Correctness and Privacy of Mutable Data Structures

Anonymous Author(s)

ABSTRACT

(0.1: TS says: OLD ABSTRACT) This work initiates the study of abstract data structures from a cryptographic perspective. We first establish a precise syntax that captures a broad class of real-world data structures. We then treat the *correctness* and *privacy* of data structures as security properties, and establish formal security notions for each. Loosely, our notion of correctness captures an (adaptive) adversary's ability to cause a data structure to err in the course of responding to a set of supported queries, and our two privacy notions neatly capture what a data structure leaks about the data it represents.

We use our formalisms to explore the security of the widely used Bloom filter [**?**] and some important variants. We find, for example, that the security of Bloom filters depends crucially on whether or not the underlying hash functions are known by the adversary prior to the filter being constructed. We also study a real-world mechanism for privacy-preserving record linkage (over hospital databases). Our notions provide a crisp view of the (in)security of this Bloom-filter-based mechanism. To demonstrate the broader applicability of our definitions, we move from data structures supporting set-membership queries (only) to dictionary data structures. Concretely, we analyze the "Bloomier filter" [**?**], which provides a compact representation of a key/value store.

CCS CONCEPTS

• **Security and privacy** → Use https://dl.acm.org/ccs.cfm to generate actual concepts section for your paper;

KEYWORDS

template; formatting; pickling

(0.2: TS says: Intro structure, thoughts...)

- Point to NY, quickly pivot to mutable DS
- Our syntax and what it captures
- Our correctness and what it captures (don't forget: immutability as an adversarial restriction, not a syntactic one)
- Loosely speaking, what the results say (see picture of chalkboard) – rough "structure" of all results: non-adaptive term, plus "(informed) guessing" term)
- Data structures we consider: BF, counting filter, cuckoo filter, CMS,... what about Bloomier filter results from old paper?
- future directions

1 INTRODUCTION

Data structures are fundamental to essentially all areas of computer science. 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 [**?**], 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 dramatically degrade the performance of real systems [**? ? ?**].

Naor and Yogev (NY) formalized a notion of adversarial correctness for Bloom-filter-like structures. (Recall that a Bloom filter provides a compact representation of a set $\mathcal{S}$, and supports set-membership queries.) In their notion, the adversary must choose $\mathcal{S}$ all at once, and then adaptively queries the data structure in order to induce erroneous responses. But many modern applications require data structures to (compactly) represent *streams* of data, not static collections (typicallly, sets) whose size is bounded by some number that is fixed *a priori*. For example, the count-min sketch structure [**?**] is used to compactly represent the number of times a particular object x has appeared, so far, in the input stream. This is done by incrementing a collection of counters that are associated with x (deterministically, via hash functions) each time x is seen. Some applications require the ability to add *and remove* elements from effective $\mathcal{S}$ that the structure represents.

As such, we significantly expand upon NY to consider data structure correctness in the presense of adversaries that adaptively create and mutate $\mathcal{S}$, in an effort to induce query-response errors. We define a syntax for mutable data structures, formalize two notions of correctness (capturing settings in which representations are public/private, respectively), and exercise these by considering a collection of real-world data structures: Bloom filters [**?**], counting filters [**?**], cuckoo filters [**?**], and count-min sketches [**?**]. (1.1: TS says: Update if we add more.) In addition to the basic version of these, we explore variations that allow for "salted" representations (i.e., per-representation randomness), secret keys (so that the mapping from $\mathcal{S}$ to representation is unpredictable, even if the representation of one or more closely related $\mathcal{S}'$ are known), or both.

The notion of query-response error becomes considerably more general in our setting. The ubiquitous Bloom filter, designed to compactly represent a fixed set $\mathcal{S}$ and to provide approximately

Anonymous Submission to ACM CCS 2019, Due 15 May 2019, London, TBD
2019. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

	Public Representation				Private Representation			
	∅	salt	key	salt+key	∅	salt	key	salt+key
(static) Bloom filter	✗							
Bloom filter	✗	✗	✗	?	✗	Thm 1	✗	Thm 2
Counting filter	✗							
Cuckoo filter	✗							
Count-min sketch	✗							

Figure 1: Summary of results. An entry of '✗' means there is a (query efficient) attack, which we discuss in the body. An entry of 'Thm n' means that we explicitly prove a security bound for the structure in Theorem n. An entry of '✓' means that the structure is secure, but we do not give an explicit result in this submission. An entry of '?' means we do not address this case.

correct responses to set-membership queries, suffers only false-positive errors, i.e., saying that $y \in \mathcal{S}$ when it isn't. Counting filters — loosely, Bloom filters with an array of counters rather than single bits — support updates of the form *remove x from the representation of $\mathcal{S}$*. If $x \in \mathcal{S}$ when this update is processed, then the appropriate counters are decremented. If $x \notin \mathcal{S}$, however, counters will be erroneously decremented, leading to the possibility of false-*negative* errors when set-membership queries are made for some $y \in \mathcal{S}$. False-positive errors remain, too. More generally, the count-min sketch supports queries of the form *how many times has x appeared, so far?* One could define error in an absolute sense, i.e., the data structure errs if its response is not exactly correct. But even in the non-adaptive setting, traditional analyses guarantee only that the response will be *close* to the exact number, with some probability that is close to one. Thus, our security notions are parameterized by a error-cost function d: if the correct response to a query is a and the data structure responds with a', the cost of the error is $d(a, a') \geq 0$. They are also parameterized by an total-cost threshold, and the adversary is considered to win if the total cost of induced errors is greater than this value. As we will see, calculating the total cost is not straightforward; in particular, determining whether or not the cost of a given error should be carried across (adaptive, adversarial) updates to $\mathcal{S}$ and its representation.

(1.2: TS says: old stuff below here)

Data structures and their correctness. We formalize a data structure as a triple of algorithms (Rep, Qry, Up) denoting the *representation*, *query-evaluation*, and *Update* algorithms, respectively. Associated to the data structure is a set of supported queries Q. The representation algorithm is randomized, taking as input a key K and a collection of data $\mathcal{S}$, and returning a representation repr of $\mathcal{S}$. (To capture unkeyed data structures, one sets $K = \varepsilon$.) The deterministic query-evaluation algorithm Qry uses K and repr in order to respond to a requested query $\mathsf{qry} \in Q$ on $\mathcal{S}$. [[...]]

For better efficiency, many data structures only approximately represent the collection $\mathcal{S}$. In this case, the query-evaluation algorithm Qry may err in its response to queries. Roughly speaking, our notion of adversarial correctness (ERR-UP) captures how difficult it is for an attacker (given repr) to find $r > 0$ distinct queries on which Qry returns an incorrect answer.

For Bloom filters, the representation repr includes a bit array M that represents a set $\mathcal{S} \subseteq \mathcal{X}$ using hash functions $h_1, \ldots, h_k$. The supported queries are the predicates $\{\mathsf{qry}_x\}_{x \in \mathcal{X}}$, where $\mathsf{qry}_x(\mathcal{S}) =$ 1 iff $x \in \mathcal{S}$. It is well known that Bloom filters may have false positives, and their false-positive rate for *independently chosen* inputs and queries is well understood. (See Appendix B.) Our correctness notion quantitatively captures the error rate even in the presence of an attacker that adaptively attempts to induce errors. [[...]]

We note that Naor and Yogev [?] were the first to formalize adversarial correctness of Bloom filters and, indeed, their work provided inspiration for this paper. Our work significantly extends theirs in several ways, as we will detail, shortly. [[...]]

Constructions we analyze. We put our syntax and security notions to work in several case studies. The brief description of Bloom filters given above was silent as to how the hash functions $h_1, \ldots, h_k$ are chosen, and whether or not they are public. In fact, these details have a significant effect on what notions of security the resulting structure satisfies:

- (Section ??) If the hash functions are fixed and known to the attacker prior to the filter being constructed, the data structure offers neither correctness nor privacy for any practically interesting parameters. We show this by exhibiting explicit attacks and analyzing their performance.
- (Section ??) If *salted* hash functions are used, and the adversary is given the salt only after the collection $\mathcal{S}$ is chosen, then the structure can achieve the same correctness guarantees in the adversarial setting as do Bloom filters in the traditional non-adversarial setting. We also show that this structure achieves our privacy notion of one-wayness.
- (Section ??) We explore a natural, keyed variant of a Bloom filter in which the hash functions are derived from a secretly keyed pseudorandom function. (This is similar to a construction proposed by Naor and Yogev [?].) We show that this variant enjoys simulation-based privacy, as well as a tighter security bound for correctness than the salted Bloom filter.

Our particular realization of the salted and secretly keyed Bloom filters leverages results from Kirsch and Mitzenmacher [?] that allow one to effectively implement $h_1, \ldots, h_k$ by making only two *actual* evaluations of an underlying hash function or PRF, respectively. In addition to the comprehensive analysis of Bloom filters described above, we also apply our definitions to:

- (Section ??) A keyed structure for privacy-preserving record linkage introduced by Schnell *et al.* [?], and subsequently

attacked by Niedermeyer *et al.* [**?**]. In our framework we are able to show precisely how their scheme breaks down.

- (Section **??**) A dictionary proposed by Charles and Chellapilla [**?**] that stores a set of n key/value pairs, where the keys are arbitrary bitstrings and the values are of length at most m, using just $O(mn)$ bits.

Future research directions.

Our goal is to establish foundations for the security of data structures. But it would certainly be interesting to analyze high-level protocols that use these data structures, e.g. content-distribution networks [**?**], where many servers propagate representations of their local cache to their neighbors. The Bloom filter family alone has a wide range of practical applications, for example in large database query processing [**?**], routing algorithms for peer-to-peer networks [**?**], protocols for establishing linkages between medical-record databases [**?**], fair routing of TCP packets [**?**], and Bitcoin wallet synchronization [**?**]. Analyzing higher-level primitives or protocols will require establishing appropriate syntax and security notions for those, too; hence we leave this for future work.

Comparison with Naor and Yogev. As previously noted, Naor and Yogev [**?**] 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), and consider privacy in addition to correctness. Even with respect to the specific case of correctness for set-membership structures, our work offers several advantages as compared to the Naor-Yogev treatment:

- Our syntax distinguishes between the (secret) key and the public portion of a data structure, an important distinction that is missing in their work.
- The Naor-Yogev 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.
- 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.

We give a more technical comparison of our correctness notion and theirs in Appendix A.

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, and each of these structures has its own correctness and privacy characteristics [**? ? ? ? ? ? **].

We have already mentioned the ubiquity of Bloom filters in support of efficient network communication and computing protocols. They also find use in security-critical environments, including spam filters, (distributed) denial-of-service attack detection, and deep packet inspection [**?**]. Recently, Bloom filters were proposed as a means of efficient certificate-revocation list (CRL) distribution [**?**], a crucial component of public-key infrastructures. To our knowledge, ours is the first work to propose general notions of *privacy* for abstract data structures. However, a 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 [**? ?**]. (1.3: Jon says: History-independent data structures seem relevant to mention.) (1.4: Chris says: What does this mean?)

Correctness of data structures in adversarial settings is well-motivated in the security literature and in practice. Perhaps the earliest published attack on the correctness of a data structure was due to Lipton and Naughton [**?**] 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 [**?**] 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.* [**?**] 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 [**?**]), 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.

Finally, we note that the dictionary construction considered in Section **??** bares resemblance (at least structurally) to *garbled Bloom filters*, a tool used recently for efficient private-set intersection [**? ? **].

2 SYNTAX

Preliminaries. (2.1: Any to-do: Fill in with usual things.)

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

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

Allowing each of the algorithm to take a key K lets us separate (in our security notions) any secret randomness that is across data structure operations, from per-operation randomness (e.g., salts). 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 [**?**] and the Stable Bloom filter [**?**].

Some important data structures admit update operations that are "inverses" of each other. For example, the count-min sketch structure supports the addition and deletion of elements from the underlying (multi)set that it represents. Thus, we say that Π is *invertible* if, for every representation repr , update $\mathsf{up} \in \mathcal{U}$, and key $K \in \mathcal{K}$, there is $\mathsf{up}' \in \mathcal{U}$ such that $\Pr[\ \textsc{Up}(K, \textsc{Up}(K, \mathsf{repr}, \mathsf{up}), \mathsf{up}') = \mathsf{repr}\ ] = 1$.

The query algorithm Qry is formalized as 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

We define two adversarial notions of correctness given by a pair of related experiments for a mutable data structure Π, error function $d : \mathcal{R}^2 \to [0, \infty)$, and error capacity r. The vales $d(x, y)$ 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 $d(x, x) = 0$ for all x, but otherwise place no restrictions on what the error function might look like.

In the first correctness notion, the representations of the true data are public (ERR-UP), and one in which they are private (ERR-UP(S)). We will describe the former, as the latter is a closely related derivative. Throughout, we assume that the adversary A does not make any pointless queries.

While many potential errors may exist for a given representation of a set, in the form of queries qry that would return an inaccurate answer when sent to **Qry**, we want an experiment that forces the adversary to actually find specific erroneous queries. In particular, we only give the adersary credit when they actually make a **Qry** call that produces an error.

Both experiments aim to capture the total weight of the errors caused by the adversary, at any point in time, with respect to 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. Specifically, for each $\mathcal{S}_i$, both experiments maintain a set C_i that is initially set to empty (when $\mathcal{S}_i$ is first assigned via a **Rep**-query), and it is reset to empty whenever $\mathcal{S}_i$ is updated via an **Up**-query. This is capturing the fact that applying a non-empty update function up to $\mathcal{S}_i$ produces a new data object.

To track errors, both experiments maintain an array $\mathit{err}_i[]$ for every data object $\mathcal{S}_i$ that has been defined. Initially, $\mathit{err}_i[]$ is implicitly assigned the value of `undefined` at every index. For purposes of value comparison, we adopt the convention that $\texttt{undefined} < n$ for all $n \in \mathbb{R}$. Now, the array err_i is indexed by query functions qry, and the value of $\mathit{err}_i[\mathsf{qry}]$ 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 $\mathit{err}_i[\mathsf{qry}]$ is updated within the **Qry**- and **Up**-oracles, but observe that $\mathit{err}_i[\mathsf{qry}] = \texttt{undefined}$ 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 $\mathit{err}_i[\mathsf{qry}]$ is overwritten iff the error caused by qry is larger than the existing value of $\mathit{err}_i[\mathsf{qry}]$. The first time (i, qry) is queried to **Qry** this is guaranteed, since the minimum possible value of d is 0. Since the set C_i is used to prevent (WLOG) the adversary from repeating a query $\mathbf{Qry}(i, \mathsf{qry})$ for a given $\mathcal{S}_i$, increases to the value of $\mathit{err}_i[\mathsf{qry}]$ may only be made "across" updates to $\mathcal{S}_i$. This may seem to be overly conservative, as an error-heavy $\mathcal{S}_i$ may become less so after an update. But we account for this within the **Up**-oracle. In particular, calls to the **Up**-oracle may only *decrease* the value of $\mathit{err}_i[\mathsf{qry}]$.

When a query $\mathbf{Up}(i, \mathsf{up})$ is made, the oracle first updates the data object $\mathcal{S}_i$ and its corresponding representation. The set C_i is reset to empty, because the data object $\mathcal{S}_i$ is "new" again. Now, for each defined value $\mathit{err}_i[\mathsf{qry}]$, we reevaluate the error that *would* be caused by the previously asked qry w.r.t. the newly updated $\mathcal{S}_i$ and repr_i. If the existing value of $\mathit{err}_i[\mathsf{qry}]$ is larger than the error that qry would cause to w.r.t. the newly updated $\mathcal{S}_i$ and repr_i, then we overwrite $\mathit{err}_i[\mathsf{qry}]$ with the smaller value. Doing so insures that the array err_i does not overcredit the attacker for errors against the current data object and representation.

For a concrete example of why these choices are necessary, consider a representation repr of the set $\mathcal{S} = \{1, 2, 3\}$ in some structure that supports set membership queries. Suppose an adversary learns that 4 is a false-positive value for repr. If the adversary later uses an **Up** query to add 4 to $\mathcal{S}$, this should no longer count as a false positive. Our definition ensures that these known false positives are checked for and no longer counted if added to the set.

Given the experiments defined here, we define the advantage of an adversary A as the probability it succeeds at the experiment, i.e. $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) = \Pr\left[\ \mathbf{Exp}^{\text{err-up}}_{\Pi,r}(A) = 1\ \right]$ in the public-representation case and $\mathbf{Adv}^{\text{err-up(s)}}_{\Pi,r}(A) = \Pr\left[\ \mathbf{Exp}^{\text{err-up(s)}}_{\Pi,r}(A) = 1\ \right]$ in the private-representation case. For constants t, q_R, q_T, q_U, q_H, we define $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(t, q_R, q_T, q_U, q_H)$ to be the maximum advantage attained by an adversary running in t time steps and making q_R calls to **Rep**,

Structure	Data Objects	Supported Queries	Supported Updates	Parameters
Bloom filter	Sets $\mathcal{S} \subseteq \mathcal{X}$	$\mathsf{qry}_x(\mathcal{S}) = [x \in \mathcal{S}]$	$\mathsf{up}_x(\mathcal{S}) = \mathcal{S} \cup \{x\}$	n, max $\lvert\mathcal{S}\rvert$ k, # hash functions m, array size (bits)
Counting filter	Multisets $\mathcal{S} \in \mathrm{Func}(\mathcal{X}, \mathbb{N})$	$\mathsf{qry}_x(\mathcal{S}) = [\mathcal{S}(x) > 0]$	$\mathsf{up}_{x,0}(\mathcal{S})(x) = \mathcal{S}(x) + 1$ $\mathsf{up}_{x,1}(\mathcal{S})(x) = \mathcal{S}(x) - 1$ $\mathsf{up}_{x,b}(\mathcal{S})(y) = \mathcal{S}(y)$ for $x \neq y$	n, max $\lvert\mathcal{S}\rvert$ k, # hash functions m, array size (counters) d, bits per counter
Cuckoo filter	Multisets $\mathcal{S} \in \mathrm{Func}(\mathcal{X}, \mathbb{N})$	$\mathsf{qry}_x(\mathcal{S}) = [\mathcal{S}(x) > 0]$	$\mathsf{up}_{x,0}(\mathcal{S})(x) = \mathcal{S}(x) + 1$ $\mathsf{up}_{x,1}(\mathcal{S})(x) = \mathcal{S}(x) - 1$ $\mathsf{up}_{x,b}(\mathcal{S})(y) = \mathcal{S}(y)$ for $x \neq y$	n, max $\lvert\mathcal{S}\rvert$ m, # buckets b, bucket size (entries) f, fingerprint size (bits)
Count-min sketch	Multisets $\mathcal{S} \in \mathrm{Func}(\mathcal{X}, \mathbb{N})$	$\mathsf{qry}_x(\mathcal{S}) = \mathcal{S}(x)$	$\mathsf{up}_{x,0}(\mathcal{S})(x) = \mathcal{S}(x) + 1$ $\mathsf{up}_{x,1}(\mathcal{S})(x) = \mathcal{S}(x) - 1$ $\mathsf{up}_{x,b}(\mathcal{S})(y) = \mathcal{S}(y)$ for $x \neq y$	n, max $\lvert\mathcal{S}\rvert$ k, # hash functions and arrays m, array size (counters) d, bits per counter

Table 1: The data structrues that we consider. The set $\mathcal{X}$ is some understood universe of base objects. Each data structure yields a space-efficient representation of its input data object and, in the presense of non-adaptive attacks, provides approximately correct responses to the supported queries. For counting filters, cuckoo filters, and count-min sketch, typical implementations prevent updates that would cause $\mathcal{S}(x) - 1 < 0$.

$\mathbf{Exp}^{\text{err-up}}_{\Pi,d,r}(A)$ $\mathbf{Exp}^{\text{err-up(s)}}_{\Pi,r}(A)$

$\mathcal{P} \leftarrow \emptyset$
$ct \leftarrow 0$
$K \twoheadleftarrow \mathcal{K}$
$i \twoheadleftarrow A^{\mathrm{Rep},\mathrm{Up},\mathrm{Qry}}$,Reveal
if $i \in \mathcal{P}$ then return 0
return $[\sum_{\mathrm{qry}} err_i[\mathrm{qry}] \geq r]$

oracle **Rep**($\mathcal{S}$):
repr $\twoheadleftarrow \mathrm{Rep}_K(\mathcal{S})$
if repr $= \perp$ return $\perp$
$ct \leftarrow ct + 1$
$\mathrm{repr}_{ct} \leftarrow$ repr
$C_{ct} \leftarrow \emptyset$
$\mathcal{S}_{ct} \leftarrow \mathcal{S}$
rv $\leftarrow \mathrm{repr}_{ct}$; rv $\leftarrow \top$
return rv

oracle **Up**(i, up):
repr $\twoheadleftarrow \mathrm{Up}_K(\mathrm{repr}_i, \mathrm{up})$
if repr $= \perp$ return $\perp$
$X \leftarrow \mathcal{S}_i$
$\mathcal{S}_i \leftarrow \mathrm{up}(X)$
$\mathrm{repr}_i \twoheadleftarrow$ repr
$C_i \leftarrow \emptyset$
for qry in err_i do
  $a \leftarrow \mathrm{Qry}_K(\mathrm{repr}_i, \mathrm{qry})$
  if $err_i[\mathrm{qry}] > d(a, \mathrm{qry}(\mathcal{S}_i))$ then
    $err_i[\mathrm{qry}] \leftarrow d(a, \mathrm{qry}(\mathcal{S}_i))$
rv $\leftarrow \mathrm{repr}_i$; rv $\leftarrow \top$
return rv

oracle **Qry**(i, qry):
if qry $\in C_i$ then return $\perp$
$C_i \leftarrow C \cup \{\mathrm{qry}\}$
$a \leftarrow \mathrm{Qry}_K(\mathrm{repr}_i, \mathrm{qry})$
if $err_i[\mathrm{qry}] < d(a, \mathrm{qry}(\mathcal{S}_i))$ then
  $err_i[\mathrm{qry}] \leftarrow d(a, \mathrm{qry}(\mathcal{S}_i))$
return a

oracle **Reveal**(i):
$\mathcal{P} \leftarrow \mathcal{P} \cup \{i\}$
return repr_i

Figure 2: Two notions of adversarial correctness. The ERR-UP **notion captures correctness when the representation is always known to the adversary, while the** ERR-UP(S) **notion captures correctness when the representation is secret.**

q_T calls to **Qry**, q_U calls to **Up**, and q_H calls to a random oracle. The advantage is defined analagously in the private-representation setting.

3.1 Generic results

First, we note that any structure which is insecure in the immutable case is also insecure in the mutable case. Any adversary in the immutable case is identical to a corresponding adversary in the mutable case which simply never makes use of the **Up** oracle. From this we know that standard (unsalted, unkeyed) Bloom filters cannot ensure correctness in the mutable case. (3.1: TS says: This is supported by a result that doesn't appear here. We should port over relevant results from the previous draft paper.)

Keyless structures.

In a keyless structure, we assume all details of the algorithm used are known beforehand by the adversary, except any which might be generated 'on the fly' as representations are created and modified. For example, many of these structures will assume the use of a random salt which is picked at the time the representation is created. A structure with neither randomization nor a private key is usually vulnerable to pre-computation and offline attacks, where the adversary can search for errors to produce without having to interact with the structure itself.

In many cases, it is easier to reason about an adversary which only has the chance to create and manipulate a single representation, rather than being able to call a **Rep** oracle as many times as it likes. Fortunately, in the case of keyless structures we can show that the adversary's advantage in a single-representation case is bounded above by a multiple of the advantage in the general case. We use ERR-UP1 and ERR-UP(S)1 to denote the public-representation and private-representation experiments where the adversary makes a single **Rep** query. In writing the advantage for these games we omit the q_R parameter since it is fixed at 1.

Note that in the ERR-UP(S)1 scenario we need not provide the adversary with a **Reveal** oracle. If the adversary uses only a single representation, may assume without loss of generality that they make no call to **Reveal**, since doing so would prevent the adversary from having any possibility of winning. Since the case of a single **Rep** query is considerably simpler to handle, the first step

in each of our proofs will be to reduce ERR-UP and ERR-UP(S) to ERR-UP1 and ERR-UP(S)1 respectively. In addition to this, we wish to move from actual hash functions to true random functions. Using the following lemmas, we may reduce the case of ERR-UP for any structure using a salted hash to the case of ERR-UP1 using a true random function, and we may reduce ERR-UP(S) for a structure using a secret-keyed hash to the case of ERR-UP(S)1 using a true random function.

Lemma 3.1 (ERR-UP 1 and ERR-UP(S) 1 imply ERR-UP and ERR-UP(S) for keyless structures). *Let $\Pi = (\textit{Rep}, \textit{Qry}, \textit{Up})$ be a data structure with key space $\{\varepsilon\}$. For every $t, q_R, q_T, q_U, q_H, r \geq 0$, it holds that*

$$\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(t, q_R, q_T, q_U, q_H) \leq q_R \cdot \mathbf{Adv}^{\text{err-up1}}_{\Pi,r}(O(f(t)), q_T, q_U, q_H),$$

where $f(t) = t + (q_R - 1)\mathsf{T}_{\textit{Rep}}(t) + q_T\mathsf{T}_{\textit{Qry}}(t) + q_U\mathsf{T}_{\textit{Up}}(t)$.

Proof. For a fixed $r \geq 0$, let A be an ERR-UP or ERR-UP(S) adversary which runs in t time steps and makes q_R **Rep** queries, q_T **Qry** queries, q_U **Up** queries, and q_H RO queries. We construct an adversary B for ERR-UP1 or ERR-UP(S)1, respectively, as follows.

First, B initializes a counter $ct \leftarrow 0$ and a set $C \leftarrow \emptyset$, and samples $q \twoheadleftarrow [q_R]$. Next B executes A, simulating the answers to its oracle queries as follows. When A asks the query $\mathbf{Rep}(\mathcal{S})$, B sets $ct \leftarrow ct + 1$ and stores $\mathcal{S}_{ct} \leftarrow \mathcal{S}$. Then, if $ct = q$, B forwards $\mathcal{S}$ to its own **Rep** oracle, returning the resulting value (repr in the public-representation case, or $\top$ in the private-representation case) to A. Otherwise, B computes $\text{repr}_{ct} \twoheadleftarrow \textsc{Rep}(\mathcal{S})$ and returns either repr_{ct} or $\top$. When A asks for the query $\mathbf{Qry}(i, \text{qry})$, B first checks if $(i, \text{qry}) \in C$ and returns $\perp$ if this condition holds. Otherwise B forwards (i, qry) to its **Qry** oracle and returns a if $i = q$, and reutrns $\textsc{Qry}(\text{repr}_i, \text{qry})$ otherwise. Similarly, when A makes an $\mathbf{Up}(i, \text{up})$ query, B forwards (i, up) to its **Up** oracle if $i = q$ and evaluates $\textsc{Up}(\text{repr}_i, \text{up})$ otherwise. Finally, queries from A to its RO are simply forwarded to B's RO. When A halts and outputs j, B does the same.

If $j = q$, then B wins if A does, since all queries from A to repr_j were forwarded to B's **Qry** oracle. Given that q is sampled uniformly from the range $[q_R]$, it follows that

$$\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(t, q_R, q_T, q_U, q_H) \leq q_R \cdot \mathbf{Adv}^{\text{err-up1}}_{\Pi,r}(O(f(t)), q_T, q_H).$$

Note that B makes at most q_T queries to **Qry** and q_H queries to its RO. Since A runs in t time steps and writing a bit takes 1 time step, the input length to any Rep, Qry, or Up evaluated by B is at most t bits. Hence, adversary B runs in time $O(t + (q_R - 1)\mathsf{T}_{\textsc{Rep}}(t) + q_T\mathsf{T}_{\textsc{Qry}}(t) + q_U\mathsf{T}_{\textsc{Up}}(t))$. □ □

Lemma 3.2 (In the ROM, salted hashing is almost as good as distinct random functions in ERR-UP 1). *Let $\Pi = (\textit{Rep}, \textit{Qry}, \textit{Up})$ be a data structure with key space $\{\varepsilon\}$ and salt space $\{0,1\}^\lambda$, and let Π' be the same structure using true random functions in place of salted hash functions. For every $t, q_R, q_T, q_U, q_H, r \geq 0$, it holds that*

$$\mathbf{Adv}^{\text{err-up1}}_{\Pi,r}(t, q_R, q_T, q_U, q_H) \leq \frac{q_H}{2^\lambda} + \mathbf{Adv}^{\text{err-up1}}_{\Pi',r}(t, q_R, q_T, q_U, q_H)$$

Proof. Let $\mathbf{G}_0$ be the standard ERR-UP1 game for the structure Π, and let $\mathbf{G}_1$ be the same game for Π'. There exists an adversary B such that $\Pr[\,\mathbf{G}_0(A) = 1\,] \leq \Pr[\,\mathbf{G}_1(B) = 1\,] + q_H/2^\lambda$. This adversary initializes an empty table R and simulates A. When a query w is sent to **Hash**, B returns $R[w]$ if this entry in the table is defined. Otherwise, if $w = \langle Z, x\rangle$ for some $Z \in \{0,1\}^\lambda$ and $x \in \{0,1\}^*$, forward (Z, x) to **Hash**, store the result in $R[w]$, and return this value. Finally, if $R[w]$ is not defined and w is not of this form, sample r uniformly from the range of the hash function, store the result in $R[w]$ and return that result. Queries to all other oracles are simply forwarded to B's oracle. Assuming the output of the hash function is uniformly distributed, this simulation is perfect unless A guesses the salt correctly, which happens with probability $q_H/2^\lambda$. □ □

Keyed structures.

As in the keyless case, we can reduce the case of ERR-UP(S) for a secretly-keyed hash to the case of ERR-UP(S)1 using a true random function. Though the proof is different, we achieve a similar bound as in the unkeyed private-representation case. Despite the similar-looking bounds, the secret-keyed case is preferable in practice because the q_R **Rep** queries are 'online' while the q_H **Hash** queries are 'offline', limited only by the computational capabilities of the adversary.

Lemma 3.3 (ERR-UP 1 with random functions implies ERR-UP with keyed hashing). *Let $\Pi = (\textit{Rep}, \textit{Qry}, \textit{Up})$ be a data structure with key space $\mathcal{K}$ and salt space $\{0,1\}^\lambda$, and let Π' be the same structure using true random functions in place of salted and keyed hash functions. For every $t, q_R, q_T, q_U, q_H, r \geq 0$, it holds that*

$$\mathbf{Adv}^{\text{err-up1}}_{\Pi,r}(t, q_R, q_T, q_U, q_H) \leq \frac{q_R^2}{2^\lambda} + q_R \cdot \mathbf{Adv}^{\text{err-up1}}_{\Pi',r}(t, q_R, q_T, q_U, q_H)$$

Proof. In each of the structures considered here, we model the use of a secretly keyed hash function with the use of a pseudorandom function F_K. Let $\mathbf{G}_0$ be the ERR-UP game on this structure. Our first step is to move to a game $\mathbf{G}_1$ where the pseudorandom function F_K is replaced by a true random function Rand which is lazily evaluated as necessary when **Rep**, **Qry**, and **Up** are called. By a conditioning argument, the advantage of the adversary is given by $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) = \mathbf{Adv}^{\text{prf}}_{F}(B) + \Pr[\,\mathbf{G}_1(A) = 1\,]$.

Consider another game $\mathbf{G}_2$ that ensures the salts do not repeat, by choosing Z exclusively from salts which have not been previously used. The game is identical to $\mathbf{G}_1$ until a salt repeats, which by the birthday bound occurs with probability $q_R^2/2^\lambda$. Therefore $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) = \mathbf{Adv}^{\text{prf}}_{F}(B) + q_R^2/2^\lambda + \Pr[\,\mathbf{G}_2(A) = 1\,]$.

Next, we revise the game to $\mathbf{G}_3$ where the adversary gets credit for queries $\mathbf{Qry}(i, \text{qry})$ which are false positives for any repr_j, regardless of the actual argument i given to **Qry**. Since this can only benefit the adversary, $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + q_R^2/2^\lambda + \Pr[\,\mathbf{G}_3(A) = 1\,]$.

However, because each representation makes use of a random function for determining the value of all representations, updates, and queries, the probability of a previously unqueried element

producing an error in one representation is independent of its probability of producing an error in the other representations. When we move to the final game $\mathbf{G}_4$ where the adversary is only allowed to call **Rep** once, the adversary's advantage will then decrease by a factor of at most q_R. This gives us the final result:

$$\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + \frac{q_R^2}{2^\lambda} + q_R \cdot \Pr[\,\mathbf{G}_4(A) = 1\,]$$

□ □

Invertible structures.

Several structures are designed to implement both insertion and deletion operates. This allows the adversary a great deal of variety in the updates it can perform during the security experiment. For a more general notion, we define an *invertible* structure as one where, for any initial representation repr and any other representation $\mathsf{repr}' = \textsc{Up}_K(\ldots \textsc{Up}_K(\mathsf{repr}, \mathsf{up}_1)\ldots, \mathsf{up}_n)$ generated by a sequence of update operations applied to the initial representation, there exists another sequence of update operations such that $\textsc{Up}_K(\ldots \textsc{Up}_K(\mathsf{repr}', \mathsf{up}'_1)\ldots, \mathsf{up}'_m) = \mathsf{repr}$. In other words, any sequence of updates applied to any representation can be undone by further updates.

Lemma 3.4 (Salts do not affect ERR-UP for invertible structures). *Let* $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$ *be a data structure with a salt randomly initialized at runtime and* Π' *be the same structure using a fixed value from the salt space in place of a randomized salt. For every* $t, q_R, q_T, q_U, q_H, r \geq 0$*, it holds that*

$$\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(t, q_R, q_T, q_U, q_H) \leq \mathbf{Adv}^{\text{err-up}}_{\Pi',r}(O(t), 1, q_T, q_U + 2n(q_R + q_T + q_U), q_H),$$

where n is the longest minimal sequence of update operations needed to generate a data structure in the space.

Proof. Consider an adversary A in the case of a non-salted data structure which makes q_R queries to **Rep** and q_T queries to **Qry**. We construct an adversary B for the salted case which produces the same errors as follows. First, B initializes a counter ct to 0 and calls the **Rep** oracle on the empty set, receiving an empty representation repr together with the salt used to create the representation. Then B runs A, answering its oracle queries as follows. Whenever A makes a query of the form $\mathbf{Rep}(\mathcal{S})$, B sets $ct \leftarrow ct + 1$ and calls **Up** repeatedly on repr to transform it into a representation of $\mathcal{S}$. Then B returns the modified repr to A, stores $\mathcal{S}_{ct} \leftarrow \mathcal{S}$, and performs the opposite updates in reverse order to return to the original empty representation repr. If A makes a query of the form $\mathbf{Qry}(i, \mathsf{qry})$, B calls **Up** repeatedly to transform repr into a representation of $\mathcal{S}_i$ and then returns the result of querying its own oracle with $\mathbf{Qry}(1, \mathsf{qry})$. Then once again B performs the inverse updates to transform repr back into the original empty representation. If A queries for $\mathbf{Up}(i, \mathsf{up})$, B sets $\mathcal{S}_i \leftarrow \mathsf{up}(\mathcal{S}_i)$ and again uses **Up** queries to transform repr into a representation of $\mathcal{S}_i$, returns the value of repr, and then performs opposite **Up** queries to return repr to the empty representation. Finally, B forwards any of A's RO queries to its own RO.

In general this may use as many as $2n(q_R + q_T + q_U)$ update queries, where n is the longest minimal sequence of update operations needed to generate a data structure in the space. Furthermore B succeeds if A does, so the adversaries' advantages are equal. □ □

If we raise the threshold for what counts as an error, it can only become more difficult for the adversary to succeed, so in fact all error functions of the form $d(x, y) = [|x - y| > c]$ for some constant c also have this upper bound.

Another obvious alternative is to simply use the Euclidean distance metric, so that $d(x, y) = |x - y|$. Using this notion of error in fact turns out to be at least as good for the adversary as the binary error metric. Any attack in the correctness experiment with error function $d'(x, y) = [|x - y| \geq 1]$ and $r' = r$ performs at least as well against a count-min sketch with error function $d(x, y) = |x - y|$. Similarly, any attack in the game with error function $d'(x, y) = [x - y > 2]$ and $r' = r/2$ performs at least as well in the game with error function $d(x, y) = |x - y|$, and so on for any other binary error function which gives some fixed credit for finding an error which is off from the correct value by at least some minimum constant value. This means that, if d is the Euclidean distance error function and d'_k is the binary function $d'_c(x, y) = [|x - y| \geq c]$, we have $\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s,r,d}(t, q_R, q_T, q_U, q_H) \leq \mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s,r/c,d'}(t, q_R, q_T, q_U, q_H)$.

There are two immediate distinctions that can be made between different security games. First, we can ask whether it is possible for the adversary to predict the representations ahead of time, in which case the **Rep** and **Up** oracles are largely unnecessary. Second, we can ask whether it is possible for the adversary, given its knowledge of the representations, to predict the outputs of **Qry** ahead of time.

We use simulation-based definitions for these two notions. In particular, we say a data structure has independent representations up to some leakage $\mathsf{lk}\colon \mathcal{D} \to \{0,1\}^*$ if there is a simulator consisting of two functions $\mathsf{Sim}_d\colon \{0,1\}^* \to \mathcal{D}$ and $\mathsf{Sim}_u\colon \mathcal{D} \times \{0,1\}^* \to \mathcal{U}$, such that $\mathsf{Sim}_d(\mathsf{lk}(\mathcal{S}))$ produces a set whose representation is indistinguishable from $\mathcal{S}$ and $\mathsf{Sim}_u(\mathcal{S}', \mathsf{lk}(\mathcal{S}))$ produces an update up' such that $\mathsf{up}'(\mathcal{S}')$ is indistinguishable from $\mathcal{S}$.

In the case that the adversary can predict representations ahead of time and can predict query responses based on that representation, the structure is inherently insecure because the adversary need not rely on any of the oracles in the experiment. Instead, it can simulate **Rep**, **Up**, and **Qry** itself and search for a representation which produces a large number of errors. Once this is found, the adversary can call **Rep** to construct this representation and call **Qry** for each of the errors it has found until it surpasses the error threshold r and wins. Assuming the minimum size of an error is 1, this requires only the resources $q_R = 1$ and $q_T = r$, which is the minimum necessary for any attacker to succeed. Furthermore, it always succeeds unless r is so large as to be unachievable for the given structure with the given parameters.

If the adversary cannot guess representations but can predict query responses given representations, there are still limits on how

$\underline{\mathbf{Exp}^{\text{indqry}}_{\Pi}(A)}$	oracle $\mathbf{Rep}(\mathcal{S})$:
$K \twoheadleftarrow \mathcal{K}$	$i \leftarrow i + 1$
$r \twoheadleftarrow \mathcal{R}$	$\text{repr}_i \twoheadleftarrow \text{Rep}_K(\mathcal{S})$
$\mathcal{S} \twoheadleftarrow \mathcal{D}$	return repr_i
$\text{repr} \twoheadleftarrow \text{Rep}_K(\mathcal{S})$	$\underline{\text{oracle } \mathbf{Up}(i, \text{up}):}$
$\text{qry} \twoheadleftarrow \text{Sim}(\text{lk}(\text{repr}), r)$	$\text{repr}_i \twoheadleftarrow \text{Up}_K(\text{repr}_i, \text{up})$
if $\text{Qry}_K(\text{repr}, \text{qry}) \neq r$ return 1	return repr_i
$i \leftarrow 0$	
$r' \twoheadleftarrow A^{\mathbf{Rep}(\cdot),\mathbf{Up}(\cdot,\cdot)}(\text{repr}, \text{qry})$	
return $[r = r']$	

secure the structure can be. An example of this occurs with keyed but unsalted Bloom filters. Even if the adversary cannot guess the secret key, knowing the representation of a singleton $\{x\}$ allows them to determine with certainty whether any other filter has x as an element by checking whether all the bits set to 1 in the representation of $\{x\}$ are also set in the other representations.

Finally, if both representations and queries are unpredictable by the adversary, we can intuitively say that the adversary should not be able to do any better than random guessing, which would mean that the adversary cannot do any better than the non-adaptive case.

We can show this formally by moving to an alternate game where **Rep** ignores the specific elements given as input and instead creates a representation of a randomly-generated structure with the same leakage, where **Up** performs a randomly-generated update that has the same leakage as the real updated set would have, and where **Qry** returns a random result that is consistent with the adversary's knowledge of the underlying set. By independence of representations, this produces a representation which is indistinguishable from the true representation. Furthermore, by independence of queries, **Qry** behaves indistinguishably (from the adversary's perspective) from Qry on the true representation. Any adversary therefore has an equal probability of succeeding in this game as in the original game. However, since the queries are being answered randomly, the adversary's choice of inputs to the query oracle are irrelevant: they effectively are guessing randomly.

The best the adversary can hope to do, then, is to create a structure with maximum error rate and then make as many arbitrary **Qry** calls as possible. Extra **Rep** and **Up** calls are worthless because, by query independence, seeing additional representations does not provide the adversary with any additional information about what the responses to queries will be. The probability of getting an error with a random call to **Qry** is in each case given by the nonadaptive error bound for that data structure. If this error probability is p, and the adversary is attempting to accumulate r errors over q_T trials, the binomial distribution provides a bound on the adversary's advantage in the game. Expressed in terms of the regularized incomplete beta function, we have $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) \leq I_p(r, q_T - r + 1)$.

In the data structures we consider, it is never the case that representations and queries are completely unpredictable. However, if we can show that the probability of making accurate predictions is bounded by some small probability, we can add that probability into the error bound and move to a game with ideal, unpredictable operations.

In order to show that a salted, secretly-keyed Bloom filter is secure, we want to show that the structure has query independence up to leakage that is defined to be the number of bits in the filter which are set to 1. If the PRF F is good, and the key is unknown to the adversary, the adversary cannot use the random oracle to predict what representations will look like ahead of time. In particular, by a straightforward conditioning argument, $\mathbf{Adv}^{\mathbf{g}_0}_{\Pi,r}(A) \leq \mathbf{Adv}^{\text{prf}}(F) + \mathbf{Adv}^{\mathbf{g}_1}_{\Pi,r}(A)$, where $\mathbf{G}_1$ is the same game but with the pseudorandom function replaced by a lazily-evaluated random function. In this game, the adversary cannot possibly guess queries ahead of time with better than 50/50 odds, since the output of the query depends on the output of a true random function on an untested input. We can then immediately apply the independence lemma to see that $\mathbf{Adv}^{\mathbf{g}_1}_{\Pi,r}(A) \leq I_p(r, q_T - r + 1)$, where p is the false positive probability. In a filter where $\text{lk}(\text{repr})$ out of m bits are set to 1, this probability is simply $(\text{lk}(\text{repr})/m)^k$. The adversary can maximize this value by setting as many bits to 1 as possible. With a threshold of n bits allowed to be set to 1, the overall advantage is then

$$\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(A) \leq \mathbf{Adv}^{\text{prf}}(F) + I_{(n/m)^k}(r, q_T - r + 1)$$

4 BLOOM FILTER RESULTS

(4.1: DC (lead) to-do: Explain away all cases except the one(s) for which we give security (upperbound) theorems.) The standard Bloom filter shows a variety of different behaviors depending on its exact implementation. If the hash functions used are chosen beforehand and potentially known to the adversary, this public information allows offline attacks to be mounted against the data structure which can produce potentially damaging false positives. In the case of immutable Bloom filters, making use of a per-representation salt is sufficient to prevent these attacks, though depending on the use case the use of non-fixed per-representation randomness may or may not be feasible. Furthermore, in the case of mutable Bloom filters there are additional difficulties with offline attacks due to adversarially-chosen updates. To guarantee correctness in this case we must additionally guarantee that representations can be kept private from the adversary.

A generic attack against unsalted, unkeyed data structures handling set membership queries is for the adversary to choose a large set $\mathcal{S}$ of potential filter elements and simulate **Rep** to produce representations for $\{x\}$ for each $x \in \mathcal{S}$. Given these, it can perform offline computations to determine disjoint $\mathcal{T}, \mathcal{R} \subseteq \mathcal{S}$ such that the elements of $\mathcal{R}$ all produce errors when queried for membership in

Rep($\mathcal{T}$). After a sufficiently large $\mathcal{R}$ has been found, the adversary makes a single **Rep** call on $\mathcal{T}$ and makes one **Qry** call per element of $\mathcal{R}$, resulting in guaranteed success for the adversary with only a minimal number of queries performed.

In general, such an attack may not be computationally feasible in the real world. However, because the attack is entirely offline, many structures with non-negligible error probabilities are vulnerable to these attacks. For example, consider a Bloom filter with false positive probability 10^{-5} with an adversary wishing to construct $r = 10$ false positives. The adversary can fix $\mathcal{T}$ in advance and perform somewhere on the order of a million hash queries to random elements in order to construct a $\mathcal{R}$ of size 10, and then perform only a single **Rep** calls and ten **Qry** calls to the service hosting the Bloom filter. This is likely to be much more feasible for the adversary than performing on the order of a million **Qry** calls in an online attack which randomly guesses elements until it accumulates 10 false positives.

The use of a salt without a private key in the public representation setting is insufficient to defeat this attack. In this setting, the adversary need only make its **Rep** query for $\mathcal{T}$ in advance, at which point it will receive both the representation repr and the salt Z used to construct it. Using this known salt, the adversary is still able to simulate **Rep** for arbitrary singleton representations. The previous attack therefore still works with the same (minimal) number of **Qry** calls at the very end of the experiment, after it has determined $\mathcal{R}$ using offline computations.

The opposite of this, using a private key without a salt, does weaken the attack somewhat. Even with public representations, the adversary cannot locally simulate **Rep** without guessing the private key. However, they can still outperform random **Qry** calls by making **Rep** queries for singleton elements without fixing any $\mathcal{T}$ in advance. After selecting a random set $\mathcal{S}$ of size $q_R - 1$, the adversary performs offline computations to find $\mathcal{T}, \mathcal{R} \subseteq \mathcal{S}$ such that the elements of $\mathcal{R}$ are false positives for the representation of $\mathcal{T}$ (which can be computed from the representations of the singleton subsets of $\mathcal{T}$). The adversary wins if there is a partition where $\mathcal{R}$ produces at least r errors on **Rep**($\mathcal{T}$).

Using a salted Bloom filter in the private representation setting, however, does provide some security. At the time a representation is created, the structure chooses a salt Z which it will use for all further queries and updates. In order for maximum security to be guaranteed, we must ensure that the representation, and in particular the salt, is kept secret from the adversary. We define this structure SBF$[H, k, m, n, \lambda]$ as the Bloom filter structure that uses $H(s) = (h_1(s), \ldots, h_k(s))$ for hashing inputs to k values in $[m]$. Furthermore, each call of Rep first involves picking a salt Z from the salt space $\{0,1\}^\lambda$, and all hashes made to insert or query for an element x are determined using $H(x\|Z)$. Finally, the parameter n means that any attempts to represent sets with more than n elements fail. (4.2: DC lead to-do: Specify what exactly is being analyzed. What are the updates?)

Theorem 4.1 (Correctness Bound for Private-Representation Salted Bloom Filters). *Fix integers $k, m, n, \lambda, r \geq 0$, let $H\colon \{0,1\}^* \to [m]$ be a function, and let $\Pi_s = SBF[H, k, m, n, \lambda]$. For every $t, q_R, q_T, q_U, q_H \geq 0$, it holds that*

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_{\text{sbf}}, r}(t, \quad q_R, \quad q_T, q_U, q_H) \leq$$
$$q_R \cdot \left[\frac{q_H}{2^\lambda} + \binom{q_T + q_U}{r} p(k, m, n+s)^r \right],$$

where s is defined to be $\min(r, q_U)$, H is modeled as a random oracle, and $p(k, m, n+s)$ is the standard, non-adaptive false-positive probability on a Bloom filter with the given parameters.

This proof first reduces to the single-representation case, which as shown in lemma 3.1 will reduce the adversary's advantage by at most a factor of q_R. The main idea behind the proof is to remove the adversary's adaptivity a step at a time. We isolate the possibility of the adversary guessing the salt, which would allow it to mount its own offline attack on the filter without relying on the **Qry** oracle. If the adversary does not guess the salt, the outputs of the **Rep**, **Qry**, and **Up** oracles are unpredictable to the adversary, producing uniformly randomly distributed bits to set (for **Rep** and **Up**) or to check (for **Qry**). Under the assumption that the adversary does not predict the salt, queries made to distinct elements are independent of each other. The only remaining issue is that the adversary can potentially gain an advantage by testing whether some object x is a false positive for the filter, and then updating the filter to include x only if the test query returned 'false'. An analysis shows that this is now (once imperfect pseudorandom functions and salt collisions have been dealt with) the only way for the adversary to gain an advantage over making queries to an immutable Bloom filter. Because this adaptive strategy introduces tricky conditional possibilities, we cannot compute an exact value for the adversary's advantage. Instead, we move to an alternate scenario where each **Qry** also produces a free update and every **Up** first performs a free query. This makes **Qry** and **Up** calls indistinguishable, so that the adversary is effectively making a series of independent random queries that each have a chance to increment the error counter. Because the number of 1s in the filter can only increase, the probability of a false positive from any one of these queries is bounded above by the probability of a false positive on the final maximally-sized filter, a probability which is given by the Kirsch and Mitzenmacher bound.

Proof. We first reduce from the ERR-UP(S) case to the ERR-UP(S)1 case, which by lemma 3.1 may scale the adversary's advantage only by a factor of q_R. The game $\mathbf{G}_0$ is exactly equivalent to the ERR-UP(S)1 experiment, so $\mathbf{Adv}^{\text{err-up1}}_{\Pi_s, r}(A) = \Pr[\, \mathbf{G}_0(A) = 1 \,]$. In $\mathbf{G}_1$ we split the hash oracle into three, giving the adversary access $\mathbf{Hash}_1$ in both stages of the game, while $\mathbf{Hash}_2$ is reserved for oracular use by Rep1, **Qry**, and **Up**. For any A for $\mathbf{G}_0$, there is B for $\mathbf{G}_1$ which produces the same advantage by simulating A. This adversary first creates its own table R with all values initially undefined. When A makes a query w to H, B returns $R[w]$ if that entry in the table is defined. Otherwise, if there are $Z \in \{0,1\}^\lambda$, $j \in [k]$, and $x \in \{0,1\}^*$ such that $w = \langle Z, j, x \rangle$, forward (Z, x) to $\mathbf{Hash}_1$. For each $j \in [k]$, set $R[\langle Z, j, x \rangle] = \vec{v}_j$, where $\vec{v}$ is the output of the $\mathbf{Hash}_1$ oracle. If there is no such triple $\langle Z, j, x \rangle$, just sample r from $[m]$ uniformly and set $R[w] = r$. In either case, return $R[w]$ to A. When A outputs its collection $\mathcal{S}$, B outputs $\mathcal{S}$ as well. Any queries by A to

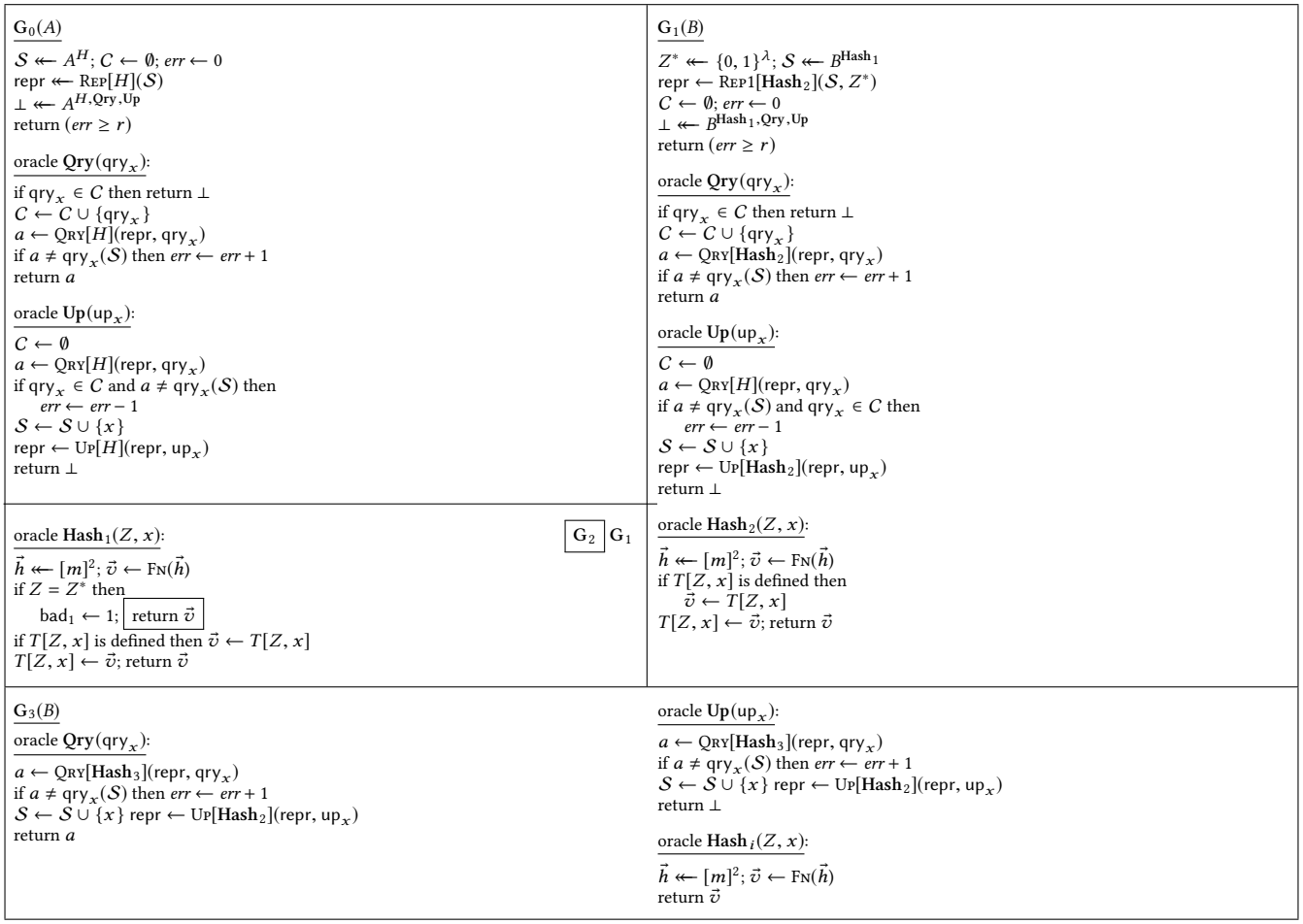


Figure 3: Games 0–3 for proof of Theorem 4.1.

Qry or **Up** are forwarded to B's corresponding oracle. The simulation is perfect because $\textsc{Rep}[H](\mathcal{S})$ and $\textsc{Up}[H](\mathcal{S}, \mathsf{up})$ are identically distributed to $\textsc{Rep}[\mathbf{Hash}_2](\mathcal{S})$ and $\textsc{Up}[\mathbf{Hash}_2](\mathcal{S}, \mathsf{up})$. Because we have a perfect simulation, $\mathbf{Adv}^{\text{err-up(s)1}}_{\Pi_s, r}(A) = \Pr[\,\mathbf{G}_1(A) = 1\,]$.

The game $\mathbf{G}_2$ is the same as $\mathbf{G}_1$ until bad_1 is set, which occurs exactly when B sends (Z^*, x) to $\mathbf{Hash}_1$ for some x. In the first phase, there is again a $q_1/2^\lambda$ chance of the adversary guessing the salt. In the second phase, the random sampling used by $\mathbf{Hash}_i$ ensures that each call the adversary makes to the $\mathbf{Hash}_i$ oracle is independent of all previous calls. We therefore have a $q_2/2^\lambda$ chance of the adversary guessing the salt during this phase, for a total chance of $q_H/2^\lambda$ chance of the adversary guessing the salt at some point during the experiment. Then $\mathbf{Adv}^{\text{err-up(s)}_1}_{\Pi_s, r}(A) \le \Pr[\,\mathbf{G}_2(B) = 1\,] + q_H/2^\lambda$. Having taken this into account, we may now assume the adversary never guesses the salt.

We want to show that alternating between sequences of queries and sequences of updates is no better than making one long series of updates and then one long sequence of queries. There are three types of updates the adversary can make: updates to add elements that have been queried and found to be false positives; updates to add elements that have been queried and found not to be false positives; and updates to add elements that have not been queried yet. We may assume without loss of generality that the adversary never makes the first type of update, since doing so is never beneficial (it does not change the representation at all and decreases the number of errors the adversary has found).

Note that the choices of $\vec{v}$ constructed by the $\mathbf{Hash}_i$ oracles are independent of all previous queries. Because of this, any update of type 3 is equivalent to any other update of type 3; the probability of any bit being flipped by one update is the same as the probability of the bit being flipped by the other update. Similarly, any update of type 2 is equivalent to any other update of type 2, but is not the same as type 3 since the probability is conditioned on $\vec{v}$ not being a false positive. We assume the worst case, namely that all updates are type 2 (i.e. at least one bit is flipped by each update).

Because the adversary never guesses the salt, $\mathbf{Hash}_1$ simply functions as a random oracle. Furthermore, we can assume the adversary never adds an element of $\mathcal{S}$ to $\mathcal{S}$ and never makes a **Qry** call for an element which is already in $\mathcal{S}$, since neither of these

provides any additional information and neither affects the rest of the experiment in any way.

Now we move to the game $\mathbf{G}_3$. Here each **Qry** query also calls **Up** to add that element to $\mathcal{S}$. Additionally, the penalty for adding known false positives is removed. To avoid penalizing the adversary by prematurely maxing out the number of elements in $\mathcal{S}$ because of added false positives, we also increase the maximum size of $\mathcal{S}$ from n to $n+s$, where $s = \min(r, q_U)$. Because the adversary (without loss of generality) stops after accumulating r errors, only $\min(r, q_U)$ false positives will be added to $\mathcal{S}$ and so a maximum size of $n+s$ is sufficient to produce no penalty for the adversary. Furthermore, each **Up** call is preceded by a **Qry** call. Neither of these changes can produce a worse result for the adversary, so $\Pr[\,\mathbf{G}_2(B) = 1\,] \le \Pr[\,\mathbf{G}_3(B) = 1\,]$. Now, however, there is no longer any distinction between **Qry** and **Up** calls. All calls to either oracle are independent of each other and produce the same effect, querying and then updating $\mathcal{S}$. Each of these queries for false positives is at most as successful as a query to a Bloom filter with $n+r$ elements, so the adversary's probability of finding a false positive on any query is bounded above by the standard success rate for a Bloom filter with those parameters. The adversary is required to produce r errors over the course of $q_T + q_U$ queries, which by the binomial theorem gives an advantage bound of $\Pr[\,\mathbf{G}_3(B) = 1\,] \le \binom{q_T+q_U}{r} p(k,m,n+s)^r$.

The full adversarial advantage is then

$$\mathbf{Adv}^{\text{err-up(s)}_1}_{\Pi_s,r}(A) \le q_R \cdot \left(\frac{q_H}{2^\lambda} + \binom{q_T+q_U}{r} p(k,m,n+s)^r\right).$$

□ □

Once we have reduced to the case of $\mathbf{Hash}_1$ functioning as a random oracle and **Qry** calls only being made to distinct elements not in $\mathcal{S}$, the adversary has only three ways of interacting with the filter: querying an element not in $\mathcal{S}$ which has not yet been queried, inserting a new element into $\mathcal{S}$ which has not yet been queried, and inserting a new element into $\mathcal{S}$ which has been queried and was found to not be a false positive. In the non-threshold case, the last of these is exploitable, since an element which is confirmed not to be a false positive will necessarily flip at least one bit when inserted into the representation. However, when a maximum proportion p of the bits are allowed to be set to 1, adding known non-false-positives rather than unqueried elements only causes the threshold to be reached faster, without providing any additional benefit to the adversary. In any case the adversary can do no better than setting exactly pm bits to 1, in which case the false positive rate will be exactly p^k and the adversarial advantage will be

$$\mathbf{Adv}^{\text{err-up(s)}_1}_{\Pi_s,r}(A) \le q_R \cdot \left(\frac{q_H}{2^\lambda} + \binom{q_T}{r} p^{kr}\right).$$

The case of a Bloom filter whose representations are always public but which uses a private key in addition to a salt is almost identical. In practice the bound is actually somewhat stronger: the instance of q_H in the bound is replaced with q_R, which means that the adversary which attempts to guess the salt must rely entirely on 'online' queries as opposed to 'offline' queries to q_H.

These two theorems together show that in order to guarantee maximal correctness in the streaming setting, one must ensure that representations are kept private from potential adversaries, and possibly use a secret key in addition to a salt when implementing a Bloom filter. In either case, the hidden information is necessary to protect the filter from an adversary.

Theorem 4.2 (Correctness Bound for Keyed Bloom Filters). *Fix integers $k, m, n, \lambda, r \ge 0$, where $m \ge \lambda$. For every $t, q_R, q_T, q_U, q_H \ge 0$ such that $q_T \ge r$, it holds that*

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_{\text{sbf}},r}(t,\quad q_R,\quad q_T, q_U, q_H) \le q_R \cdot \left[\frac{q_R}{2^\lambda} + \binom{q_T+q_U}{r} p(k,m,n+r)^r\right].$$

Despite the similarity of the bounds, the proof is somewhat different than in the private-representation salted-but-unkeyed case. The first thing we want to do is move from a pseudorandom function to a truly random function, about which we can produce bounds much more easily. This is problematic only if the adversary can easily distinguish between the pseudorandom function and a true random function, which for a good choice of pseudorandom function will not occur. Next, we note that for a true random function, the queries and updates in different representations will be independent unless the randomly-chosen salt Z is the same across two or more representations. The birthday bound gives us the chances of this occurring, but for sufficiently large salt size λ the chance of this will also be small. Once we have reduced to a case where the filter operates on true random functions and salts never repeat, we are again left with a series of almost-independent queries where the adversary may be able to gain an advantage by following up a **Qry** call with an **Up** call. We take the same approach as in the previous case, turning **Qry** and **Up** into identical independent queries and then applying the Kirsch and Mitzenmacher upper bound for the largest set represented during the game.

Proof. We define algorithms $\textsc{Rep1}[f] : \{0,1\}^* \times \{0,1\}^\lambda \to \{0,1\}^*$, $\textsc{Qry1}[f] : \{0,1\}^* \times \mathcal{Q} \to \mathcal{R}$, and $\textsc{Up1}[f] : \{0,1\}^* \times \mathcal{U} \to \{0,1\}^*$ for arbitrary functions $f : \{0,1\}^* \to [m]$. The algorithm $\textsc{Rep1}[f]$ takes a set and salt, returning a representation of the Bloom filter constructed as usual using the given salt, but with calls to F_K replaced with calls to f. Note that the salt is included in the output representation. Similarly, $\textsc{Qry1}[f]$ and $\textsc{Up1}[f]$ perform queries and updates on representations using f. in place of F_K.

First we move to the game $\mathbf{G}_0$, which does not change the semantics of the usual correctness experiment. We assume without loss of generality that the adversary does not make the same query twice between updates, so there is no need to keep track of $\mathcal{S}$. We instead have sets $\mathcal{I}, \mathcal{Z}, \mathcal{Q}$ not defined in the normal game. As yet these sets have no effect on the game. Instead of $\textsc{Rep}_K(\cdot)$, the game uses $\textsc{Rep1}[F_K](\cdot, Z)$ for freshly-sampled Z, but these algorithms have identical behavior.

Next consider the game $\mathbf{G}_1$. Here we replace F_K entirely with a true random function Rand which is lazily evaluated as necessary when **Rep**, **Qry**, and **Up** are called. The advantage of the adversary is then $\mathbf{Adv}^{\text{err-up}}_{\Pi_{\text{kbf}},r}(A) = \mathbf{Adv}^{\text{prf}}_F(B) + \Pr[\,\mathbf{G}_1(A) = 1\,]$.

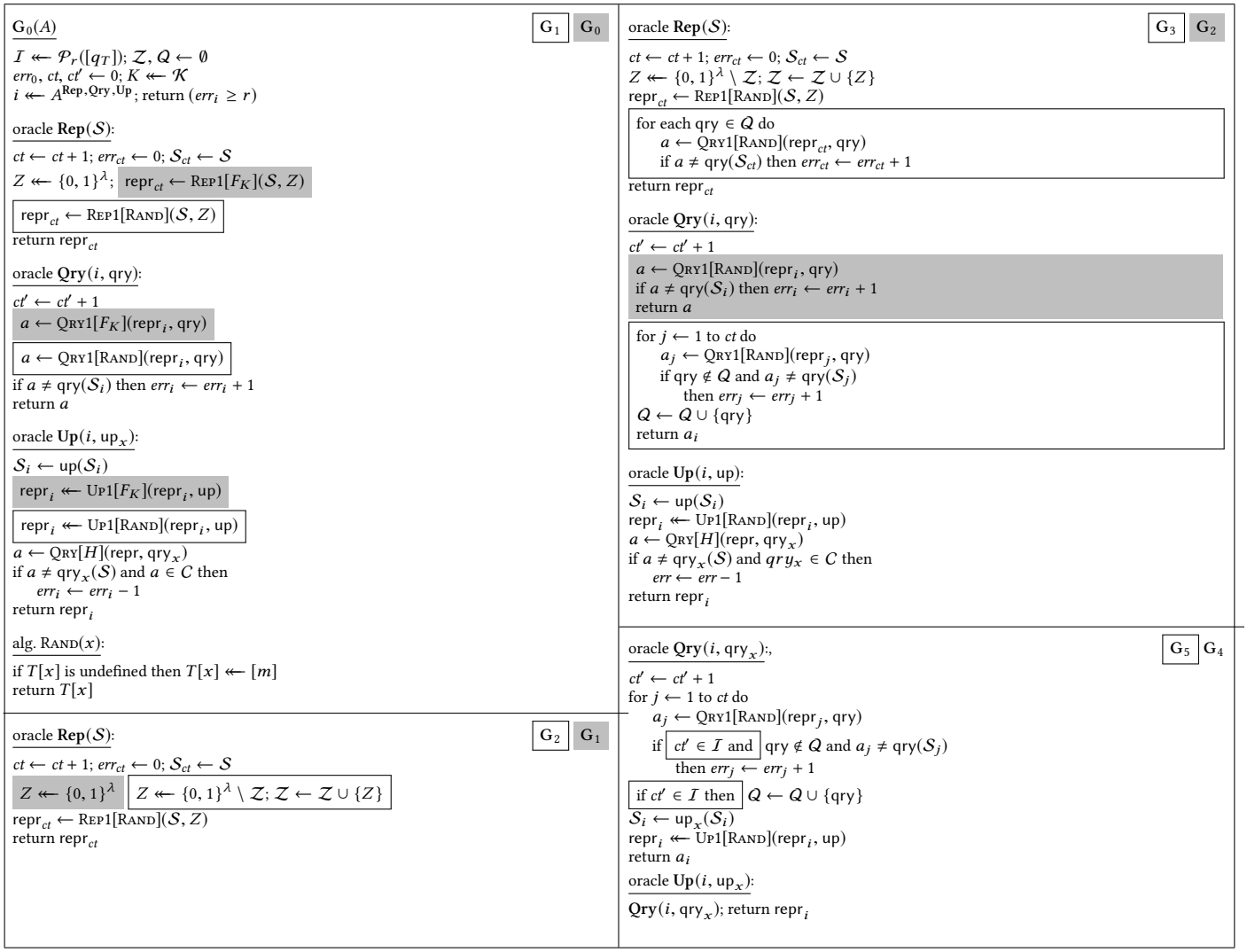


Figure 4: Games 0–5 for the proof of Theorem 4.2.

We want to ensure that salts do not repeat, so $\mathbf{G}_2$ provides a game where the Z is randomly chosen exclusively from salts which have not been previously used. The game is idenitcal to $\mathbf{G}_1$ until a salt repeats, which using the birthday bound will occur with probability $q_R^2 2^{-\lambda}$. Therefore $\mathbf{Adv}^{\text{err-up}}_{\Pi_{\text{kbf}},r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + q_R^2 2^{-\lambda} + \Pr[\,\mathbf{G}_2(A) = 1\,]$.

We then revise the game to $\mathbf{G}_3$, where the adversary gets credit for queries $\mathbf{Qry}(i, \text{qry})$ which are false positives for any repr_j, regardless of the actual argument i given to $\mathbf{Qry}$. The oracle simply iterates through all representations the adversary has created and increments err_j for any j where $\text{qry}(\mathcal{S}_j)$ disagrees with $\text{Qry1}[\text{Rand}](\text{repr}_j, \text{qry})$. The adversary's view does not change as a result of this modification, and its advantage may only increase as additional opportunities to increment each err_i occur, so that $\mathbf{Adv}^{\text{err-up}}_{\Pi_{\text{kbf}},r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + q_R^2 2^{-\lambda} + \Pr[\,\mathbf{G}_3(A) = 1\,]$.

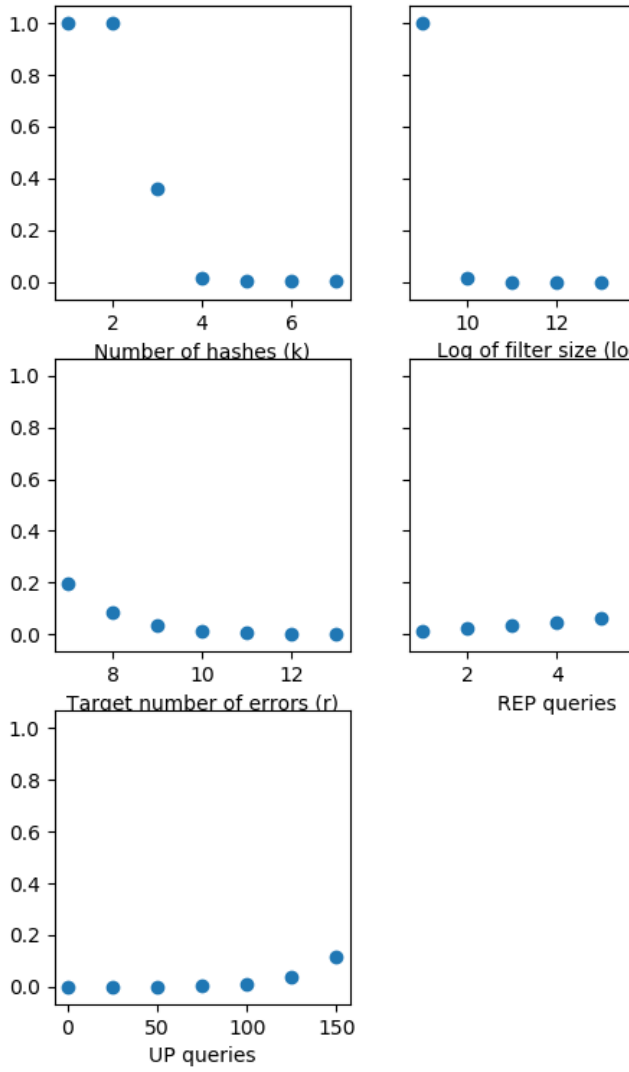
Next, in $\mathbf{G}_4$ we modify $\mathbf{Qry}$ and $\mathbf{Up}$ to perform the same function. Each one tests the given element for false positive status across all extant representations, and then updates only the queried representation with that element. As in the case of the salted Bloom filter, we must remove the penalty for adding a false positive and increase the maximum size of the filter from n to $n+r$ so that this change can never reduce the adversary's chances of success. Because all queries are independent of each other, adding a known non-false-positive is again the optimal choice for the adversary under any circumstance. Furthermore, note that while the adversary may not be aware of its score in each representation, we can assume without loss of generality that it halts if it has sent several $\mathbf{Qry}$ calls to the same representation and r of those have returned a false positive. This means that no more than r false positives will be added to any single representation. Then $\mathbf{Adv}^{\text{err-up}}_{\Pi_{\text{kbf}},r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + q_R^2 2^{-\lambda} + \Pr[\,\mathbf{G}_4(A) = 1\,]$.

Finally, in $\mathbf{G}_5$ we keep track of how many **Qry** calls have previously been made, and use Q to keep track of those queries with indices in $\mathcal{I}$. Only these r queries can increment the error counters, but they do so if they cause an error in any of the currently-existing repr_j. To win, the adversary must therefore return an i so that all queries in Q produce errors when directed at repr_i. Assuming without loss of generality that the adversary does not add any elements to the set which it knows to be false positives, this condition is sufficient as well as necessary. Given that an adversary is successful in $\mathbf{G}_3$, it is successful in $\mathbf{G}_4$ with probability at least $\binom{q_T+q_U}{r}^{-1}$ since that is the chance of randomly selecting r successful queries from the total collection of $q_T + q_U$ queries. Then we have $\mathbf{Adv}^{\text{err-up}}_{\Pi_{\text{kbf}},r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + q_R^2 2^{-\lambda} + \binom{q_T+q_U}{r} \Pr[\,\mathbf{G}_4(A) = 1\,]$. But since the salts are unique for each representation, the probability of winning in each representation is independent of the probability of winning in the others, and the so $\Pr[\,\mathbf{G}_4(A) = 1\,]$ is just q_R times the standard (non-adpative) Bloom filter bound, yielding the final result of:

$$\mathbf{Adv}^{\text{err-up}}_{\Pi_{\text{kbf}},r}(A) \leq \mathbf{Adv}^{\text{prf}}_{F}(B) + \frac{q_R^2}{2^{\lambda}} + q_R \binom{q_T + q_U}{r} p(k, m, n + r)^r .$$

□ □

We can use this final and best bound to demonstrate some of the possible bounds with various parameter settings. Figure ? shows the upper bound for a various combination of parameters, starting from the default parameters of $k = 4$, $m = 1024$, $n = 100$, $r = 10$, $q_R = 1$, $q_T = 100$, and $q_U = 100$. The most significant factor in the adversarial advantage is q_R, with even small increases producing a drastic increase in the adversarial advantage. Since this is the only part of the error bound that is not clearly associated with an attack, it is possible that this term could be reduced or eliminated.



5 COUNTING FILTER RESULTS

Next, we consider the case of a counting Bloom filter. In this setting, an update to add an element increments each of the counters mapped to by the hash functions, with some maximum ceiling c above which the counter cannot increase. A Bloom filter would then just be a counting Bloom filter with a maximum counter value of 1, but we make the additional change that a counting Bloom filter can perform delete operations which work by hashing the element and decrementing the counters at those indices.

Because a counting filter without deletion behaves identically (in terms of responding to queries) to a standard Bloom filter, any attack on a Bloom filter can be trivially converted into an attack against a counting filter simply by performing the exact same operations in the same order. This means that the adversarial advantage against a counting filter, under any of the correctness notions and using any parameters, is bounded below by the advantage against a Bloom filter with the same parameters.

This lower bound is not tight, however, since there are some attacks which exploit the more powerful **Up** oracle provided to the adversary in the case of a counting filter. Consider the public-representation case for an example, where we suppose the structure makes use of a per-representation salt. With a counting filter, the adversary is able to construct representations of arbitrary singleton sets using a single **Rep** call and multiple **Up** calls. They begin the game by using **Rep** to initialize an empty filter. Then, for some each element $x_1, \ldots, x_{|\mathcal{S}|}$ that the adversary wishes to test, they may first use **Up** to insert the element and then use a second **Up** call to delete the element, returning again to an empty filter. After determining the representation of each singleton subset of $\mathcal{S}$, they can again find $\mathcal{T}, \mathcal{R} \subseteq \mathcal{S}$ offline computations, where each element of $\mathcal{R}$ is a false positive for the representation of $\mathcal{T}$. Since there is only a single representation being manipulated, the salt is in fact constant across the entire experiment. This attack requires a single **Rep** call to initialize the filter, $2|\mathcal{S}|$ calls to **Up** to test the elements plus an additional $|\mathcal{T}|$ calls to **Up** to reinsert the elements found to produce false positives, and $|\mathcal{R}|$ calls to **Qry** to win the experiment.

In order to achieve good security for a counting filter, it is necessary to use the thresholding assumption that insertions are rejected when the proportion of nonzero counters in the filter reaches some limit p. Without this assumption, the adversary can attempt to optimize the number of nonzero counters while keeping the size of the filter (in terms of the number of elements stored in the underlying set) below some constant maximum value. For example, if some x is known to be a false positive for two different sets $\mathcal{S}_1$ and $\mathcal{S}_2$, this increases the probability that elements of $\mathcal{S}_1$ are false positives for the filter representation of $\mathcal{S}_2$ and vice versa. However, using the assumption that there is a limit on the allowed number of nonzero counters, we can derive a false positive bound similar to that of a standard Bloom filter.

Theorem 5.1 (Correctness Bound for Counting Filters). *Fix integers $k, m, n, \lambda, r \geq 0$, a threshold ratio $p \in [0, 1]$, and an error function d. For every $t, q_R, q_T, q_U, q_H \geq 0$, it holds that*

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_{\text{sbf}},r}(t, \quad q_R, \quad q_T, q_U, q_H) \leq \\ q_R \cdot \left[\frac{q_H}{2^\lambda} + \binom{q_T}{\mu} \left(p + \frac{k\mu}{m} \right)^{k\mu} \right],$$

where $\mu = \lceil r / \max(k \cdot d(0,1), d(1,0)) \rceil$

Proof. Applying lemma 3.1 and lemma 3.2, we reduce the ERR-UP(S) experiment to the ERR-UP(S)1 experiment with a salted hash function replaced by true random sampling from $[m]$ for each of the k distinct hash values. This is given in $\mathbf{G}_0$. In $\mathbf{G}_1$, we account for the possibility that the adversary can glean information about the overlap between sets by querying each of them separately. In particular, we give the adversary an additional oracle **Int** that returns the number of locations in which two elements overlap. However, we constrain **Int** to only return an answer when the element has previously been sent to some other oracle.

Because new **Qry** and **Up** calls are based on random sampling, this does not provide the adversary with any additional information about how additional elements which might be queried, inserted, or deleted in the future might behave. However, any element which is sent to **Qry** or **Up** which acts as a false positive will be immediately recognized as such. Because of this, our next step will be to move to a game $\mathbf{G}_2$ where any insertion of a false positive immediately gives the adversary credit for the worst possible error that could be caused by either inserting or deleting that element.

Define $\delta = \max(k \cdot d(0,1), d(1,0))$ and $\mu = \lceil r/\delta \rceil$. In $\mathbf{G}_2$, we modify the game so that producing a false positive increments *err* by δ, but so that elements cannot be removed from the set. Because false negatives can only be produced by deleting false positives, and the deletion of n false positives can only produce as many as nk false negatives, this means that the adversary has no need to cause false negatives. Doing so can only decrease the chances of producing additional errors by reducing the values of counters in the filter, and so this does not decrease the adversary's ability to produce errors. Furthermore, we increase the allowed number of nonzero counters from mp to $mp + k\lceil r/\delta \rceil$. The only way in which not being able to remove false positives can decrease the adversary's effectiveness is if the maximum capacity of the filter is reached. Since each false positive removed causes k nonzero counters to be returned to zero, and the adversary stops after accumulating r errors, an allowance of an extra $k\lceil r/\delta \rceil$ nonzero counters is enough to make up for this.

Since each new element queried from outside $\mathcal{S}$ has its corresponding indices determined by a true random function, the adversary can do no better than maximizing the number of nonzero counters in its attempt to produce false positives. Assuming the adversary is able to achieve a maximum-capacity number of nonzero counters, the probability of any particular counter being nonzero is $(mp + k\lceil r/\delta \rceil)/m = p + \frac{k}{m}\lceil \frac{r}{\delta} \rceil$. Over k total hashes, the probability of a false positive is $\left((mp + k\lceil r/\delta \rceil)/m = p + \frac{k}{m}\lceil \frac{r}{\delta} \rceil\right)^k$. Since the adversary needs to accumulate $\lceil \frac{r}{\delta} \rceil$ errors over the course of q_T queries, the probability of the adversary succeeding in this game is

$$\Pr[\, \mathbf{G}_2(A) = 1 \,] = \binom{q_U + q_T}{\mu} \left(p + \frac{k\mu}{m} \right)^{k\mu}$$

Therefore the advantage of the adversary in the original game is

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s,r}(A) \leq q_R \cdot \left(\frac{q_H}{2^\lambda} + \binom{q_T}{\mu} \left(p + \frac{k\mu}{m} \right)^{k\mu} \right).$$

□

□

$\underline{\mathbf{G}_0(A)}$

$\mathcal{S} \twoheadleftarrow A^H$; $C \leftarrow \emptyset$; $\mathcal{B} \leftarrow \mathcal{S}$; $err \leftarrow 0$
repr $\twoheadleftarrow$ Rep[**Hash**]($\mathcal{S}$)
$\perp \twoheadleftarrow A^{\mathbf{Hash},\mathbf{Qry},\mathbf{Up},\mathbf{Int}}$
return $(err \geq r)$

oracle $\underline{\mathbf{Qry}(x)}$:
$\mathcal{B} \leftarrow \mathcal{B} \cup x$
if $x \in C$ then return $\perp$
$C \leftarrow C \cup \{x\}$
$a \leftarrow$ Qry[**Hash**](repr, x)
if $a \neq [x \in \mathcal{S}]$ then $err \leftarrow err + 1$
return a

oracle $\underline{\mathbf{Up}(x, b)}$:
$\mathcal{B} \leftarrow \mathcal{B} \cup x$
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[**Hash**](repr, qry_x)
if $x \in C$ and $b = a \neq [x \in \mathcal{S}]$ then
    $err \leftarrow err - 1$
if $b = 1$ then
    $\mathcal{S} \leftarrow \mathcal{S} + \{x\}$
else
    $\mathcal{S} \leftarrow \mathcal{S} - \{x\}$
repr $\leftarrow$ Up[**Hash**](repr, $\text{up}_{x,b}$)
return $\perp$

oracle $\underline{\mathbf{Hash}(x)}$:
$\vec{h} \twoheadleftarrow [m]^2$; $\vec{v} \leftarrow$ Fn($\vec{h}$)
if $T[x]$ is defined then $\vec{v} \leftarrow T[x]$
$T[x] \leftarrow \vec{v}$; return $\vec{v}$

$\underline{\mathbf{G}_1(A)}$

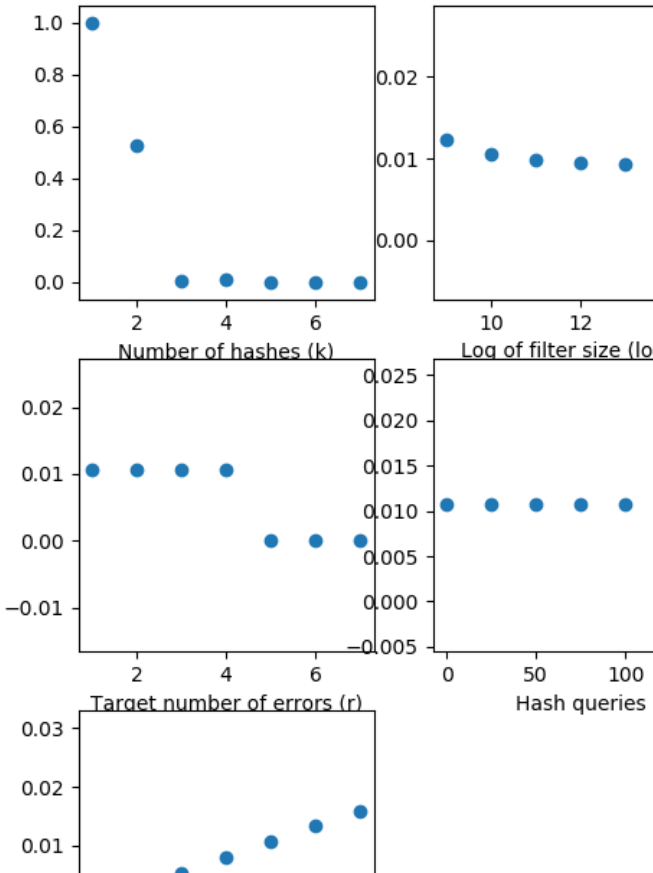
oracle $\underline{\mathbf{Int}(x, y)}$:
if $x \notin \mathcal{S}$ or $y \notin \mathcal{S}$ then
    return $\perp$
$i \leftarrow 0$
for h in **Hash**(x) do
    if h in **Hash**(y) then $i \leftarrow i + 1$
return i

$\underline{\mathbf{G}_2(A)}$

oracle $\underline{\mathbf{Up}(x)}$:
if **Qry**(x) and $x \notin \mathcal{S}$ then
    $err \leftarrow err + \max(k \cdot d(0,1), d(1,0))$
$\mathcal{B} \leftarrow \mathcal{B} \cup x$
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[H](repr, qry_x)
if $x \in C$ and $a \neq [x \in \mathcal{S}]$ then
    $err \leftarrow err - 1$
$\mathcal{S} \leftarrow \mathcal{S} + \{x\}$
repr $\leftarrow$ Up[H](repr, $\text{up}_{x,b}$)
return $\perp$

Figure 5: Games 0–3 for proof of Theorem 5.1.

For concrete error bounds, we assume that $d(0, 1) = d(1, 0) = 1$, so that $\mu = \lceil r/k \rceil$. The default parameters are the same as in the case of an ordinary Bloom filter.



6 CUCKOO FILTER RESULTS

Cuckoo filters provide a way of filtering elements with lower space overhead than counting filters while still allowing for deletion. While they are not as closely related to Bloom filters as counting filters are, and so we do not have the precise lower bound that *any* attack against a Bloom filter also works against a cuckoo filter, it is true that since cuckoo filters are set-membership structures the specific bounds given in the Bloom filter section apply here. To establish acceptable upper bounds, we consider the case of a salted cuckoo filter with private representations. By 'salted', we mean that both the hashing and the fingerprinting makes use of a salt chosen on a per-represntation basis. Using this definition, we can make use of the lemmas that allow us to talk about the structure in the single-represntation case with true random functions. A bound for the false positive rate in cuckoo filters is given by $\binom{n}{2k+1}\left(\frac{2}{2^f \cdot m}\right)^{2k}$ for a set of n elements in a filter with m buckets holding up to k entries each, with fingerprint size f.

Theorem 6.1 (Correctness Bound for Cuckoo Filters). *Fix integers $n, b, k, \lambda, r \geq 0$, where $m \geq \lambda$. For every $t, q_R, q_T, q_U, q_H \geq 0$ such that $q_T \geq r$, it holds that*

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi,r}(t, q_R, q_T, q_U, q_H) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + \binom{q_T + q_U}{r} P(n, m, b, k+r)^r\right] \qquad \mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s,r}(A) \leq q_R \cdot \left(\frac{q_H}{2^\lambda} + \binom{q_U + q_T}{r} P(n, m, b, k+r)^r\right).$$

where $P(n, m, b, k)$ is the standard, non-adaptive false-positive probability on a cuckoo filter with those parameters.

Proof. Again we can use lemma 3.1 and lemma 3.2 to reduce to the case of ERR-UP(S)1 with true random functions in place of salted hash functions. We use this as the first game $\mathbf{G}_0$, so that $\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s,r}(A) \leq q_R \cdot \left(\frac{q_H}{2^\lambda} + \Pr[\,\mathbf{G}_0(A) = 1\,]\right)$.. □

Because membership tests rely on objects having the same fingerprint, and a single bucket may contain multiple copies of the same fingerprint, deleting n false positives can only produce up to n false negatives, one per fingerprint removed. Unless the filter becomes full, it is therefore never helpful for an adversary to delete a false positive, since it cannot increase the number of errors. We may then assume, subject to the condition that the adversary never has to worry about the filter becoming full, that the adversary only makes insertion updates. Furthermore, we may assume the adversary never inserts elements which are already in the set, or elements which have already been queried and found to be false positives, since these also cannot increase the number of errors.

We want to show that alternating between sequences of queries and updates is no better than making updates first and queries afterwards. We have shown that the adversary only makes two types of updates: inserting unqueried elements which are not in the set and inserting known true negatives.

We move to a second game where each **Qry** call also inserts that element. Additionally, the penalty for adding known false positives is removed. To avoid penalizing the adversary by producing an overly-full filter which rejects further insertions, we also increase the size of each bucket by r. Because we may assume without loss of generality that the adversary stops after finding r errors, only r false positives will be inserted and so these extra r slots in each bucket are sufficient to ensure that if the buckets in the original game did not fill then the buckets in this game will not fill. Furthermore, we modify **Up** to query for the element before inserting it. This cannot produce a worse result for the adversary, and so $\Pr[\,\mathbf{G}_0(A) = 1\,] \leq \Pr[\,\mathbf{G}_1(A) = 1\,]$. Since **Qry** and **Up** calls are identical and each call is independent of all previous calls, the adversary's probability of success is exactly the probability of accumulating r random errors from queries. Each query is a standard random query to a cuckoo filter, so by the binomial theorem we can get a bound of $\binom{q_T+q_U}{r} P(b, k+r)^r$. This gives the result: □

As with the other data structures, the graphs show how each of the parameters affects the adversarial advantage. The default parameters are $k = 4$, $m = 256$, $n = 100$, $f = 4$, $r = 10$, $q_R = 1$, and $q_H = q_T = q_U = 100$.

$\underline{\mathbf{G}_0(A)}$

$\mathcal{S} \twoheadleftarrow A^H$; $C \leftarrow \emptyset$; $err \leftarrow 0$
repr $\twoheadleftarrow$ Rep[**Hash**]($\mathcal{S}$)
$\perp \twoheadleftarrow A^{\text{Hash,Qry,Up}}$
return $(err \geq r)$

oracle **Qry**(x):
if $x \in C$ then return $\perp$
$C \leftarrow C \cup \{x\}$
$a \leftarrow$ Qry[**Hash**](repr, qry_x)
if $a \neq [x \in \mathcal{S}]$ then $err \leftarrow err + 1$
return a

oracle **Up**(x, b):
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[**Hash**](repr, qry_x)
if $x \in C$ and $a \neq [x \in \mathcal{S}]$ then
  $err \leftarrow err - 1$
if $b = 1$ then
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$
else
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x\}$
repr $\leftarrow$ Up[**Hash**](repr, $\text{up}_{x,b}$)
return $\perp$

oracle **Hash**(x):
$\vec{h} \twoheadleftarrow [m]^2$; $\vec{v} \leftarrow$ Fn($\vec{h}$)
if $T[x]$ is defined then $\vec{v} \leftarrow T[x]$
$T[x] \leftarrow \vec{v}$; return $\vec{v}$

$\underline{\mathbf{G}_1(A)}$

oracle **Qry**(x):
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[**Hash**](repr, qry_x)
if $a \neq [x \in \mathcal{S}]$ then
  $err \leftarrow err + d(a, [x \in \mathcal{S}])$
if $b = 1$ then
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$
else
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x\}$
repr $\leftarrow$ Up[**Hash**](repr, $\text{up}_{x,b}$)
return $\perp$

oracle **Up**(x, b):
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[**Hash**](repr, qry_x)
if $a \neq [x \in \mathcal{S}]$ then
  $err \leftarrow err + d(a, [x \in \mathcal{S}])$
if $b = 1$ then
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{x\}$
else
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{x\}$
repr $\leftarrow$ Up[**Hash**](repr, $\text{up}_{x,b}$)
return $\perp$

Figure 6: Games 0–3 for proof of Theorem 6.1.

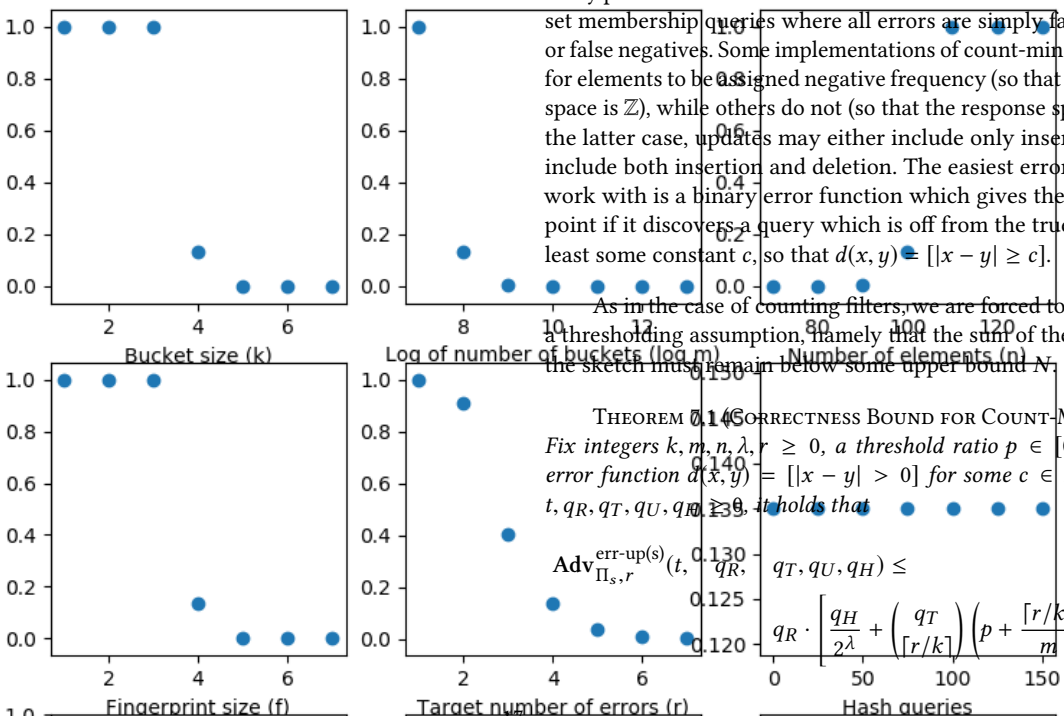
7 COUNT-MIN SKETCH RESULTS

The count-min sketch data structure comes in several varieties which each have slightly different security properties. There are also many possible error functions which can be considered, unlike with set membership queries where all errors are simply false positives or false negatives. Some implementations of count-min sketch allow for elements to be assigned negative frequency (so that the response space is $\mathbb{Z}$), while others do not (so that the response space is $\mathbb{N}$). In the latter case, updates may either include only insertion or may include both insertion and deletion. The easiest error function to work with is a binary error function which gives the adversary 1 point if it discovers a query which is off from the true value by at least some constant c, so that $d(x, y) = [|x - y| \geq c]$.

As in the case of counting filters, we are forced to make use of a thresholding assumption, namely that the sum of the counters in the sketch must remain below some upper bound N.

Theorem 7.1 (Correctness Bound for Count-Min Sketch). *Fix integers $k, m, n, \lambda, r \geq 0$, a threshold ratio $p \in [0, 1]$, and an error function $d(x, y) = [|x - y| > 0]$ for some $c \in \mathbb{N}$. For every $t, q_R, q_T, q_U, q_H \geq 0$, it holds that*

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s, r}(t, q_R, q_T, q_U, q_H) \leq q_R \cdot \left[\frac{q_H}{2^\lambda} + \binom{q_T}{\lceil r/k \rceil} \left(p + \frac{\lceil r/k \rceil}{m} \right)^{k \lceil r/k \rceil} \right],$$



$\underline{\mathbf{G}_0(A)}$

$\mathcal{S} \twoheadleftarrow A^H$; $C \leftarrow \emptyset$; $\mathcal{B} \leftarrow \mathcal{S}$; $err \leftarrow 0$
repr $\twoheadleftarrow$ Rep[**Hash**]($\mathcal{S}$)
$\perp \twoheadleftarrow A^{\mathbf{Hash},\mathbf{Qry},\mathbf{Up},\mathbf{Int}}$
return $(err \geq r)$

oracle $\underline{\mathbf{Qry}(x)}$:
$\mathcal{B} \leftarrow \mathcal{B} \cup x$
if $x \in C$ then return $\perp$
$C \leftarrow C \cup \{x\}$
$a \leftarrow$ Qry[**Hash**](repr, x)
if $a \neq [x \in \mathcal{S}]$ then $err \leftarrow err + 1$
return a

oracle $\underline{\mathbf{Up}(x, b)}$:
$\mathcal{B} \leftarrow \mathcal{B} \cup x$
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[**Hash**](repr, qry_x)
if $x \in C$ and $b = a \neq [x \in \mathcal{S}]$ then
  $err \leftarrow err - 1$
if $b = 1$ then
  $\mathcal{S} \leftarrow \mathcal{S} + \{x\}$
else
  $\mathcal{S} \leftarrow \mathcal{S} - \{x\}$
repr $\leftarrow$ Up[**Hash**](repr, $\text{up}_{x,b}$)
return $\perp$

oracle $\underline{\mathbf{Hash}(x)}$:
$\vec{h} \twoheadleftarrow [m]^2$; $\vec{v} \leftarrow$ Fn($\vec{h}$)
if $T[x]$ is defined then $\vec{v} \leftarrow T[x]$
$T[x] \leftarrow \vec{v}$; return $\vec{v}$

$\underline{\mathbf{G}_1(A)}$

oracle $\underline{\mathbf{Int}(x, y)}$:
if $x \notin \mathcal{S}$ or $y \notin \mathcal{S}$ then
  return $\perp$
$i \leftarrow 0$
for h in **Hash**(x) do
  if h in **Hash**(y) then $i \leftarrow i + 1$
return i

$\underline{\mathbf{G}_2(A)}$

oracle $\underline{\mathbf{Up}(x)}$:
if **Qry**(x) and $x \notin \mathcal{S}$ then
  $err \leftarrow err + \max(k \cdot d(0,1), d(1,0))$
$\mathcal{B} \leftarrow \mathcal{B} \cup x$
$C \leftarrow \emptyset$
$a \leftarrow$ Qry[H](repr, qry_x)
if $x \in C$ and $a \neq [x \in \mathcal{S}]$ then
  $err \leftarrow err - 1$
$\mathcal{S} \leftarrow \mathcal{S} + \{x\}$
repr $\leftarrow$ Up[H](repr, $\text{up}_{x,b}$)
return $\perp$

Figure 7: Games 0–3 for proof of Theorem 7.1.

Proof. Applying lemma 3.1 and lemma 3.2, we reduce the ERR-UP(S) experiment to the ERR-UP(S)1 experiment with a salted hash function replaced by true random sampling from $[m]$ for each of the k distinct hash values. This is given in $\mathbf{G}_0$. In $\mathbf{G}_1$, we account for the possibility that the adversary can glean information about the overlap between sets by querying each of them separately. In particular, we give the adversary an additional oracle **Int** that returns the number of locations in which two elements overlap. However, we constrain **Int** to only return an answer when the element has previously been sent to some other oracle.

Because new **Qry** and **Up** calls are based on random sampling, this does not provide the adversary with any additional information about how additional elements which might be queried, inserted, or deleted in the future might behave. However, any element which is sent to **Qry** or **Up** which acts as a false positive will be immediately recognized as such. Because of this, our next step will be to move to a game $\mathbf{G}_2$ where any insertion of a false positive immediately gives the adversary credit for the worst possible error that could be caused by either inserting or deleting that element.

In $\mathbf{G}_2$, we modify the game so that producing a false positive increments err by k, but so that elements cannot be removed from the set. Because underestimation errors can only be produced by deleting 'false positives' (i.e. elements that should have 0 frequency but which are reported to have positive frequency), and the deletion of n false positives can only produce as many as nk underestimation errors, this means that the adversary has no need to cause underestimation errors. Doing so can only decrease the chances of producing additional errors by reducing the values of counters in the sketch, and so this does not decrease the adversary's ability to produce errors. Furthermore, we increase the allowed number of nonzero counters from mp to $mp + k\lceil r/k \rceil$. The only way in which not being able to remove false positives can decrease the adversary's effectiveness is if the maximum capacity of the sketch is reached. Since each false positive removed causes at most 1 nonzero counters to be returned to zero, and the adversary stops after accumulating r errors, an allowance of an extra $\lceil r/k \rceil$ nonzero counters per row is enough to make up for this.

Since each new element queried from outside $\mathcal{S}$ has its corresponding indices determined by a true random function, the adversary can do no better than maximizing the number of nonzero counters in its attempt to produce false positives. Assuming the adversary is able to achieve a maximum-capacity number of nonzero counters, the probability of any particular counter being nonzero is $(mp + \lceil r/k \rceil)/m = p + \frac{1}{m}\lceil \frac{r}{k} \rceil$. Over k total hashes, the probability of a false positive is $((mp + \lceil r/k \rceil)/m)^k = \left(p + \frac{1}{m}\lceil \frac{r}{k} \rceil\right)^k$. Since the adversary needs to accumulate $\lceil \frac{r}{k} \rceil$ errors over the course of q_T queries, the probability of the adversary succeeding in this game is

$$\Pr[\,\mathbf{G}_2(A) = 1\,] = \binom{q_U + q_T}{\lceil r/k \rceil} \left(p + \frac{\lceil r/k \rceil}{m}\right)^{k\lceil r/k \rceil}$$

Therefore the advantage of the adversary in the original game is

$$\mathbf{Adv}^{\text{err-up(s)}}_{\Pi_s,r}(A) \le q_R \cdot \left(\frac{q_H}{2^\lambda} + \binom{q_T}{\lceil r/k \rceil} \left(p + \frac{\lceil r/k \rceil}{m} \right)^{k\lceil r/k \rceil} \right).$$

□

□

8 CONCRETE DATA STRUCTURES

(8.1: David to-do: This text needs to be broken up and massaged in to appropriate places in the document, e.g., in the sections on specific structures. Some of it I will lift to the intro for future work, limitations of our formalisms.) In general, each probabilistic data structure has some bound on the error size per query, assuming the adversary is not fully adaptive. In each case we want to show that allowing the adversary full adaptivity does not significantly increase the error rate. Generally, but not always, we have a response space $\mathcal{R}$ and a data object space $\mathcal{D} \subseteq \mathit{Func}(\mathcal{X}, \mathcal{R})$ (8.2: TS says: This is not intuitively obvious; people are used to thinking about sets, data streams, etc. This needs to be explained.) for some universe $\mathcal{X}$, so that each object in the universe is associated with a single correct response. The usual query space consists of point functions qry_x for $x \in \mathcal{X}$, so that $\mathsf{qry}_x(\mathcal{S}) = \mathcal{S}(x)$ (8.3: TS says: This only makes sense under the data-object-as-function interpretation, and (again) is not intutitive.) for all $\mathcal{S} \in \mathcal{D}$. The update space at least consists of insertions to add a single element, but may also include deletions of single elements.

In each of the following cases, we consider four variations, where each structure may be constructed with or without a randomly-chosen per-representation salt and with or without a private key. We also sometimes consider the possibility of adding a 'threshold' to detect when a structure is too full. For example, we may modify the Up algorithm of a counting filter to fail if it is called to insert an element when the sum over all the counters in the filter has reached some threshold value. This variation can in some cases provide substantially better error bounds by preventing the adversary from exploiting more naive limits on the fullness of a filter, such as the number of elements in the set being represented.

A typical case occurs with standard Bloom filters [**?**]. In this case we have a response space $\mathcal{R} = \{0, 1\}$ and an object space $\mathcal{D} = [\mathcal{X}]^{\le n}$ consisting of all subsets of some universe $\mathcal{X}$ that have cardinality no greater than a constant n. The queries are standard membership queries for each element of $\mathcal{X}$, and the update space consists of insertions. An update to insert the element x is denoted by up_x. These data structures are constructed using a set of hash functions, each of which maps an object to a position in an array of bits. When an object is added to the filter, either during the filter's creation or during a later update, each of the bits the object is mapped to by each of the hash functions is set to 1. To make a membership query, one can simply hash the queried object and check whether all of the associated bits are 1. Updates may only add more elements to the set: with a traditional Bloom filter structure, deletion is impossible. Because of this, Bloom filters have no false negatives, and the size of the error a non-adaptive adversary is expected to create per query is simply equal to the false positive rate, which is on the order of $(1 - e^{-\frac{kn}{m}})^k$ for an m-bit array with k hash functions storing up to n values.

Counting bloom filters [**?**], or counting filters, allow deletion by broadening the data object space to $\mathcal{D} \subseteq \mathit{Func}(\mathcal{X}, \mathbb{Z})$. With insertion and deletion both possible, we denote an insertion update for x by $\mathsf{up}_{x,1}$ and a deletion update for x by $\mathsf{up}_{x,0}$. Under ideal conditions the range of the data objects is not just in $\mathbb{Z}$ but in $\mathbb{N}$ specifically, but negative multiplicities may be introduced by the deletion of objects which are not actually in the set. The queries for a counting Bloom filter are still binary, operating by projecting the underlying multiset down to an ordinary set which contains a given element if and only if the multiset contains that element with multiplicity at least 1. The introduction of a deletion operation means that false positives and false negatives are both possible.

Cuckoo filters [**?**] are use the same extension of the data object space and update space, still allowing for only set membership queries along with insertion and deletion updates. Though the structure itself is different, the syntax is the same as for counting filters..

Depending on the implementation, count-min sketch [**?**] may similarly use a data object space $\mathcal{D} \subseteq \mathit{Func}(\mathcal{X}, \mathbb{Z})$ or may use a smaller space $\mathcal{D} \subseteq \mathit{Func}(\mathcal{X}, \mathbb{N})$. In the former case we have two further sub-cases, the case where all updates increment the value associated with $x \in X$ and the case where updates may either increment or decrement the associated value (but not below 0). Additionally, count-min sketch supports multiple different types of queries, in each case yielding a response in $\mathbb{Z}$. For a point query, the difference between the query and the true value is bounded by $n\epsilon$ with probability $1 - \delta$, where n is the sum of the true frequencies of the stream elements. The maximum error of a single query is simply n, in the case that an element has never actually been added to the set but has incorrectly had all its counters incremented each of n times another element has been added, so the expected non-adaptive error size is bounded above by $n\epsilon(1-\delta)+n\delta = n(\delta+\epsilon-\delta\epsilon)$. Similarly, we find that the error size of an inner product query is bounded above by $n_1 n_2 \epsilon(1-\delta) + n_1 n_2 \delta = n_1 n_2(\delta + \epsilon - \delta\epsilon)$. Finally, range queries [...]

Stable Bloom filters [**?**] are an example of a structure in the Bloom filter family which our notions cannot work with easily. One way in which they are unique is in featuring a probabilistic update algorithm, causing objects in the filter to probabilistically decay over time. While our syntax accounts for that, the more significant difference is that a stable Bloom filter only has good accuracy guarantees once it has seen enough updates to 'stabilize'. Our current security notions do not encompass this type of conditional accuracy guarantee. (8.4: TS says: future work)

Many of these structures follow the same design philosophy in order to reduce false positives. Specifically, the value associated with an element (such as a single bit in the case of a set represented by a Bloom filter, or an integer in the case of a multiset represented by count-min sketch) is stored several times. When the data is to be retrieved, we check each of the locations the data has been stored

and take the minimal value across all the locations. For example, a Bloom filter returns 0 if any of the hash locations has a 0 bit, while a point query for count-min sketch returns the minimum value across all stored locations. This is as strict an approach as possible when attempting to avoid overestimating the number of times an element has been added to the underlying (multi)set, but can cause problems in an adversarial environment. The tendency to underestimate the number of times an element is represented in the (multi)set means that an adversary can cause errors by decrementing only a single one of the several values associated with the element.

Consider the case of a counting Bloom filter representing a multiset $\mathcal{S}$. When the filter is constructed, each of the $x \in \mathcal{S}$ causes several counters in the filter to be incremented. When a query is made for an element x', we hash x' several times and check whether the value associated with each of the hash values is positive, returning 1 if so and 0 otherwise. If $x' \notin \mathcal{S}$ but x' does return 1 when queried, we can update the filter to decrement the multiplicity of x', decrementing each of the hash values as well. For every 1 which is decremented to 0 by this operation, a false negative is created: an object which is contained in $\mathcal{S}$ but which will return 0 when queried. This potentially introduces as many false negatives as there are locations associated with an object. In the case of Bloom filters this is an optimal choice, since false negatives are impossible and we can simply optimize for the lowest possible chance of a false positive.

We can consider the possibility of a more symmetric data structure which balances between resistance to false positives and resistance to false negatives. For example, consider a 'balanced' counting Bloom filter which answers queries by checking whether the *majority* of associated values are positive, rather than checking whether all associated values are positive. For count-min sketch, we can consider the case of a 'count-median-sketch' which returns the mean or median of the associated values rather than the minimum. This variation has been defined previously, but we might also consider arbitrary count-f-sketches, where $f : \mathbb{Z}^k \to \mathbb{Z}$ is a function mapping the k counter values to a frequency estimate. (8.5: TS says: future work)

More generally, for any structure of this type with object space $\mathcal{D} \subset \mathit{Func}(\mathcal{U}, \mathcal{V})$ for some universe $\mathcal{U}$ of objects with associated values in $\mathcal{V}$, we can take a function $f : \mathcal{V}^* \to \mathcal{R}$ and use it to define qry : $\mathcal{U} \times \mathit{Func}(\mathcal{U}, \mathcal{V}) \to \mathcal{R}$. We evaluate the function by evaluating $f(v_0, \ldots)$ where the v_i are the several values associated with the element being queried. Each of the above structures uses $f = \min$, but many other functions could be used. (8.6: TS says: future work)

An important consideration for probabilistic data structures in the streaming setting is the question of when the structure is considered to be full. For example, a Bloom filter of a fixed size using some number of hash functions can only guarantee good performance up to some maximum number of elements contained. In the streaming setting, we may not know in advance exactly when this threshold is reached. Allowing the insertion of arbitrarily many elements is undesirable because an overfull filter will produce large numbers of false positives. In addition to the straightforward possibility of keeping track of a counter of the number of elements considered in order to not ensure a maximum is surpassed, we consider an alternative where some property of the structure itself is kept under a fixed threshold. For example, with Bloom filters we might require that the total number of bits set to 1 be no more than some proportion p of the total bits, while with counting filters we might do the same for the total number of nonzero counters. (8.7: TS says: This is now incorporated into the main part of the document, right?)

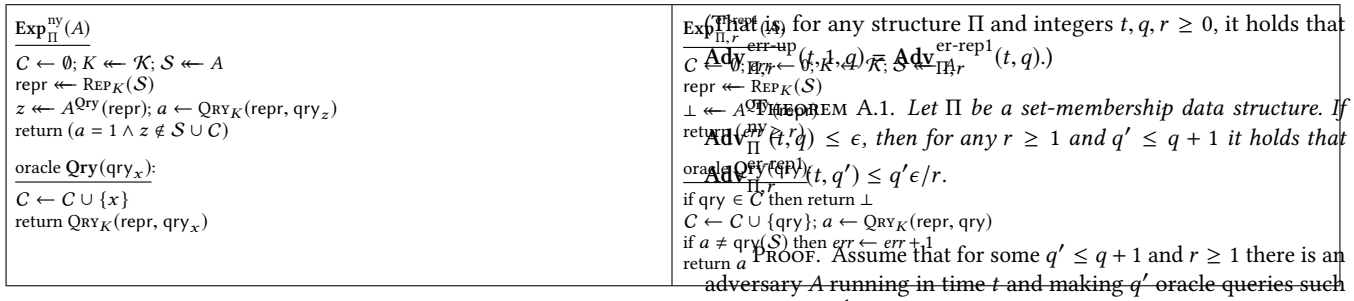


Figure 8: Left: The Naor-Yogev (NY) definition of correctness for set-membership structure $\Pi = (\text{Rep}, \text{Qry})$. Right: The ER-REP1 notion for (set-membership) structure $\Pi = (\text{Rep}, \text{Qry})$. (Equivalent to ERR-UP with $q_R = 1$.)

A COMPARISON TO NAOR-YOGEV

In this appendix we compare the Naor-Yogev definition of correctness [**?**] and ours. (We focus only on data structures with what they call *steady representations*, which is the only kind of data structure we study in this work.) Although their definitions is specific to the case of Bloom filters, and do not incorporate keys, we generalize it in the natural way. (A.1: TS says: I don't like this. We say in the text, later on, that NY *only* covers secret representations, but the psuedocode experiment that supposedly formalizes the NY notion (in our notation) hides this fact! It makes it look like we are less different from NY than we are. Also, NY needs to be called out on their shit... the public vs. secret representation issue is especially important in the mutable setting!) Let $\Pi = (\text{Rep}, \text{Qry})$ be a set-membership structure for $\mathcal{X}$ with key space $\mathcal{K}$. Consider the NY experiment defined in the left panel of Figure 8, associated with Π and an adversary A. First, A outputs a set $\mathcal{S}$ of size n. Next, a key K is chosen and the representation algorithm is executed on K and $\mathcal{S}$, resulting in repr. Then A is executed with input repr and with access to an oracle **Qry** as in our definition of correctness. Finally, A outputs a value $z \in \mathcal{X}$; it succeeds if $z \notin \mathcal{S}$, it never previously queried qry$_z$ to **Qry**, and $\text{Qry}_K(\text{repr}, \text{qry}_z) = 1$. We define the advantage of A in attacking Π as

$$\mathbf{Adv}^{\text{ny}}_{\Pi}(A) \stackrel{\text{def}}{=} \Pr\left[\, \mathbf{Exp}^{\text{ny}}_{\Pi}(A) = 1 \,\right] ,$$

and let $\mathbf{Adv}^{\text{ny}}_{\Pi}(t, q)$ denote the maximum of this value, taken over all A running for at most t steps and making at most q oracle queries.

We remark that there are two important differences between this definition and [**?** , Definition 2.4]. First, the adversary in Figure 8 is given the representation repr, but not the key, in the second stage of its attack; in contrast, Naor-Yogev (implicitly) assume the entire data structure is private. Second, we allow the attacker in Figure 8 to choose the set $\mathcal{S}$, whereas Naor-Yogev treat it as a parameter of the experiment.

With these modifications in place, we have a basis for comparing the NY definition to our own ERR-UP. For the sake of exposition, we will restrict ourselves to the case of $q_R = 1$. In the right-hand panel of Figure 8, we define a game ER-REP1, which is equivalent to ERR-UP when the adversary is restricted to just one **Rep** query. (That is, for any structure Π and integers $t, q, r \geq 0$, it holds that $\mathbf{Adv}^{\text{err-up}}_{\Pi,r}(t, 1, q) = \mathbf{Adv}^{\text{er-rep1}}_{\Pi,r}(t, q)$.)

Theorem A.1. *Let Π be a set-membership data structure. If $\mathbf{Adv}^{\text{ny}}_{\Pi}(t, q) \leq \epsilon$, then for any $r \geq 1$ and $q' \leq q + 1$ it holds that $\mathbf{Adv}^{\text{er-rep1}}_{\Pi,r}(t, q') \leq q'\epsilon/r$.*

Proof. Assume that for some $q' \leq q + 1$ and $r \geq 1$ there is an adversary A running in time t and making q' oracle queries such that $\mathbf{Adv}^{\text{er-rep1}}_{\Pi,r}(A) > q'\epsilon/r$. (Note we may assume that A always makes exactly q' queries without loss of generality.) This means that with probability at least $q'\epsilon/r$ in an execution of $\mathbf{Exp}^{\text{er-rep1}}_{\Pi,r}(A)$, we have that A makes at least r distinct queries to **Qry** for which an incorrect answer is returned. Let A' be the algorithm that simply runs A, but chooses uniformly one of the q' queries of A to its **Qry** oracle and outputs that query as its final output. Then with probability at least $r/q' \cdot (q'\epsilon/r) = \epsilon$ the query chosen by A' leads to an incorrect answer, and was not previously asked to the **Qry** oracle. Since the running time of A' is at most t, and it makes at most $q' - 1 \leq q$ queries to its oracle, this is a contradiction. □ □

We remark that the above is tight, at least for $r = 1$. Specifically, consider a scheme in which every query is independently answered incorrectly with probability ϵ. Such a scheme satisfies $\mathbf{Adv}^{\text{ny}}_{\Pi}(t, q) \leq \epsilon$ for any t, q, however an adversary making $q = 1/\epsilon$ queries has constant advantage with respect to our correctness definition (for $r = 1$).

In the other direction, we show that correctness for $r = 1$ easily implies correctness with respect to the Naor-Yogev definition.

Theorem A.2. *Let Π be a set-membership structure. If $\mathbf{Adv}^{\text{er-rep1}}_{\Pi,1}(t, q) \leq \epsilon$, then $\mathbf{Adv}^{\text{ny}}_{\Pi}(t, q - 1) \leq \epsilon$.*

Proof. Assume there is an adversary A running in time t and making at most $q - 1$ oracle queries such that $\mathbf{Adv}^{\text{ny}}_{\Pi}(A) > \epsilon$. Let A' be the algorithm that simply runs A, forwarding the oracle queries of A to its own oracle, until A terminates with output z; then, A' sends qry$_z$ to **Qry**. It is immediate that A' makes at most q oracle queries, and $\mathbf{Adv}^{\text{er-rep1}}_{\Pi,1}(A') \geq \mathbf{Adv}^{\text{ny}}_{\Pi}(A)$, a contradiction. □ □

For $r > 1$, however, we have the following separation:

Theorem A.3. *For every integer $r \geq 1$ and set $\mathcal{X}$, there is a set-membership structure Π for $\mathcal{X}$ for which $\mathbf{Adv}^{\text{er-rep1}}_{\Pi,r+1}(t, q) = 0$ for all integers $t, q \geq 0$, but $\mathbf{Adv}^{\text{ny}}_{\Pi}(O(1), 0) = 1$.*

Proof. Fix an integer $r \geq 1$, a set $\mathcal{X}$, and distinct values $x_1, \ldots, x_r \in \mathcal{X}$. Define $\Pi = (\text{Rep}, \text{Qry})$ so that $\text{Rep}(\mathcal{S})$ outputs $\mathcal{S}$ and $\text{Qry}(\mathcal{S}, \text{qry}_y)$ outputs $\text{qry}_y(\mathcal{S})$ if $y \notin \{x_1, \ldots, x_r\}$, but outputs $1 - \text{qry}_y(\mathcal{S})$ otherwise. This scheme always answers incorrectly for r fixed queries, and answers correctly for every other query. The claim follows. □ □

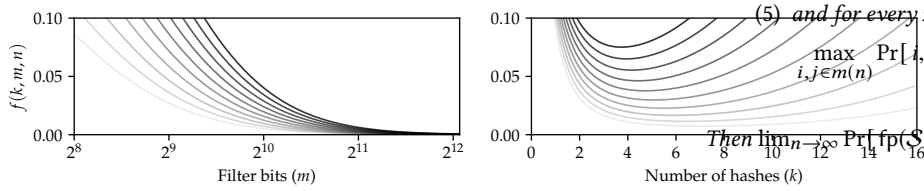


Figure 9: Function $f(k, m, n) = (1-e^{-kn/m})^k$ for various values of k, m, and n. In both plots, darker lines are for larger set sizes (n); values range from $n = 100$ to 200. Left: varying filter length (m) and $k = 4$ (the number of hash functions used in Squid). Right: varying number of hashes (k) and $m = 1024$.

B THE STANDARD BLOOM FILTER BOUND

This appendix summarizes the results of Kirsch and Mitzenmacher [**?**] for the false-positive probability of Bloom filters using double hashing. Our presentation is less general than theirs, but suffices for understanding the results in our paper.

Additional notation. If $\vec{v}$ is a vector, then let $\mathsf{mset}(\vec{v})$ denote the multiset comprised of its elements. Following [**?**], if $\mathcal{M}$ is a multiset, we write $i, i \in \mathcal{M}$ to mean that i appears in $\mathcal{M}$ at least twice.

Hashing schemes. Let $\mathcal{X}$ be a set. A *hashing scheme* Γ for $\mathcal{X}$ is a quadruple $(\textsc{Hash}, m, k, \mathcal{N})$. The last element is an infinite set $\mathcal{N} \subseteq \mathbb{N}$ denoting the permitted *set sizes*. (We do not require that $\mathcal{N} = \mathbb{N}$.) Functions m and k specify the filter length $m(n)$ and number of hashes $k(n)$ respectively for set size n. The first element is a function $\textsc{Hash}\colon \mathbb{N} \times \mathcal{X} \to \mathbb{N}^*$ mapping a parameter $n \in \mathcal{N}$ and $x \in \mathcal{X}$ to a $k(n)$-vector of natural numbers, which represent bit locations in a filter of length $m(n)$. Formally, for every $n \in \mathcal{N}$, $\textsc{Hash}(n, \cdot)$ is a random function specifying a joint distribution on a collection of random variables $\{\textsc{Hash}(n, x)\colon x \in \mathcal{X}\}$. We associate to Γ, set $\mathcal{S} \subseteq \mathcal{X}$, and $z \in \mathcal{X} \setminus \mathcal{S}$ a *false positive event*, denoted $\mathsf{fp}(\mathcal{S}, z)$, which occurs if for every $j \in [k(n)]$ there exists some $x \in \mathcal{S}$ such that $\vec{v}_j \in \mathsf{mset}(\textsc{Hash}(n, x))$, where $\vec{v} = \textsc{Hash}(n, z)$ and $n = |\mathcal{S}|$.

Fix a set $\mathcal{X}$ and a hashing scheme $\Gamma = (\textsc{Hash}, m, k, \mathcal{N})$ for $\mathcal{X}$. Fix $z \in \mathcal{X}$ and a collection of subsets $\{\mathcal{S}_n\}_{n\in\mathcal{N}}$ of $\mathcal{X}$, where $z \notin \mathcal{S}_n$ and $|\mathcal{S}_n| = n$ for each $n \in \mathcal{N}$. The following says that, if Γ satisfies certain conditions, the false positive probability converges to the approximate bound of [**?**] as n increases.

Theorem B.1 (Lemma 4.1 of [**?**]). *Suppose there exist $\lambda, k \in \mathbb{N}$ and a function $\gamma(n) \in o(1/n)$ such that for every $n \in \mathcal{N}$, it holds that*

(1) *$k(n) = k$;*

(2) *$m(n) = O(n)$;*

(3) *$\{\textsc{Hash}(n, x)\colon x \in \mathcal{X}\}$ are independent and identically distributed;*

(4) *for every $x \in \mathcal{X}$, it holds that*

$$\max_{i\in m(n)} \left|\Pr[\, i \in \mathsf{mset}(\textsc{Hash}(n, x))\,] - \lambda/kn\right| = O(\gamma(n))\,;$$

(5) *and for every $x \in \mathcal{X}$, it holds that*

$$\max_{i,j\in m(n)} \Pr[\, i, j \in \mathsf{mset}(\textsc{Hash}(n, x))\,] = O(\gamma(n))\,.$$

Then $\lim_{n\to\infty} \Pr[\, \mathsf{fp}(\mathcal{S}_n, z) = 1\,] = \left(1 - e^{-\lambda/k}\right)^k$.

Kirsch and Mitzenmacher prove that the *double hashing scheme* defined by

$$\textsc{Hash}(n, x)_j = 1 + (h_1(n, x) + j \cdot h_2(n, x) \mod m(n))\,,$$

for each $j \in [k]$, where $k(n) = k$ and $m(n) = cn$ for some constants k and c, and $h_1(n, \cdot)$ and $h_2(n, \cdot)$ are independent and identically distributed random functions with range $[m(n)]$, satisfies the conditions of Theorem B.1 for $\lambda = k^2/c$ and $\gamma(n) = 1/n^2$. (See [**?** , Thorem 5.2].) Thus, the false-positive probability converges to $(1-e^{-k/c})^k$ as $n\to\infty$. But how close to this limit is the false positive probability for some fixed n? To address this question, Kirsch and Mitzenmacher provide an analysis of the rate of convergence.

Theorem B.2 (Theorem 6.1 of [**?**]). *Suppose Γ satisfies the conditions of Theorem B.1 for λ, k, and $\gamma(n)$. For every $n \in \mathcal{N}$, it holds that*

$$\left|\Pr[\, \mathsf{fp}(\mathcal{S}_n, z) = 1\,] - \left(1 - e^{-\lambda/k}\right)^k\right| = O(n\gamma(n) + 1/n)\,.$$

For the double hashing scheme in particular, we have that the false positive probability is at most $(1 - e^{-k/c})^k + O(1/n)$ for any $n \in \mathcal{N}$. (See [**?** , Theorem 6.2].)

Fix integers $k, m, n, \lambda \geq 0$ and let $H\colon \{0,1\}^* \to [m]^k$ and $F\colon \{0,1\}^\lambda \times \{0,1\}^* \to [m]^k$ be functions. Let $\Pi_{\mathsf{sbf}} = \mathrm{SBF}[2\textsc{Hash}[H], k, m, n, \lambda]$ and $\Pi_{\mathsf{kbf}} = \mathrm{KBF}[2\textsc{Hash}[F], k, m, n, \lambda]$ as defined in Figure **??**. If H and F are random functions, then $2\textsc{Hash}[H]$ and $2\textsc{Hash}[F]$ are both realizations of the double hashing scheme for a particular choice of n. From Thoerem B.2 it follows that the false positive probability for Π_{sbf} and Π_{kbf} is at most $(1 - e^{-kn/m})^k + O(1/n)$. (See Figure 9 for a visualization of this bound.) In the proof of security for Π_{sbf} (Theorem **??**), we model H as a random oracle; for Π_{kbf} (Theorem **??**), we can treat F as a random function assuming it is a good PRF.

C PROOFS