochastic fair blue: A queue management algorithm for enforcing fairness. In *INFOCOM 2001: Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Society* (2001).

[17] Fredman, M. L., Komlós, J., and Szemerédi, E. Storing a sparse table with 0(1) worst case access time. *Journal of the ACM 31*, 3 (1984).

[18] Gerbet, T., Kumar, A., and Lauradoux, C. The power of evil choices in Bloom filters. In *Proceedings of the 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (2015).

[19] Gervais, A., Capkun, S., Karame, G. O., and Gruber, D. On the privacy provisions of Bloom filters in lightweight Bitcoin clients. In *ACSAC 2014: Proceedings of the 30th Annual Computer Security Applications Conference* (2014).

[20] Kirsch, A., and Mitzenmacher, M. Less hashing, same performance: Building a better Bloom filter. *Random Structures and Algorithms 33*, 2 (2008).

[21] Larisch, J., Choffnes, D., Levin, D., Maggs, B. M., Mislove, A., and Wilson, C. CRLite: A scalable system for pushing all TLS revocations to all browsers. In *The Proceedings of the 38th IEEE Symposium on Security and Privacy.* (2017).

[22] Lipton, R. J., and Naughton, J. F. Clocked adversaries for hashing. *Algorithmica 9*, 3 (1993).

[23] Mironov, I., Naor, M., and Segev, G. Sketching in adversarial environments. *SIAM Journal on Computing 40*, 6 (2011).

[24] Naor, M., and Yogev, E. Bloom filters in adversarial environments. In *CRYPTO 2015: Proceedings of the 35th Annual Cryptology Conference* (2015).

[25] Nojima, R., and Kadobayashi, Y. Cryptographically secure Bloom-filters. *Transactions on Data Privacy 2*, 2 (2009).

[26] Reynolds, P., and Vahdat, A. Efficient peer-to-peer keyword searching. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware* (2003).

[27] Schnell, R., Bachteler, T., and Reiher, J. A novel error-tolerant anonymous linking code, 2011. Working paper series no. WP-GRLC-2011-02, German Record Linkage Center.

[28] Tarkoma, S., Rothenberg, C., and Lagerspetz, E. Theory and practice of Bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials 14*, 1 (2012).

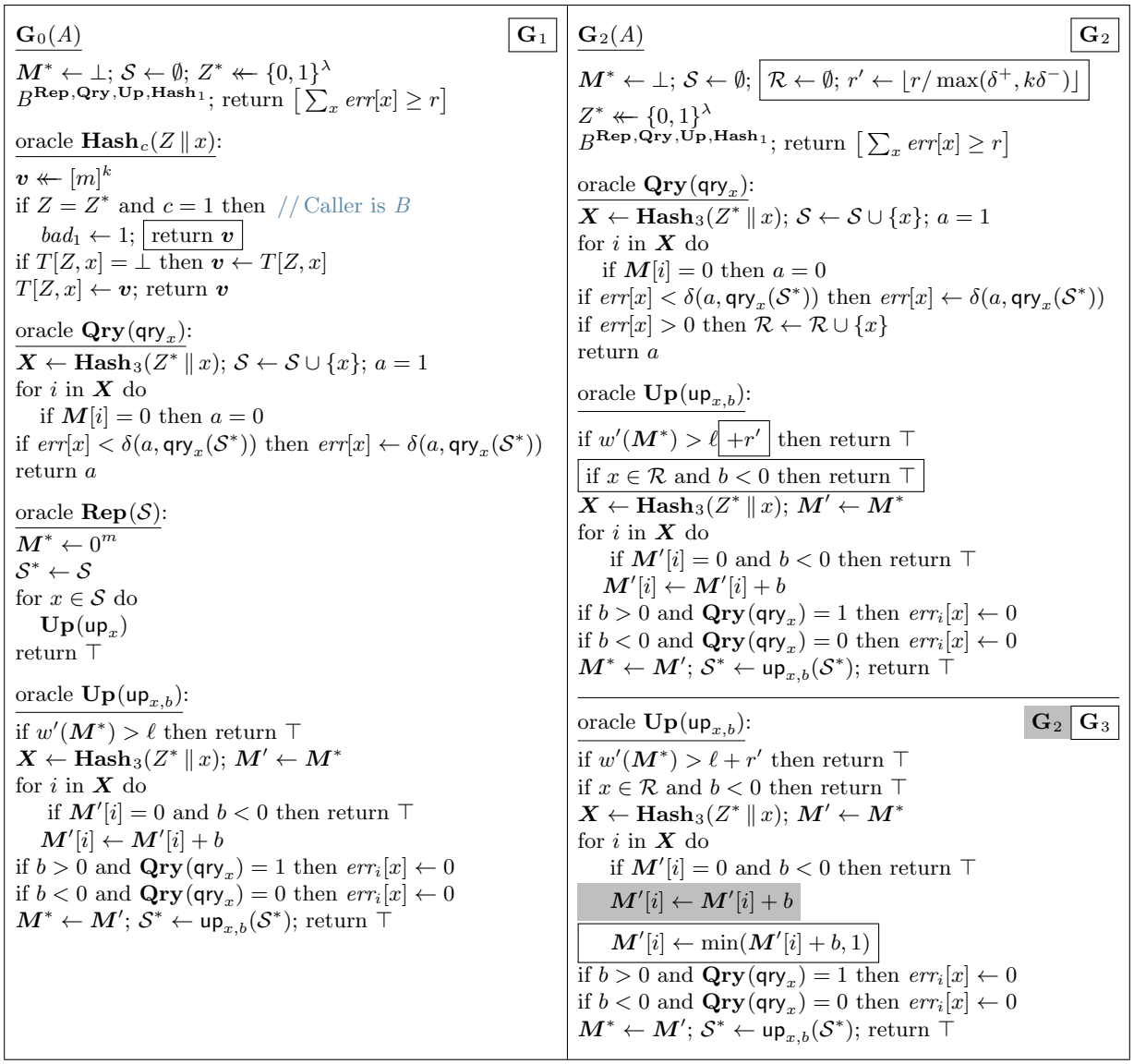

Figure 17: Games 0–3 for proof of Theorem 5.

A Proof of Theorem 5

As with the proof of Theorem 1, we derive a bound in the ERR-Priv1 case and then use Lemma 1 to move from ERR-Priv1 to the more general ERR-Privcase. Because we are in the ERR-Priv1 case, we may assume without loss of generality that the adversary does not call **Reveal**, since revealing the only representation automatically prevents the adversary from winning.

We begin with a game $\mathbf{G}_0$, shown in Figure 17, which has identical behavior to the ERR-Priv1 experiment for a counting filter. As in the proof of Theorem 1, we have a bad_1 flag that gets set if the adversary ever calls $\mathbf{Hash}_1$ with the actual salt used by the representation. By a very similar argument, we can move to $\mathbf{G}_1$, where the behavior is different only when the bad_1 flag is set, with a bound of

$$\Pr[\,\mathbf{G}_0(A) = 1\,] \leq q_H/2^\lambda + \Pr[\,\mathbf{G}_1(A) = 1\,]\ . \tag{34}$$

Note that if x is found to be a false positive, deleting x may cause up to k elements of $\mathcal{S}$ to become false negatives. We therefore move to a game $\mathbf{G}_2$ where the adversary gets credit for either a single false positive or for k false negatives whenever it finds a false positive, but where the adversary cannot delete

any false positives that it finds. We let $r' = \lfloor r/\max(\delta^+, k\delta^-) \rfloor$ represent the number of false positives the adversary has to find in $\mathbf{G}_2$ in order to win. In order to prevent the adversary from being penalized by the filter becoming full too early, we also raise the threshold from ℓ to $\ell + r'$ in $\mathbf{G}_2$. Now for any A for $\mathbf{G}_1$, we can construct B for $\mathbf{G}_2$ that simulates A, keeping track of all query responses and forwarding all oracle queries in the natural way, except that calls to delete false positives are ignored. Since **Up** never fails for B due to the increased threshold, and since B gets automatic credit for any false negatives that might have been caused by deleting false positives, B succeeds whenever A does, i.e. $\Pr[\,\mathbf{G}_1(A) = 1\,] \le \Pr[\,\mathbf{G}_2(B) = 1\,]$.

Since the remaining deletions do not cause errors, we can use the same argument as in the later proof of Theorem 6 to reduce from B to an adversary C which does not make deletions at all. In $\mathbf{G}_3$, we further reduce from a counting filter to a normal Bloom filter by capping each of the counters in the filter at 1. Since no deletions are performed, a counter in $\mathbf{G}_3(C)$ is nonzero if and only if the same counter in $\mathbf{G}_2(C)$ is nonzero. So **Qry** behaves the same in $\mathbf{G}_3$ as it did in $\mathbf{G}_2$, and $\Pr[\,\mathbf{G}_2(C) = 1\,] \le \Pr[\,\mathbf{G}_3(C) = 1\,]$.

Note that $\mathbf{G}_3$ is actually simulating an ordinary Bloom filter, since all 'counters' in the filter are restricted to the range $\{0, 1\}$, there are no deletions, and any insertions just set the corresponding bits to 1. In fact, this game is identical to $\mathbf{G}_2$ in the proof of Theorem 4 except that the adversary need only accumulate r' errors instead of r errors and the threshold is $\ell + r'$ instead of ℓ. An identical argument allows us to reach the binomial bound of

$$\Pr[\,\mathbf{G}_3(C) = 1\,] \le \sum_{i=r'}^{q_T} \binom{q_T}{i} p_\ell^i (1 - p_\ell)^{q_T - i}\,, \tag{35}$$

where p_ℓ is now defined to be $((\ell + k + r')/m)^k$. Then the standard Chernoff bound, along with Lemma 1, yields the final bound of

$$\mathbf{Adv}^{\text{err-priv}}_{\Pi,\delta,r}(A) \le q_R \cdot \left[\frac{q_H}{2^\lambda} + e^{r' - p_\ell q_T} \left(\frac{p_\ell q_T}{r'} \right)^{r'} \right]. \tag{36}$$