DATA STRUCTURES IN ADVERSARIAL ENVIRONMENTS

By

SAM A. MARKELON

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2025

© 2025 Sam A. Markelon

For my dearest Julia, without whom this would mean less

ACKNOWLEDGEMENTS

- Family – Dad, Mom, Jack, Hannah, Nonni and Boppi, Kelly and Brian; plus extended – mention upbringing in Burlington, CT
- UConn mentors – Joo, Krawec, Fuller, Herzberg – others?
- Tom
- Committee – VB, SR, PT, JB
- FICS lab – KB, students past and present
- ETHZ Applied Cryptography group: Kenny, Mia, Nico – more by name?
- TUD Cryptoplexity Group – Marc and Moritz
- Friends – Gainesville (Tim, JT, Jefferson, Logan, Gavin, Trivia Group), CT (all in Rod Stewart Chat), all those I forgot to name.
- Julia

TABLE OF CONTENTS

page

LIST OF TABLES

Tables page

LIST OF FIGURES

Figures page

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DATA STRUCTURES IN ADVERSARIAL ENVIRONMENTS

By

Sam A. Markelon

August 2025

Chair: Vincent Bindschaedler
Major: Computer Science

This is the abstract tex file, which should have been set in the main file using the command

`\setAbstractFile{Drive:/file/location/abstractFile}`.

This is what will appear in the place of an abstract, no formatting or other content is needed, just fill this file with your actual abstract, eg; In this paper we give examples of the various files and configurations used in the graduate school LaTeX template for dissertations and thesis papers. It should be 350 words or less.

CHAPTER 1
INTRODUCTION

The following needs to be re-organized and re-written.

Data structures define representations of possibly dynamic (multi)sets, along with the operations that can be performed on this representation of the underlying data. Efficient data structures are crucial for designing efficient algorithms [1]. The development and analysis of data structures has largely been driven by operational concerns, e.g., efficiency, ease of deployment, support for broad application. Security concerns, on the other hand, have traditionally been afterthoughts (at best). However, recent research has highlighted that many widely-used data structures do not behave as expected when in the presence of adversaries that have the ability to control the data they represent. Further, complex protocols that have sophisticated security goals are increasingly using a variety of bespoke data structures as fundamental components of their design. Therefore, it is wise to begin applying the provable security paradigm to data structures themselves.

For instance, consider probabilistic data structures (PDS). They provide compact (sublinear) representations of potentially large collections of data and support a small set of queries that can be answered efficiently. Prime examples of such structures include the Bloom filter [2], the HyperLogLog [3], and the Count-min Sketch [4]. These space and (by extension) performance gains come at the expense of correctness. Specifically, PDS query responses are computed over the compact representation of the data, as opposed to the complete data. As a result, PDS query responses are only guaranteed to be *close* to the true answer with *large* probability, where *close* and *large* are typically functions of structure parameters (e.g., the representation size) and properties of the data. These guarantees are stated under the assumption that the data and the internal randomness of the PDS are independent. Informally, this is tantamount to assuming that the entire collection of data is (or can be) determined *before* any random choices are made by the PDS. For many PDS, this means before some number of hash functions are sampled, as the PDS operates deterministically after that. Recent works have begun to explore the impact on

correctness guarantees for data that *may* depend upon the internal randomness of the structure, and the initial findings are negative.

Moreover, consider the class of data structures we refer to as *skipping data structures*. Unlike the probabilistic data structures we discussed earlier, this class of structure are not space-efficient (compact) and, in turn, give exact answers to queries. These data structures (e.g., hash tables, skip lists, and treaps) offer fast average-case runtime of their operations, but have worst-case runtime that is poor. They achieve this by using some form of randomness to determine the representation of the underlying data collection. Recent research shows that adaptive adversaries are able to force worst-case runtime for these structure, often demonstrated by attacks on real-world systems. Therefore, instead of focusing on adversarial correctness as in the PDS section, we focus on preserving the expected run time of these structures with large probability in the presence of an adversary.

CHAPTER 2
BACKGROUND

The following needs to be re-organized and re-written.

2.1 Notation

Bitstring and Set Operations.

Let $\{0,1\}^*$ denote the set of bitstrings and let ε denote the empty string. Let $X \| Y$ denote the concatenation of bitstrings X and Y. When $\mathcal{S}$ is an abstract data-object (e.g., a (multi)set, a list) and e is an object that can be appended (in some understood fashion) to $\mathcal{S}$, we overload the $\|$ operator and write $\mathcal{S} \| e$.

Let $x \twoheadleftarrow \mathcal{X}$ denote sampling x from a set $\mathcal{X}$ according to the distribution associated with $\mathcal{X}$; if $\mathcal{X}$ is finite and the distribution is unspecified, then it is uniform. Let $[i..j]$ denote the set of integers $\{i, \ldots, j\}$; if $i > j$, then define $[i..j] = \emptyset$. For all $m \geq 2$, let $[m] = \{1, 2, \ldots, m\}$.

Let $\mathcal{A}$ and $\mathcal{B}$ be sets. We take $\mathcal{A} \cup \mathcal{B}$ to be the union of the sets, $\mathcal{A} \cap \mathcal{B}$ to be the intersection of the sets, and $\mathcal{A} \setminus \mathcal{B}$ to be set-theoretic difference of $\mathcal{A}$ and $\mathcal{B}$.

Functions.

Let $\mathrm{Func}(\mathcal{X}, \mathcal{Y})$ denote the set of functions $f : \mathcal{X} \to \mathcal{Y}$. For every function $f : \mathcal{X} \to \mathcal{Y}$, define $\mathrm{ID}^f : \{\varepsilon\} \times \mathcal{X} \to \mathcal{Y}$ so that $\mathrm{ID}^f(\varepsilon, x) = f(x)$ for all x in the domain of f. This allows us to use unkeyed hash functions H in situations where, syntactically, a function is required to take a key along with its input.

Arrays and Tuples.

We use the distinguished symbol $\star$ to mean that a variable is uninitialized. By $[\mathrm{item}] \times \ell$ for $\ell \in N$ we mean a vector of ℓ replicas of item. We use $\mathrm{zeros}(m)$ denote a function that returns an m-length array of 0s and, likewise, $\mathrm{zeros}(k, m)$ to denote a function that returns an $k \times m$ array of

0s. We index into arrays (and tuples) using $[\cdot]$ notation; in particular, if R is a function returning a k-tuple, we write $R(x)[i]$ to mean the i-th element/coordinate of $R(x)$. If $X = (x_1, x_2, \ldots, x_t)$ is a tuple and $\mathcal{S}$ is a set, we overload standard set operators (e.g., $X \subseteq \mathcal{S}$) treating the tuple as a set; if we write $X \setminus \mathcal{S}$, we mean to remove all instances of the elements of $\mathcal{S}$ from the tuple X, returning a tuple X' that is "collapsed" by removing any now-empty positions.

2.2 A Syntax for Data Structures

We present (a slightly modified) syntax for data structures first provided by [5]. While originally used to describe a variety of probabilistic data structures, the syntax is appropriately general. A syntactic formalization of data structures in this way not only allows us to elegantly describe numerous data structures, but also craft security definitions that are directly related to the operations the data structure allows. We will do exactly this in our case studies throughout the rest of this work.

We start by fixing three 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 $\Pi = (\textsc{Rep}, \textsc{Qry}, \textsc{Up})$, where:

- $\textsc{Rep}\colon \mathcal{K} \times \mathcal{D} \to \{0,1\}^* \cup \{\perp\}$ is a (possibly) randomized *representation algorithm*, taking as input a key $K \in \mathcal{K}$ and data object $\mathcal{S} \in \mathcal{D}$, and outputting the representation $\mathsf{repr} \in \{0,1\}^*$ of D, or $\perp$ in the case of a failure. We write this as $\mathsf{repr} \leftarrow \textsc{Rep}_K(\mathcal{S})$.

- $\textsc{Qry}\colon \mathcal{K} \times \{0,1\}^* \times \mathcal{Q} \to \mathcal{R} \cup \{\perp\}$ is a deterministic *query-evaluation algorithm*, taking as input $K \in \mathcal{K}$, $\mathsf{repr} \in \{0,1\}^*$, and $\mathsf{qry} \in \mathcal{Q}$, and outputting an answer $a \in \mathcal{R}$, or $\perp$ in the case of a failure. We write this as $a \leftarrow \textsc{Qry}_K(\mathsf{repr}, \mathsf{qry})$.

- $\textsc{Up}\colon \mathcal{K} \times \{0,1\}^* \times \mathcal{U} \to \{0,1\}^* \cup \{\perp\}$ is a (possibly) randomized *update algorithm*, taking as input $K \in \mathcal{K}$, $\mathsf{repr} \in \{0,1\}^*$, and $\mathsf{up} \in \mathcal{U}$, and outputting an updated representation repr', or $\perp$ in the case of a failure. We write this as $\mathsf{repr}' \leftarrow \textsc{Up}_K(\mathsf{repr}, \mathsf{up})$.

Allowing each of the algorithms to take a key K permits one to separate (for some security notion) any secret randomness used across data structure operations, from per-operation randomness (e.g., generation of a salt). Note that this syntax admits the common case of *unkeyed* data structures, by setting $\mathcal{K} = \{\varepsilon\}$. Moreover, we can set $\mathcal{K} = \mathsf{priv}$ to be a private key and allow the corresponding public key pub to be a public parameter in the case the data structure relies on asymmetric cryptographic primitives.

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.

Note that Up takes a function operating on data objects as an argument, even though Up itself operates on *representations* of data objects. This is intentional, to match the way these data structures generally operate. In a data structure representing a set or multiset, we often think of performing operations such as 'insert x' or 'delete y'. When the set or multiset is not being stored, but instead modeled via a representation, the representation must transform these operations into operations on the actual data structure it is using for storage. This is common for operation on probabilistic data structures.

We also note that the query algorithm Qry is deterministic, which reflects the overwhelming majority of data structures in practice. Allowing Qry to be randomized would allow for a greater degree of syntactic expressiveness, particularly for some data structures that provide privacy guarantees. However, it can make it more difficult to craft correctness properties in that it may be difficult to discern the errors caused by an adaptive adversary versus "intended" error arising from the randomized query algorithm. Care must be taken when both designing structures and defining security properties to ensure issues do not arise from this.

2.3 Streaming Data

A *stream* data-object $\vec{S} = e_1, e_2, \ldots$ is a finite sequence of elements $e_i \in \mathcal{U}$ for some universe $\mathcal{U}$. The elements of a stream are not necessarily distinct, and the (stream) frequency of some $x \in \mathcal{U}$ is $|\{i : e_i = x\}|$. From the perspective of the PDS, the stream is presented one element at a time, with no buffering or "look ahead". That is, processing of a stream is performed in order, and the processing of e_i is completed before the processing of e_{i+1} may begin; once e_i has been processed, it cannot be revisited.

2.4 Redis

Redis (Remote Dictionary Server) is a general purpose, in-memory database that supports a rich array of functionality, including various Probabilistic Data Structures (PDS), such as Bloom filters, Cuckoo filters, as well as cardinality and frequency estimators. These PDS typically perform well in the average case. However, given that Redis is intended to be used across a diverse array of applications, it is crucial to evaluate how these PDS perform under worst-case scenarios, i.e., when faced with adversarial inputs. We offer a comprehensive analysis to address this question. We begin by carefully documenting the different PDS implementations in Redis, explaining how they deviate from those PDS as described in the literature. Then we show that these deviations enable a total of 10 novel attacks that are more severe than the corresponding attacks for generic versions of the PDS. We highlight the critical role of Redis' decision to use non-cryptographic hash functions in the severity of these attacks. We conclude by discussing countermeasures to the attacks, or explaining why, in some cases, countermeasures are not possible.

CHAPTER 3
COMPACT FREQUENCY ESTIMATORS IN ADVERSARIAL ENVIRONMENTS

Count-Min Sketch (CMS) and HeavyKeeper (HK) are two realizations of a compact frequency estimator (CFE). These are a class of probabilistic data structures that maintain a compact summary of (typically) high-volume streaming data, and provides approximately correct estimates of the number of times any particular element has appeared. CFEs are often the base structure in systems looking for the highest-frequency elements (i.e., top-K elements, heavy hitters, elephant flows). Traditionally, probabilistic guarantees on the accuracy of frequency estimates are proved under the implicit assumption that stream elements do not depend upon the internal randomness of the structure. Said another way, they are proved in the presence of data streams that are created by non-adaptive adversaries. Yet in many practical use-cases, this assumption is not well-matched with reality; especially, in applications where malicious actors are incentivized to manipulate the data stream. We show that the CMS and HK structures can be forced to make significant estimation errors, by concrete attacks that exploit adaptivity. We analyze these attacks analytically and experimentally, with tight agreement between the two. Sadly, these negative results seem unavoidable for (at least) sketch-based CFEs with parameters that are reasonable in practice. On the positive side, we give a new CFE (Count-Keeper) that can be seen as a composition of the CMS and HK structures. Count-Keeper estimates are typically more accurate (by at least a factor of two) than CMS for "honest" streams; our attacks against CMS and HK are less effective (and more resource intensive) when used against Count-Keeper; and Count-Keeper has a native ability to flag estimates that are suspicious, which neither CMS or HK (or any other CFE, to our knowledge) admits.

The use of probabilistic data structures (PDS) has grown rapidly in recent years in correlation with the rise of distributed applications producing and processing huge amounts of data. Probabilistic data structures provide compact representations of (potentially massive) data, and support a small set of queries. The trade-off for compactness is that query responses are only guaranteed to be

"close" to the true answer (i.e., if the query were evaluated on the full data) with a certain probability. For example, the ubiquitous Bloom filter [2] admits data-membership queries (*Does element x appear in the data?*). Bloom filters are used in applications such as increasing cache performance [6], augmenting the performance of database queries [7], indexing search results [8], and Bitcoin wallet synchronization [9]. The probabilistic guarantee on the correctness of responses assumes that the data represented by the Bloom filter is independent of the randomness used to sample the hash functions that are used to populate the filter, and to compute query responses. This is equivalent to providing correctness guarantees in the presence of adversarial data sets and queries that are *non-adaptive*, i.e., made in advance of the sampling of the hash functions. A number of recent works — notably those of Naor and Yogev [10], Clayton, Patton and Shrimpton [5], and Filić et al. [11] — have provided detailed analyses of Bloom filters under *adaptive* attacks; the results are overwhelmingly negative. Paterson and Raynal [12] provided similar results for the HyperLogLog PDS, which can be used to count the number of distinct elements in a data collection [3].

In this work, we focus on PDS that can be used to estimate the number of times any particular element x appears in a collection of data, i.e., the *frequency* of x. Such compact frequency estimators (CFEs) are commonly used in streaming settings, to identify elements with the largest frequencies — so-called *heavy hitters* or *elephants*. Finding extreme elements is important for network planning [13], network monitoring [14], recommendation systems [15], and approximate database queries [16], to name a few applications.

The Count-min Sketch (CMS) [4] and HeavyKeeper (HK) [17] structures are two CFEs that we consider, in detail. The CMS structure has been widely applied to a number of problems outlined above. Details on these applications are thoroughly examined in the survey paper by Sigurleifsson et al. [18]. The HK structure is the CFE of choice in the RedisBloom module [16], a component of the Redis database system [19].

Of particular interest to us is the 2019 ACM SIGSAC work of Clayton, Patton, and Shrimpton [5] that both furthers the adversarial analysis on Bloom filters and also presents a general model for analyzing probabilistic data structures for provable security. This paper gives a first look at the security of the Count-min sketch in adversarial environments. However, in this paper a very conservative security model for the CMS was used, which counted any overestimation of a particular element as an adversarial gain, rather than tying the security to the non-adaptive guarantees of the structure. Further, a thresholding mechanism is used to achieve security for the CMS, a solution which we deem untenable for real world uses of the CMS.

As is the case for Bloom filters, HyperLogLog and other PDS, the accuracy guarantees for CFEs effectively assume that the data they represent were produced by a non-adaptive strategy. Our work explores the accuracy of CMS and HK estimates when the data is produced by *adaptive* adversarial strategies (i.e., adaptive attacks). We give explicit attacks that aim to make as-large-as-possible gaps between the estimated and true frequencies of data elements. We give concrete, not asymptotic, expressions for these gaps, in terms of specific adversarial resources (i.e., oracle queries), and support these expressions with experimental results. And our attacks fit within a well-defined "provable security"-style attack model that captures four adversarial access settings: whether the CFE representations are publicly exposed (at all times) or hidden from the adversary, and whether the internal hash functions are public (i.e., computable offline) or private (i.e. visible only, if at all, by online interaction with the structure).

In this work we draw explicit attention to the fact that probabilistic data structures, and in particular frequency estimators, were not designed with security in mind by presenting attacks that degrade the correctness of the query responses these structures provide.

Our findings are negative in all cases. No matter the combination of public and private, a well resourced adversary can force CMS and HK estimates to be arbitrarily far from the true frequency. As one example of what this means for larger systems, things that have never appeared in the stream can be made to look like heavy hitters (in the case of CMS), and legitimate heavy hitters

can be made to disappear entirely (in the case of HK). This is somewhat surprising in the "private-private" setting, where the attack can only gain information about the structure and its operations via frequency estimate queries. Of course, there are differences in practice: when attacks are forced to be online, they are easier to detect and throttle, so the query-resource terms in our analytical results are likely capped at smaller values than when some or all of an attack can progress offline.

Our attacks exploit structural commonalities of CMS and HK. At their core, each of these processes incoming data elements by mapping them to multiple positions in an array of counters, and these are updated according to simple, structure-specific rules. Similarly, when frequency estimation (or *point*) queries are made, the queried element is mapped to its associated positions, and the response is computed as a simple function of values they hold. So, our attacks concern themselves with finding *cover sets*: given a target x, find a small set of data elements (not including x) that collectively hash to all of the positions associated with x. Intuitively, inserting a cover set for x into the stream will give the structure incorrect information about x's relationship to the stream, causing it to over- or underestimate its frequency.

The existence of a cover set in the represented data is necessary for producing frequency estimation errors in HK, and both necessary and sufficient in CMS. Sadly, our findings suggest that preventing an adaptive adversary from finding such a set seems futile, no matter what target element is selected. The task can be made harder by increasing the structural parameters, but this quickly leads to structures whose size makes them unattractive in practice, i.e., *linear* in the length of the stream.

Motivating a more robust CFE. Say that the array M in CMS has k rows and m counters (columns) per row. The CMS estimate for x is $\hat{n}_x = \min_{i \in [k]}\{M[i][p_i]\}$, where p_i is the position in row i to which x hashes. In the insertion-only stream model it must be that $\hat{n}_x \geq n_x$, where n_x is the true frequency of x. To see this, given an input stream $\vec{S}$, let
$V_x^i = \{y \in \vec{S} \mid y \neq x \text{ and } h_i(y) = p_i\}$ be the set of elements that hash to the same counter as x, in

the i-th row. Then we can write $M[i][p_i] = n_x + \sum_{y \in V_x^i} n_y$, where the $n_y > 0$ are the true frequencies of the colliding ys. Viewed this way, we see that the CMS estimate $\hat{n}_x$ minimizes the impact of "collision noise", i.e., $\hat{n}_x = n_x + \min_{i \in [k]} \{\sum_{y \in V_x^i} n_y\}$.

We could improve this estimate if we knew some extra information about the value of the sum, or the elements that contribute to it. Let's say that, with a reasonable amount of extra space, we could compute $C_i = \epsilon_i \left(\sum_{y \in V_x^i} n_y\right)$ for some $\epsilon_i \in [0, 1]$ that is bounded away from zero. Then we would improve the estimate to $\hat{n}_x = n_x + \min_{i \in [k]} \left\{(1 - \epsilon_i) \left(\sum_{y \in V_x^i} n_y\right)\right\}$. How might we do this? Consider the case that for some row $i \in [k]$ there is an element $y^* \in V_x^i$ that dominates the collision noise, e.g. $n_{y^*} = (1/2) \sum_{y \in V_x^i} n_y$. Then even the ability to accurately estimate n_{y^*} would give a significant improvement in accuracy of $\hat{n}_x$, by setting C_i to this estimate. It turns out that HK provides something like this. It maintains a $k \times m$ matrix A, where $A[i][j]$ holds a pair (fp, cnt). In the first position is a *fingerprint* of the current "owner" of this position, and, informally, cnt is the number of times that $A[i][j]$ "remembers" seeing the current owner. (Ownership can change over time, as we describe in the body.) If we use the same hash functions to map element x into the same-sized M and A, then there is possibility of using the information at $A[i][p_i]$ to reduce the additive error (w.r.t. n_x) in the value of $M[i][p_i]$. This observation forms the kernel of our new Count-Keeper structure.

The Count-Keeper CFE.

We propose a new structure that, roughly speaking, combines equally sized (still compact) CMS and HK structures, and provide analytical and empirical evidence that it reduces the error (by at least a factor of two) that can be induced once a cover set is found. It also requires a type of cover set that is roughly twice as expensive (in terms of oracle queries) to find. Moreover, it can effectively detect when the reported frequency of an element is likely to have large error. In this way we can dampen the effect of the attacks, by catching and raising a *flag* when a cover set has been found and is inserted many times to induce a large frequency error estimation on a particular element.

Intuitively, our Count-Keeper (CK) structure has improved robustness against adaptive attacks because CMS can only overestimate the frequency of an element, and HK can only underestimate the frequency (under a certain, practically reasonable assumption). We experimentally demonstrate that CK is robust against a number of attacks we give against the other structures. Moreover, it performs comparably well if not better than the other structures we consider in frequency estimation tasks in the non-adversarial setting.

As a side note, we uncovered numerous analytical errors in [17] that invalidate some of their claims about the behaviors of the HK structure. We have communicated with the authors of [17] and contacted Redis, whose RedisBloom library implements HK (and CMS) with fixed, public hash functions (i.e., the internal randomness is fixed for all time and visible to attackers).

In [20], the authors consider adding robustness to streaming algorithms using differential privacy. Meanwhile, Hardt and Woodruff [21], Cohen et al. [22] and Ben-Eliezer et al.[23] have shown that linear sketches (including CMS but not HK) are not "robust" to well-resourced adaptive attacks, when it comes to various L_p-norm estimation tasks, e.g., solving the k-heavy-hitters problem relative to the L_2-norm. These works are mostly of theoretical importance, whereas we aim to give concrete attacks and results that are (more) approachable for practitioners.

3.1 Formal Attack Model

To enable precise reasoning about the correctness of frequency estimators when data streams may depend, in arbitrary ways, on the internal randomness of the data structure, we give a pseudocode description of our attack model in Figure 3-1. The experiment parameters u, v determine whether the adversary $\mathcal{A}$ is given K and repr, respectively. Thus, there are actually four attack models encoded into the experiment.

The adversary is provided a target $x \in \mathcal{U}$, and given access to oracles that allow it to update the current representation (**Up**) — in effect, to control the data stream — and to make any of the queries permitted by the structure (**Qry**). We abuse notation for brevity and write $\mathbf{Up}(e)$ to mean

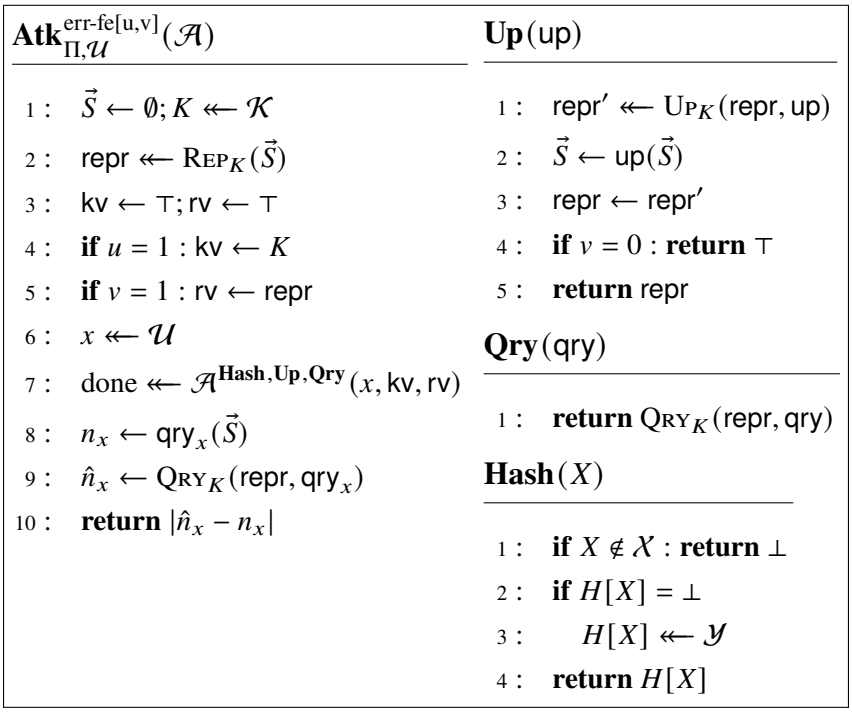


Figure 3-1. The ERR-FE (ERRor in Frequency Estimation) attack model. When experiment parameter $v = 1$ (resp. $v = 0$) then the representation is public (resp. private); when $u = 1$ (resp. $u = 0$) then the structure key K is rendered public (resp. private). The experiment returns the absolute difference between the true frequency n_x of an adversarially chosen $x \in \mathcal{U}$, and the estimated frequency $\hat{n}_x$. The **Hash** oracle computes a random mapping $\mathcal{X} \to \mathcal{Y}$ (i.e., a random oracle), and is implicitly provided to Rep, Up and Qry.

an insertion of e into the structure and $\mathbf{Qry}(e)$ to get a point query on e for some element $e \in \mathcal{U}$. Note that when $v = 0$, the **Up**-oracle leaks nothing about updated representation, so that it remains "private" throughout the experiment. The adversary (and, implicitly, Rep, Up, Qry) is provided oracle access to a random oracle $\mathbf{Hash} : \mathcal{X} \to \mathcal{Y}$, for some structure-dependent sets $\mathcal{X}, \mathcal{Y}$. The output of the experiment is the absolute error between the true frequency n_x of x in the adversarial data stream, and the structure's estimate $\hat{n}_x$ of n_x.

Remark. Conventionally, one would define an "advantage" function over the security experiment, and there are various interesting ways this could be done. As examples, one could parameterize by a threshold function $T : \mathbb{Z} \to \mathbb{Z}$, and have the advantage measure the probability that the value $|\hat{n}_x - n_x| > T(q_U)$; or, one could compare this value to known non-adaptive error guarantees. As we will not be proving the security of any structures, we use $\mathbf{Atk}_{\Pi}^{\text{err-fe[u,v]}}(\cdot)$ as a precise description of the attack setting. We will explore *lower* bounds on the values returned by the experiment, for explicit attacks that we give.

We capture various settings related to the view of the adversary in our attack interface. We have a setting in which the data structure representation is kept private from the adversary, and we also have a setting in which the specific choice of hash functions selected by a particular representation are kept private from the adversary. These settings can be examined together, separately, or both can be disregarded and the adversary can be given a "full view". That is we consider when the both the representation and hash functions are private, when the representation is public and the hash functions are private, when the representation is private and the hash functions are public, and when both the representation and hash functions are public.

In practice the private representation setting occurs due to suppression of information leaked by the oracles. In particular in this setting, the **Rep** and **Up** oracles return nothing, thus leaking nothing about the underlying data representation. Further, we make hash functions "private" by keying them with a (non-empty) randomly generated secret key.

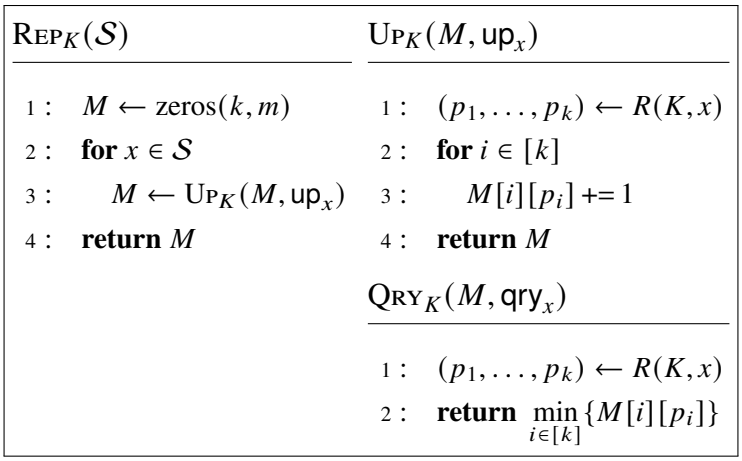


Figure 3-2. Keyed count-min sketch structure $\text{CMS}[R, m, k]$ admitting point queries for any $x \in \mathcal{U}$. The parameters are integers $m, k \geq 0$, and a keyed function $R : \mathcal{K} \times \mathcal{U} \to [m]^k$ that maps data-object elements (encoded as strings) to a vector of positions in the array $\boldsymbol{M}$. A concrete scheme is given by a particular choice of parameters.

3.2 Count-min Sketch

Figure 3-2 gives a pseudocode description of the count-min sketch (CMS), in our syntax. An instance of CMS consists of a $k \times m$ matrix M of (initially zero) counters, and a mapping R between the universe $\mathcal{U}$ of elements and $[m]^k$. An element x is added to the CMS representation by computing $R(K, x) = (p_1, p_2, \ldots, p_k)$, and then adding 1 to each of the counters at $M[i][p_i]$. Traditionally, it is assumed that $(p_1, \ldots, p_k) = (h_1(x), \ldots, h_k(x))$ where the h_i are sampled at initialization from some family H of hash functions, but we generalize here to make the exposition cleaner, and to allow for the mapping to depend upon secret randomness (i.e., a key K).

The point query $\text{QRY}(\text{qry}_x)$ returns $\hat{n}_x = \min_{i \in [k]} \{M[i][p_i]\}$. We note that (in the insertion-only model) it must be that $\hat{n}_x \geq n_x$. To see this, let $V_x^i = \{y \in \vec{S} \mid y \neq x \text{ and } R(y)[i] = p_i\}$ be the set of elements that "collide" with x's counter in the i-th row. Then we can write $M[i][p_i] = n_x + \sum_{y \in V_x^i} n_y$, where $n_y \geq 0$. Viewed this way, we see that a CMS estimate $\hat{n}_x$ minimizes the "collision noise", i.e., $\hat{n}_x = n_x + \min_{i \in [k]} \{\sum_{y \in V_x^i} n_y\}$.

For any $\epsilon, \delta \geq 0$, any $x \in \mathcal{U}$, and any stream $\vec{S}$ (over $\mathcal{U}$) of length N, it is guaranteed that $\Pr[\, \hat{n}_x - n_x > \epsilon N \,] \leq \delta$ when: (1) $k = \lceil \ln \frac{1}{\delta} \rceil$, $m = \lceil \frac{e}{\epsilon} \rceil$, and (2) $R(K, x) = (h_1(K \,\|\, x), h_2(K \,\|\, x), \ldots, h_k(K \,\|\, x))$ for h_i that are uniformly sampled from a

pairwise-independent hash family H [4]. Implicitly, there is a third requirement, namely (3) the stream and the target x are independent of the internal randomness of the structure (i.e., the coins used to sample the h_i). This is equivalent to saying that the stream $\vec{S}$ and the target x are determined before the random choices of the structure are made.

3.3 HeavyKeeper

$\textsc{Rep}_K(\mathcal{S})$

```
1 : / initialise k × m (fp,cnt) 2-d array
2 : for i ∈ [k]
3 :     A[i] ← [(⋆, 0)] × m
4 : for x ∈ S
5 :     A ← Up_K(A, up_x)
6 : return A
```

$\textsc{Qry}_K(A, \mathsf{qry}_x)$

```
1 : (p_1, ..., p_k) ← R(K, x)
2 : fp_x ← T(K, x)
3 : cnt_x ← 0
4 : for i ∈ [k]
5 :     if A[i][p_i].fp = fp_x
6 :         cnt ← A[i][p_i].cnt
7 :         cnt_x ← max {cnt_x, cnt}
8 : return cnt_x
```

$\textsc{Up}_K(A, \mathsf{up}_x)$

```
1 : (p_1, ..., p_k) ← R(K, x)
2 : fp_x ← T(K, x)
3 : for i ∈ [k]
4 :     if A[i][p_i].fp ∉ {fp_x, ⋆}
5 :         r ↞ [0, 1)
6 :         if r ≤ d^(A[i][p_i].cnt)
7 :             A[i][p_i].cnt −= 1
8 :     / overtake the counter if 0
9 :     if A[i][p_i].cnt = 0
10 :        A[i][p_i].fp ← fp_x
11 :    / increase the count if fp = fp_x
12 :    if A[i][p_i].fp = fp_x
13 :        A[i][p_i].cnt += 1
14 : return A
```

Figure 3-3. Keyed structure $\mathrm{HK}[R, T, m, k, d]$ supporting point-queries for any potential stream element $x \in \mathcal{U}$ (qry_x). The parameters are a function $R : \mathcal{K} \times \mathcal{U} \to [m]^k$, a function $T : \mathcal{K} \times \mathcal{U} \to \{0,1\}^n$ for some desired fingerprint length n, decay probability $0 < d \leq 1$, and integers $m, k \geq 0$.

Like CMS, an instance of the HeavyKeeper data structure is parameterized by positive integers k, m, and a function $R\colon \mathcal{K} \times \mathcal{U} \to [m]^k$; in addition, it is parameterized by real-valued $d \in (0, 1]$, and *fingerprinting function* $T\colon \mathcal{K} \times \mathcal{U} \to \{0,1\}^n$ for some fixed $n > 0$. The HK structure (see the pseudocode in Figure 3-3) maintains a $k \times m$ matrix A. However, each $A[i][j]$ holds a pair $(\mathrm{fp}, \mathrm{cnt})$, initialized as $(\star, 0)$ where $\star$ is a distinguished symbol. Informally, for a given stream $\vec{S}$, any $z \in \vec{S}$ such that $A[i][j].\mathrm{fp} = T(K, z)$ is an *owner* of this position; there may be more than one such owner at a time, if $T(K, \cdot)$ admits many collisions. Ownership can change as a stream is

processed: if some y arrives whose fingerprint is different than that of the current owner(s), then the current (positive) value c of $A[i][j]$.cnt is decremented with probability d^{-c}. Loosely, decrementing c is akin to $A[i][j]$ "forgetting" a prior arrival of its current owner(s); with this viewpoint, the value of $A[i][j]$.cnt is the number of times that this position "remembers" seeing its current owner(s). If y causes that number to become zero, then it becomes an owner: the stored fingerprint is changed to $\text{fp}_y = T(K, y)$, and the counter is set to 1. Note that for CMS, $M[i][j]$ "remembers" the total number of elements that it observed, but nothing about *which* elements. This observation will motivate our Count-Keeper structure, later on.

The HK provides frequency estimates via point-queries. Writing $(p_1, \ldots, p_k) \leftarrow R(K, x)$ and $\text{fp}_x \leftarrow T(K, x)$, a point-query for x returns $\max\left\{A[i][p_i].\text{cnt} \mid A[i][p_i].\text{fp} = \text{fp}_x, i \in [k]\right\}$, i.e., the largest counter value among those positions in A that "remember" having seen x. If that set is empty, the point-query returns 0.

Yang et al.[17] do state a probabilistic guarantee on the size of estimation errors, under an assumption that each $A[i][j]$ has one and only one owner for the duration of the stream, but the statement is insufficiently precise and its proof is flawed, so we will not quote it. In the full version of our paper [24], we recover a meaningful result (under their assumptions).

3.4 Attacks on CMS and HK

In the following discussion of attacks against CMS and HK in our formal model, we will implement the mappings $R : \mathcal{U} \to [m]^k$ and $T : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^n$ via calls to the **Hash**-oracle. In detail, given some unambiguous encoding function $\langle \cdot, \cdot, \cdot \rangle$, for CMS we set $R(K, x) = (\textbf{Hash}(\langle 1, K, x\rangle), \textbf{Hash}(\langle 2, K, x\rangle, \ldots, \textbf{Hash}(\langle k, K, x\rangle)))$, and for HK, we set

$R(K, x)[i] = \textbf{Hash}(\langle \text{“cnt”}, i, K, x\rangle)$ and $T(K, x) = \textbf{Hash}(\langle \text{“fp”}, k+1, K, x\rangle)$. Note that the traditional analysis of CMS correctness assumes that the row-wise hash functions are sampled (uniformly) from a pairwise-independent family of functions, whereas our modeling treats the row-wise hash functions as k independent random functions from $\mathcal{U} \to [m]$. This makes the

adversary's task more difficult, as our attacks cannot leverage adaptivity to exploit structural characteristics of the hash functions. For the HK, the strings "cnt" and "fp" provide domain separation, and we implicitly assume that the outputs of calls to the **Hash**-oracle can be interpreted as random elements of $[m]^k$ when called with "cnt", and as random elements of the appropriate fingerprint-space,e.g., $\{0,1\}^n$ for some constant $n \geq 0$, when called with "fp".[1]

3.4.1 Cover Sets

Say $\hat{n}_x$ is the CMS estimate. As noted in Section 3.2, the estimate $\hat{n}_x = n_x + \min_{i\in[k]}\{\sum_{y\in V_x^i} n_y\}$; thus $\hat{n}_x = n_x$ if there exists an $i \in [k]$ such that $\sum_{y\in V_x^i} n_y = 0$. Since $n_y > 0$ for any $y \in V_x^i$, we can restate this as $\hat{n}_x > n_x$ if and only if $V_x^1, \ldots, V_x^k$ are all non-empty. When this is the case, the union $C = \bigcup_{i\in[k]} V_x^i$ contains a set of stream elements that "cover" the counters $M[i][p_i]$ associated to x. Since the presence of a covering C within the stream is necessary (and sufficient) for creating a frequency estimation error for the CMS, we formalize the idea of a "cover" in the following definition.

Definition 1. *Let $\mathcal{U}$ be the universe of possible stream elements. Fix $x \in \mathcal{U}$, $r \in \mathbb{Z}$, and $\mathcal{Y} \subseteq U$. Then a set $C = \{y_1, y_2, \ldots, y_t\}$ is an* $(\mathcal{Y}, x, r)$-cover *if: (1) $C \subseteq \mathcal{Y}\backslash\{x\}$, and (2)$\forall i \in [k]$ $\exists j_1, \ldots, j_r \in [t]$ such that $R(K,x)[i] = R(K,y_{j_1})[i], \ldots, R(K,x)[i] = R(K,y_{j_r})[i]$.* ♦

For the CMS, we will be interested in $\mathcal{Y} = \mathcal{U}$, $r = 1$, and we will shorten the notation to calling this a 1-cover (for x), or just a cover. For the HK, we will still be interested in $r = 1$, but with a different set $\mathcal{Y}$. In particular, HK has a fingerprint function $T(K,\cdot)$, and we define the set $\mathcal{FP}(K,x) = \{y \in \mathcal{U} \mid T(K,y) \neq T(K,x)\}$. We will typically write fp_x as shorthand for the result of computing $T(K,x)$, dropping explicit reference to the key K;

In analyzing their HK structure, Yang et al. [17], rely on there being "no fingerprint collisions", to ensure that HK have only one-sided error. (In general, the HK returned estimates may over- or underestimate the true frequency.) But, no precise definition of this term is given. We define it (by negation) as follows: stream $\vec{S}$ does not satisfy the *no-fingerprint collision* (NFC) condition with

[1] This separation could be more directly handled by augmenting the attack model with an additional hashing oracle, but for simplicity and ease of reading, we chose not to do so.

respect to x (and key K) if there exists $y, z \in \vec{S}||x$ such that $T(K, y) = T(K, z)$ *and* $\exists i$ such that $R(K, y)[i] = R(K, z)[i]$; otherwise $\vec{S}$ does satisfy the NFC condition with respect to x (and K). In other words, $\vec{S}||x$ cannot contain distinct elements that have the same fingerprint and share a counter position. Our analysis treats the fingerprint function $T(K, \cdot)$ and position hash functions $R(K, \cdot)[i]$ as random oracles, the particular value of K will not matter, only whether or not it is publicly known. As such, explicit mention of K can be elided without loss of generality, and we shorten $\mathcal{FP}(K, x)$ to $\mathcal{FP}_x$. Further, in the random oracle model the fingerprint computation and row position computation are independent, so the probability of their conjunction is much smaller than the simple "birthday bound" event on fingerprint collisions. Anyway, for our HK analysis (Section 3.4.3), we will be interested in $(\mathcal{FP}_x, x, 1)$-covers, which are just $(\mathcal{U}, x, 1)$-covers under NFC condition.

When analyzing our new CK structure (Section 3.5), which inherits the fingerprint function from HK, we will be interested in $(\mathcal{FP}_x, x, 2)$-covers, as $r = 1$ will no longer enable attacks to drive up estimation error.

Exploring time-to-cover. Observe that even when the stream elements and the target x are independent of the internal randomness of the structure, a sufficiently long stream will almost certainly contain a cover for x. For example, for CMS, this results in $\hat{n}_x$ being an overestimate of n_x. How long the stream needs to be for this to occur is what we explore next. Each of CMS, HK and CK use a mapping $R(K, \cdot)$ to determine the positions to which stream elements are mapped. Let L_i^r be the number of *distinct-element* evaluations of $R(K, .)$ needed to find elements covering the target's counter in the i^{th} row r times. Then L_i^r is a negative binomial random variable with success probability $p = \frac{1}{m}$ and $\Pr[L_i^r = z] = \binom{z-1}{z-r}(1-p)^{z-r}p^r$. This is because L_i^r counts the *minimal* number of evaluations needed to find r elements $y_1, \ldots, y_r$ with $R(K, y_j)[i] = p_i$. This holds for any $i \in [k]$, and all L_i^r are independent. Thus, letting $L^r = \max\{L_1^r, L_2^r, \ldots, L_k^r\}$, we have

$$\Pr[L^r \le z] = \prod_{i=1}^{k} \Pr[L_i^r \le z] = \left(p^r \sum_{t=0}^{z-r} \binom{t+r-1}{t} (1-p)^t \right)^k . \tag{3-1}$$

Note that relation (3-1) fully defines z for any fixed values of $\Pr[\, L^r \le z \,]$, m, k, r. Thus, we will be able to relate $\Pr\left[\, \mathsf{Cover}_x^r \,\right]$ and $\Pr[\, L^r = z \,]$ via the resources used in attacks, e.g., Cover_x^r occurs iff $L^r \le f_{m,k,r}(q_H, q_U, q_Q)$ for some function $f_{m,k,r}$ of the adversarial resources.

When $r = 1$, this simplifies to The L_i^1 are geometric random variables with success probability p, and

$$\Pr[L^1 \le z] = \left((1-q)(1+q+q^2+\cdots+q^{z-1}) \right)^k = (1-q^z)^k \tag{3-2}$$

with $q = 1 - p$. When $r = 2$ we arrive at a more complicated expression

$$\Pr[L^2 \le z] = (1 - zq^{z-1} + (z-1)q^z)^k. \tag{3-3}$$

One can show that $\mathbb{E}[L^1] = \sum_{z=0}^{\infty}(1-(1-q^z)^k)$; for typical values of m, we have the very good approximation $\mathbb{E}[L^1] \approx mH_k$, H_k being the k-th harmonic number. [2] This constant depends only on parameters m and k.

3.4.2 Cover-Set Attacks on CMS

In our attack model, if the mapping $R(K, \cdot)$ is public, we may use the **Hash** oracle (only) to find a cover set for the target x "locally", i.e., the step is entirely offline. When this is not the case, we use a combination of queries to the **Up** and **Qry** oracles to signal when a cover set *exists* among the current stream of insertions; then we make additional queries to learn a subset of stream elements that yield a cover.

[2] Concretely, when $k = 5, m = 1000$ we have $\mathbb{E}[L^1] \approx 2283$. Experimentally, we verified this result over $10,000$ trials with an average of 2281 insertions needed to find a cover set for a per-trial randomly chosen element x.

Before exploring each setting, we build up some general results. Let Cover_x^r be the event that in the execution of $\mathbf{Atk}_{\Pi}^{\text{err-fe[u,v]}}(\mathcal{A})$, the adversary queries the **Up**-oracle with $\mathsf{up}_{e_i}, \ldots, \mathsf{up}_{e_t}$ and $e_1, \ldots, e_t$ is an r-cover for the target. For concision, define random variable $\mathsf{Err} = \mathbf{Atk}_{\Pi}^{\text{err-fe[u,v]}}(\mathcal{A})$. We will mainly focus on $\mathbb{E}[\mathsf{Err}]$ when analyzing the behavior of structures, so here we observe that the non-negative nature of Err allows us to write $\mathbb{E}[\mathsf{Err}] = \sum_{\xi>1} \Pr[\,\mathsf{Err} \geq \xi\,]$. In determining the needed probabilities, it will be beneficial to condition on Cover_x^r, as this event (for particular values of r) will be crucial for creating errors.

Our attacks against CMS (and, later, HK and CK) have two logical stages. The first stage finds the necessary type of cover for the target x, and the second stage uses the cover to drive up the estimation error. The first stage is the most interesting, as the second will typically just insert the cover as many times as possible for a given resource budget (q_H, q_U, q_Q). We note that whether or not the first stage is adaptive depends on the public/private nature of the structure's representation and hash functions, whereas the second stage will always be adaptive.

Say **Up**-query budget (i.e., number of adversarial stream elements) is fixed to q_U, and for the moment assume that the other query budgets are infinite. Let some $q'_U \leq q_U$ of the **Up**-queries be used in the first stage of the attack. The number q'_U is a random variable, call it Q, with distribution determined by the randomness of the structure and coins of the attacker. So, $\mathbb{E}[\mathsf{Err}]$ may depend on the value of Q, and then we calculate the expectation as $\mathbb{E}[\mathbb{E}[\mathsf{Err} \mid Q]]$. After a cover C is found by the first stage (so Cover_x^1 holds), the second stage can insert C until the resource budget is exhausted. Note that each insertion of C will increase the CMS estimation-error by one. Our attacks ensure that $|C| \leq k$, and so the number of C-insertions in the second stage is at least $\left\lfloor \frac{q_U - Q}{k} \right\rfloor$. This implies that $\mathbb{E}[\mathsf{Err} \mid Q] \geq \sum_{\xi=1}^{\lfloor (q_U-Q)/k \rfloor} \Pr\left[\,\mathsf{Err} \geq \xi \mid Q, \mathsf{Cover}_x^1\,\right] \Pr\left[\,\mathsf{Cover}_x^1 \mid Q\,\right]$ Letting $0 \leq T \leq q_U$ be the maximum number of **Up**-queries allowed in the first stage (i.e. $Q \leq T$), we have

$$\mathbb{E}[\mathsf{Err}] \geq \sum_{q'_U=0}^{T} \left\lfloor \frac{q_U - q'_U}{k} \right\rfloor \Pr\left[\,\mathsf{Cover}_x^1 \mid Q{=}q'_U\,\right] \Pr\left[\,Q{=}q'_U\,\right].$$

Public hash and representation setting. The public hash setting allows to find a cover using the **Hash** oracle only (i.e., Q=0). This step introduces no error; $\mathbb{E}\left[\mathsf{Err}\right] = \left\lfloor \frac{q_U}{k} \right\rfloor \Pr\left[\mathsf{Cover}_x^1 \mid Q{=}0\right]$. Given our definition of L^1 as the minimal number of $R(K,\cdot)$ evaluations to find a cover, the cover-finding step of the attack requires $k(1{+}L^1)$ **Hash**-queries: k to evaluate $R(K,x)$, and then kL^1 to find a cover. Say q_H is the **Hash**-oracle budget for the attack. A cover is then found iff $L^1 \leq \frac{q_H-k}{k}$. Assuming $q_U > k$ (so that a found cover is inserted at least once) and using (3-2) we arrive at

$$\Pr\left[\mathsf{Cover}_x^1 \mid Q{=}0\right] = \left(1-(1-1/m)^{\frac{q_H}{k}-1}\right)^k \tag{3-4}$$

implying $\mathbb{E}[\mathsf{Err}] \geq \left\lfloor \frac{q_U}{k} \right\rfloor \left(1-(1-1/m)^{\frac{q_H}{k}-1}\right)^k$. For $q_H/k \gg 1$, which is likely as q_H is *offline* work and practical k are small, $\mathbb{E}[\mathsf{Err}] \approx q_U/k$. The full attack can be found in Figure 3-4.

$\mathrm{CoverAttack}^{\mathbf{Hash},\mathbf{Up},\mathbf{Qry}}(x,K,\mathsf{repr})$	$\mathrm{FindCover}^{\mathbf{Hash}}(r,x,K)$
1 : cover $\leftarrow$ FindCover$^{\mathbf{Hash}}(1,x,K)$	1 : cover $\leftarrow \emptyset$; found $\leftarrow$ False
2 : **until** q_U **Up**-queries made:	2 : $\mathcal{I} \leftarrow \emptyset$; tracker $\leftarrow$ zeros(k)
3 : **for** $e \in$ cover: **Up**(e)	3 : // $R(K,x)[i] = \mathbf{Hash}(\langle i,K,x\rangle)$
4 : **return** done	4 : $(p_1,p_2,\ldots,p_k) \leftarrow R(K,x)$
	5 : **while** not found
	6 : **if** q_H **Hash**-queries made
	7 : **return** $\emptyset$
	8 : $y \twoheadleftarrow \mathcal{U} \setminus (\mathcal{I} \cup \{x\})$
	9 : $\mathcal{I} \leftarrow \mathcal{I} \cup \{y\}$
	10 : $(q_1,q_2,\ldots,q_k) \leftarrow R(K,y)$
	11 : **for** $i \in [k]$
	12 : **if** $p_i = q_i$ **and** tracker$[i] < r$
	13 : cover $\leftarrow$ cover $\cup \{y\}$
	14 : tracker$[i]$ += 1
	15 : **if** sum(tracker) $= rk$
	16 : found $\leftarrow$ True
	17 : **return** cover

Figure 3-4. Cover Set Attack for the CMS in public hash function setting. We use $R(K,x)$ to mean $(\mathbf{Hash}(\langle 1,K,x\rangle), \mathbf{Hash}(\langle 2,K,x\rangle,\ldots,\mathbf{Hash}(\langle k,K,x\rangle)))$. The attack is parametrized with the update and **Hash** query budget q_U and q_H.

Private hash and private representation setting. This is the most challenging setting to find a cover: the privacy of hash functions effectively makes local hashing useless, and the private representation prevents the adversary from learning anything about the result of online hash computations.

CoverAttack$^{\mathbf{Up},\mathbf{Qry}}(x, \perp, \perp)$

```
cover ← FindCover^{Up,Qry}(x)
until q_U Up-queries made:
    for e ∈ cover: Up(e)
return done
```

MinUncover$^{\mathbf{Up},\mathbf{Qry}}(x, a', \text{cover})$

```
b' ← −1
while a' ≠ b'
    if (q_U − |cover| + 1)Up-
        or q_Q Qry-queries made:
            return cover
    b' ← a'
    for y ∈ cover : Up(y)
    a' ← Qry(x)
return a'
```

FindCover$^{\mathbf{Up},\mathbf{Qry}}(x)$

```
// find 1-cover for x
cover ← ∅
found ← False
I⃗ ← ∅; a ← Qry(x)
while not found
    if q_U Up- or q_Q Qry-queries made
        return cover
    y ↞ U \ (I⃗ ∪ {x})
    I⃗ ← I⃗ ∪ {y}
    Up(y); a' ← Qry(x)
    if a' ≠ a :
        cover ← {y}
        found ← True
for i ∈ [2, 3, ..., k] lg : CMS : CK : change
    a ← MinUncover^{Up,Qry}(x, a', cover)
    if a = cover : return cover
    for y ∈ I // in order of insertion to I
        if q_U Up- or q_Q Qry-queries made
            return cover
        Up(y); a' ← Qry(x)
        if a' ≠ a :
            cover ← cover ∪ {y}
            I⃗ ← I \ {y}
            break
return cover
```

Figure 3-5. Cover Set Attack for the CMS in private hash function and private representation setting. The attack is parametrised with the update and query query budget q_U and q_Q.

In Figure 3-5 we give an attack for the private hash and private representation setting. This is the most challenging setting for finding a cover set: the privacy of the hash functions makes local

hash computations effectively useless, and the privacy of the representation prevents the adversary from using it to view the result of online hash computations. The attack begins by querying x to learn its current frequency estimate; let $(p_1, p_2, \ldots, p_k) \leftarrow R(K, x)$ and let $M[1][p_1] = c_1, \ldots, M[k][p_k] = c_k$ be the values of the counters associated to x at this time, i.e., $\min_{i \in [k]}\{c_i\} = a \geq 0$.

The attack then inserts distinct random elements that are not equal to x, checking the estimated frequency after each insertion until the estimated frequency for x increases to $a + 1$, as this signals that a cover set for x has been inserted. Let $\vec{I}$ be the stream of inserted elements at the moment that this happens. At this point, we begin the first "round" of extracting from $\vec{I}$ a 1-cover. Say the last inserted element was z_1. As this caused the CMS estimate to increase, z_1 must share at least one counter with x. Moreover, any counter covered by z_1 must have been minimal, i.e., still holding its initial value c_i, at the time that z_1 was inserted. Thus, we set our round-one candidate cover set $C_1 \leftarrow \{z_1\}$. Notice that by definition, the insertion of z_1 increases the estimation error by one.

Let $\mathcal{M}(C_1) = \{i \in [k] \mid \exists z \in C : R(K, z)[i] = p_i\}$, i.e., the set of rows whose x-counters are covered by C_1, and let $\delta_1 = \min_{j \notin \mathcal{M}(C_1)}\{M[j][p_j]\} - \min_{i \in \mathcal{M}(C_1)}\{M[i][p_i]\}$. Notice that δ_1 is the gap between the smallest counter(s) *not* covered by C_1, and the smallest counter(s) that are covered by C_1. (Observe that z_1 may also cover non-minimal x-counters.) Thus, if we now reinsert C_1 a total of δ_1 times, this gap shrinks to zero; reinserting it once more will cause some x-counter that is *not* covered by C_1 to become minimal, and we can observe this by making an estimation query (i.e. a **Qry** call) after each reinsertion.

Example: Say we have $k = 4$, and prior to the first insertion of z_1 (as part of $\vec{I}$) we have $M[1][p_1] = 2$, $M[2][p_2] = 3$, $M[3][p_3] = 5$ and $M[4][p_4] = 0$. Now, say that z_1 covers the x-counters in rows 1,4: then upon first inserting z_1, we have $M[1][p_1] = 3$, $M[2][p_2] = 3$, $M[3][p_3] = 5$ and $M[4][p_4] = 1$. We create $C_1 = \{z_1\}$, and compute $\delta_1 = 3 - 1 = 2$. If we were to insert C_1 twice more, we would have $M[1][p_1] = 5$, $M[2][p_2] = 3$, $M[3][p_3] = 5$ and $M[4][p_4] = 3$; if we had checked the CMS estimate for n_x after each insertion, we would have

observed responses 2 and 3. After $\delta_1 + 1 = 3$ re-insertions of C_1, we would have $M[1][p_1] = 6$, $M[2][p_2] = 3$, $M[3][p_3] = 5$, $M[4][p_4] = 4$, and the CMS estimate of n_x would remain 3 because now $M[2][p_2]$ is minimal. ◦

Notice that the $\delta_1 + 1$ re-insertions of C_1 will increase the CMS estimate of n_x by exactly δ_1. At this point we begin round 2, searching for $z_2 \in \vec{I} \setminus C_1$ that covers the newly minimal x-counters. Recall that the elements of $\vec{I}$ are distinct (by design), so if we reinsert $\vec{I} \setminus C_1$ *in order* we are guaranteed to hit some satisfying $z_2 \neq z_1$, and this can be observed by checking the CMS estimate of n_x after each element is reinserted. As was the case for z_1, we know that z_2 covers the currently minimal x-counters, and that prior to reinserting z_2 these counters had not changed in value since the end of round 1. Thus, reinserting z_2 increases the estimation error by one. We set $C_2 \leftarrow C_1 \cup \{z_2\}$, and then switch to reinserting C_2 a total of $\delta_2 + 1$ times (where δ_2 is defined analogously to δ_1) to end round 2. Again, this increases the estimation error by δ_2.

Continuing this way, after some $\ell \leq k$ rounds we will have found a complete 1-cover for x. There can be at most k rounds, because each round i adds exactly one new element z_i to the incomplete cover C_{i-1}, and there are only k counters to cover. Notice that in round ℓ, when we reinsert C_ℓ we will never observe that some new x-counter has become minimal: all x-counters are covered by C_ℓ, so all will be increased by each reinsertion. Nonetheless, each reinsertion of C_ℓ adds one to the estimation error, and these re-insertions may continue until the resource budget is exhausted, i.e., until a total of q_U elements have been inserted (via **Up**) as part of the attack.

The number of **Up**-queries (i.e. insertions) required to reach the complete cover C_ℓ is

$$q'_U \leq \ell|\vec{I}| + \sum_{i=1}^{\ell-1}(\delta_i + 1)(i) = \ell|\vec{I}| + \frac{\ell(\ell-1)}{2} + \sum_{i=1}^{\ell-1} i\delta_i$$

and so C_ℓ can potentially be reinserted at least $\lfloor (q_U - q'_U)/\ell \rfloor$ times, each time adding one to the estimation error. We say *potentially* because the **Qry**-query budget may be the limiting factor; we'll return to this in a moment. For now, assuming q_Q is not the limiting factor, the error

introduced by the attack is

$$\begin{aligned}\mathsf{Err} &\geq \left\lfloor \left(\ell + \sum_{i=1}^{\ell-1} \delta_i\right) + \left(\frac{q_U - \ell|\vec{I}| - \frac{\ell(\ell-1)}{2} - \sum_{i=1}^{\ell-1} i\delta_i}{\ell}\right)\right\rfloor \\ &= \left\lfloor \left(\frac{\ell+1}{2} + \frac{1}{\ell}\left(q_U + \sum_{i=1}^{\ell-1}(\ell-i)\delta_i\right) - L^1\right)\right\rfloor\end{aligned}$$

where the final line holds because $|\vec{I}|$ is, by construction, precisely L_1. We note that Err is a function of several random variables: $L^1, \ell, \{\delta_i\}_{i\in[\ell-1]}$.

We would like to develop an expression for $\mathbb{E}[\mathsf{Err}]$, so we observe that for practical values of k, m (e.g., $k = 4$, with $m \gg k$) it is likely that $\ell = k$. We have $\ell < k$ only if one or more of the covering elements cover multiple x-counters, and for small $k \ll m$ this is unlikely. We approximate Err with $\widehat{\mathsf{Err}}$ by replacing ℓ with k, dropping the flooring operation, arriving at

$$\mathbb{E}[\mathsf{Err}] \approx \mathbb{E}[\widehat{\mathsf{Err}}] \approx \left(\frac{k+1}{2} + \frac{1}{k}\left(q_U + \sum_{i=1}^{k-1}(k-i)\mathbb{E}[\delta_i]\right) - \mathbb{E}[L^1]\right)$$

Rearranging and using the very tight approximation $\mathbb{E}[L^1] \approx mH_k$, we have

$$\mathbb{E}[\widehat{\mathsf{Err}}] \approx \left(\frac{q_U}{k} - mH_k\right) + \frac{k+1}{2} + \left(\frac{1}{k}\sum_{i=1}^{k-1}(k-i)\mathbb{E}[\delta_i]\right)$$

We do not have a crisp way to describe the distribution of the δ_i random variables, but we can make some educated statements about them. The expected value of *any* counter $M[i][j]$ after $\vec{I}$ has been inserted is $|\vec{I}|/m \approx mH_k/m = H_k$, and $H_k < 4$ for $k \leq 30$ (and practical values of k are typically much less than 30); moverover, standard balls-and-bins arguments tell us that as the number of balls approaches $m \ln m$, the *maximum* counter value in any row approaches the expected value. Since $\delta_1 \leq \max_{j\notin\mathcal{M}(\{z_1\})}\{M[j][p_j]\} - \min_{i\in\mathcal{M}(\{z_1\})}\{M[i][p_i]\}$, we can safely assume that $\mathbb{E}[\delta_1]$ is upper-bounded by a constant that is small relative to $m, q_U/k$.

After inserting $C_1 = \{z_1\}$ a total of $\delta_1 + 1$ times, we switch to reinserting $\vec{I} \setminus C_1$ until we find a z_2 that covers the currently minimal x-counters. When we begin to reinsert C_2, we know by construction that $\delta_2 \leq \min_{j \notin \mathcal{M}(\{z_1,z_2\})}\{M[j][p_j]\} - \left(\min_{j \notin \mathcal{M}(\{z_1\})}\{M[j][p_j]\} + 1\right)$. For the first term in the difference, we "roll back" one round; say that $\alpha = \max_{j \notin \mathcal{M}(\{z_1\})}\{M[j][p_j]\}$. Then, being very pessimistic, we know that $\min_{j \notin \mathcal{M}(\{z_1,z_2\})}\{M[j][p_j]\} \leq 2\alpha + (\delta_1 + 1)$: in finding z_2, we reinsert at most all of $\vec{I} \setminus \{z_1\}$, which would add another (at most) α to that maximum counter value, and the repeated insertions of C_1 could have added at most $\delta_1 + 1$ to said maximum counter. However, the second term in the difference is at least $\delta_1 + 1$, so $\delta_2 \leq 2\alpha$ and we have already argued that α is in the neighborhood of $H_k < 4$. Continuing this this way, we reach the conclusion that the dominant term in $\mathbb{E}[\widehat{\mathsf{Err}}] \approx \mathbb{E}[\mathsf{Err}]$ will be $\frac{q_U}{k} - mH_k$. This is observed experimentally in Table 3-2. For realistic values of k, significant error will be created when $q_U \gg (mk)H_k$. For example, when $k = 4, m = 2048$ we require $q_U \gg 17067$; this is likely not a real restriction in most practical use-cases of CMS, e.g., computing the heavy hitter flows traversing a router.

Returning to the matter of exhausting the **Qry**-budget, the total number of **Qry**-queries for the attack depends somewhat heavily on whether or not $\ell = k$. If $\ell = k$ then $|C_k| = k$, and we know that a complete cover has been found. Thus, we do not need to make any **Qry**-queries during reinsertions of C_k. If $\ell < k$, however, then we must make **Qry**-queries during reinsertions of C_ℓ, because we do not know that C_ℓ contains a complete cover.

Either way, the number of **Qry**-queries need to reach C_ℓ is $q'_Q \leq 1 + \ell|\vec{I}| + \sum_{i=1}^{\ell-1}(\delta_i + 1)$, and the expected gap between q'_U and q'_Q is

$$\begin{aligned}
\mathbb{E}[q'_U - q'_Q] &\approx \mathbb{E}\left[\sum_{i=1}^{\ell-1} i(\delta_i + 1) - \sum_{i=1}^{\ell-1}(\delta_i + 1)\right] \\
&\leq \mathbb{E}\left[\sum_{i=1}^{\ell-1} \delta_i\right] + \frac{(k-1)(k-2)}{2} \\
&\leq k\mathbb{E}\left[\max_{i \in [\ell-1]}\{\delta_i\}\right] + \frac{(k-1)(k-2)}{2}
\end{aligned}$$

By the arguments just given about the δ_i, we can safely bound $\mathbb{E}\left[\max_{i\in[\ell-1]}\{\delta_i\}\right]$ by kH_k. So $\mathbb{E}[q'_U - q'_Q] = O(k^2)$ with a small hidden constant. Thus, the expected numbers of **Up**-queries and **Qry**-queries expended to find the complete cover C_ℓ are similar, especially for realistic values of k.

Now, in the most likely case that $\ell = k$, no further **Qry**-queries are needed. Hence, when $\ell = k$, the overall error induced by the attack will be determined by the insertion/**Up**-budget (q_U) when the *total* **Qry**-budget q_Q is approximately the insertion-budget required for finding the cover. When $\ell < k$, in order for the overall error to be determined by the insertion budget, the total **Qry**-budget needs to accommodate $q'_Q + (q_U - q'_U)/\ell$ queries. The second summation comes from the fact that while accumulating error via re-insertions of C_ℓ, we must make one **Qry**-query per reinsertion. This is a potentially large jump in the number of estimation queries required, from $\ell = k$ to $\ell < k$. But in reality the jump might be less important than it appears: if $\ell < k$ then given our intuition about the δ_i, it seems likely that if some C_i is taking a large number of insertions, one can likely assume that C_i is a complete cover, cease making estimation queries and switch to an insertion only strategy.

Public hash and private representation setting. Observe that the public representation is never used in our attack in the public hash and public representation setting. Therefore, in this public hash and private representation setting, the same attack can be used. The same analysis applies.

Private hash and public representation setting. The public representation allows for an attack similar to our attack in the public hash settings (Figure 3-6). Here, we use the **Up**-oracle instead of the **Hash**-oracle to find a cover. By comparing the state before and after adding an element it is easy to deduce the element's counters (as they are the only ones to change). Our attack first adds the target to get its counters. Then, we keep inserting *distinct* elements, comparing the state before and after until a cover C is found. By the definition of L^1, the cover is found with $(q'_U = 1 + L^1)$ **Up**-queries, and is after reinserted $\lfloor (q_U - q'_U)/|C| \rfloor$ times, each time adding one to the estimation

error. Hence, $\mathsf{Err} \geq \lfloor (q_U - 1 - L^1)/|C| \rfloor \geq \lfloor (q_U - 1 - L^1)/k \rfloor$ and $\mathbb{E}[\mathsf{Err}] \geq \frac{q_U - 1 - \mathbb{E}[L^1]}{k} \approx \frac{q_U - mH_k}{k}$.

```
CoverAttack^Up(x, ⊥, repr)
1 : cover ← FindCover^Up(1, x, repr)
2 : until q_U Up-queries made:
3 :     for e ∈ cover: Up(e)
4 : return done
```

```
FindCover^Up(r, x, repr)
1 : cover ← ∅; found ← False
2 : I ← ∅; tracker ← zeros(k)
3 : repr' ← Up(x)
4 : / compute x's indices
5 : for i ∈ [k]
6 :     for j ∈ [m]
7 :         if repr'[i][j] ≠ repr[i][j]
8 :             p_i ← j; break;
9 : while not found
10 :    if q_U Up-queries made : return ∅
11 :    y ↞ U \ (I ∪ {x})
12 :    I ← I ∪ {y}
13 :    repr ← repr'
14 :    repr' ← Up(y)
15 :    / compute y's indices
16 :    for i ∈ [k]
17 :        for j ∈ [m]
18 :            if repr'[i][j] ≠ repr[i][j]
19 :                q_i ← j; break;
20 :    for i ∈ [k]
21 :        / compare x's and y's indices row by row
22 :        if p_i = q_i and tracker[i] < r
23 :            cover ← cover ∪ {y}
24 :            tracker[i] + = 1
25 :    if sum(tracker) = rk
26 :        found ← True
27 : return cover
```

Figure 3-6. Cover Set Attack for the CMS in private hash function and public representation setting. The attack is parametrized with the update query budget q_U.

3.4.3 Cover-Set Attacks on HK

By examining the HK pseudocode, it is not hard to see that when a stream $\vec{S}$ satisfying the NFC condition is inserted in the HK structure, over-estimations are not possible; any error in frequency

estimates is due to underestimation. We also note that if $\vec{S}$ satisfies the NFC condition, then any cover that it contains for $x \in \mathcal{U}$ must be a $(\mathcal{FP}_x, x, r)$-cover. In attacking HK, we will build $(\mathcal{FP}_x, x, 1)$-covers; as such, in this section we will often just say "cover" as shorthand.

The intuition for our HK-attacks is, loosely, as follows. If one repeatedly inserts a cover for x, *before x is inserted*, then the counters associated to x will be owned by members of the cover, and the counter values can be made large enough to prevent any subsequent appearances of x from decrementing these counters with overwhelming probability. We will sometimes say that such hard-to-decrement counters are "locked-down". As such, the HK estimate $\hat{n}_x$ will be zero, even if $n_x \gg 0$.

We note that attacks of this nature would be particularly damaging in instances where the underlying application uses HK to identify the most frequent elements in a stream $\vec{S}$. With relatively few insertions of the cover set, one would be able to hide many occurrences of x. DDoS detection systems, for example, rely on compact frequency estimators to identify communication end-points that are subject to an abnormally large number of incoming connections [25]. In this case, the target x is an end-point identifier (e.g., an IP address and/or TCP port). Being able to hide the fact that the end-point x is a "heavy hitter" in the stream of incoming flow destinations could result in x being DDoSed.

Interestingly, while a cover is necessary to cause a frequency estimation error for x, it is *not* sufficient. Unlike the CMS, whose counters are agnostic of the order of elements in the stream, the HK counters have a strong dependence on order. Thus, if x is a frequent element and many of its appearances are at the beginning of the stream, then *it* can lock-down its counters; a cover set attack is still possible, but now the number of times the cover must be inserted may be much larger than the frequency (so far) of x.

Setting the attack parameter t. Say our attack's resource budget is (q_H, q_U, q_Q). The HK attacks find a cover $C = \{z_1, z_2 \dots\}$ and then inserts it t times. We set the value t such the

probability p of decrementing the any of the target's counters with subsequent insertions of x is sufficiently small. For our experiments we set $p = 2^{-128}$.

Let D_i^t be the event that at the end of the attack $A[i][p_i].\text{fp} = \text{fp}_x$ given that at some point during the attack we had $A[i][p_i].\text{cnt} = t$ with $A[i][p_i].\text{fp} = \text{fp}_{z_i}, z_i \neq x$. Let $\left(D^t\right) = \bigvee_{i=1} D_i^t$. Then, $\Pr[D_i^t] \leq \binom{q_U}{t} \prod_{j=1}^{t} d^j \leq (q_U)^t \, d^{\frac{t(t+1)}{2}}$.

Say $f(t) = k \, (q_U)^t \, d^{\frac{t(t+1)}{2}}$. If the attack set $A[i][p_i].\text{cnt} = t$ with $A[i][p_i].\text{fp} = \text{fp}_{z_i}, z_i \neq x$ for each i, then the probability of x overtaking any of its counters by the end of the attack is bounded by $\Pr[\bigvee_{i=1}^{k} D_i^t] \leq f(t)$.

Public hash and public representation setting. This attack (Figure 3-7) is similar to the CMS attack for the public hash setting, but with a few tweaks. The cover is inserted only t times and then the **Up** budget is exhausted by inserting target x (at least $(q_U - tk)$ times) to accumulate error. If $\neg D^t$ then this process introduces the error of at least $(q_U - tk)$. Thus, as the cover finding step uses **Hash** only and induces no error,

$$\mathbb{E}\left[\mathsf{Err}\right] \geq (q_U - tk)(1-p) \Pr\left[\mathsf{Cover}_x^1 \mid Q = 0\right].$$

For the term $\Pr\left[\mathsf{Cover}_x^1 \mid Q = 0\right]$ we can simply apply the same bound as for the CMS attack (Equation (3-4)) obtaining

$$\mathbb{E}\left[\mathsf{Err}\right] \geq (q_U - tk)(1-p)\left(1 - (1 - 1/m)^{\frac{q_H}{k} - 1}\right)^k.$$

Private hash and private representation setting. We present the attack for this setting in Figure 3-8. The attack starts by inserting x once. Starting with an empty HK implies that then x owns all of its buckets, i.e., $A[i][p_i].\text{fp} = \text{fp}_x$ for all rows i, with their associated counters $c_1, \ldots, c_k$ set to one, setting x's current frequency estimate $a = \max_{i \in [k]} \{c_i\} = 1$. The attack then

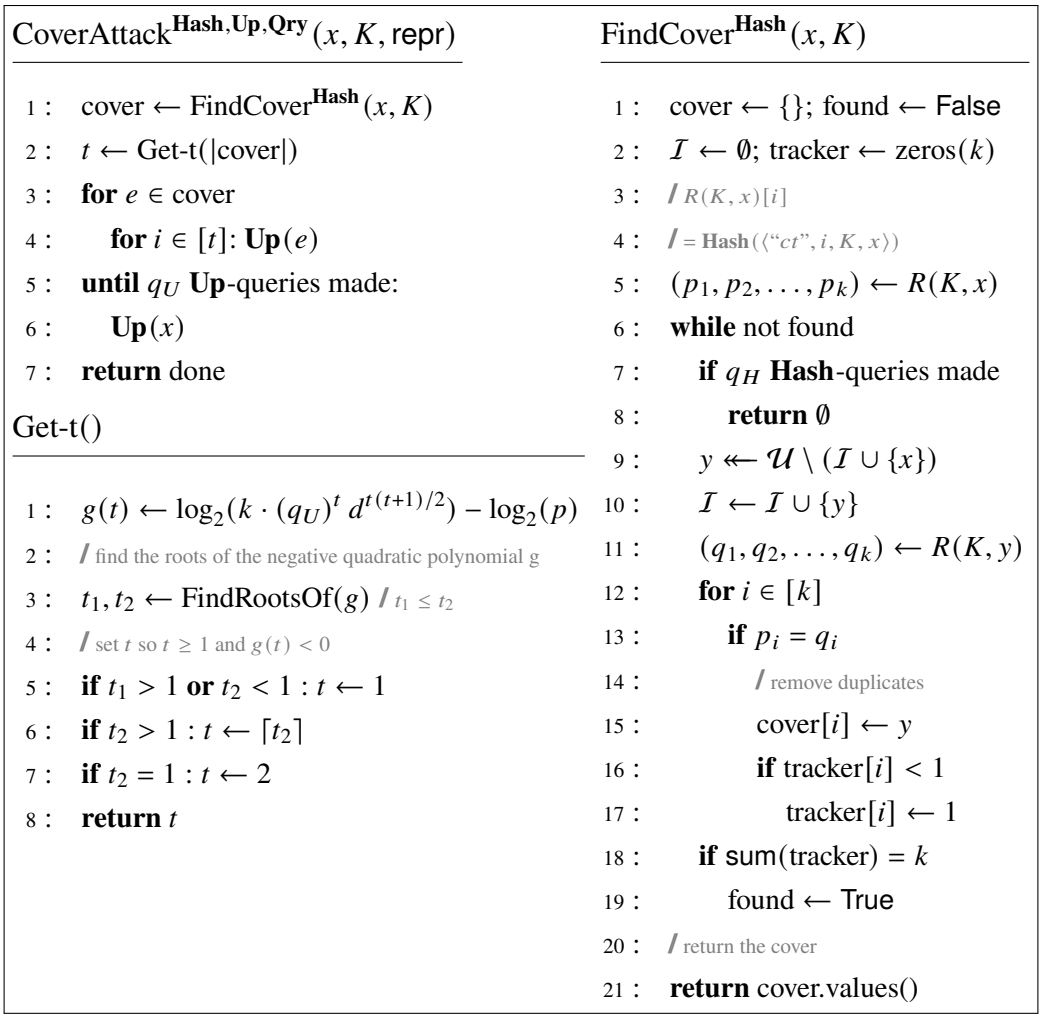


Figure 3-7. Cover Set Attack for the HK in public hash function setting. We use $R(K, x)$ to mean $(\mathbf{Hash}(\langle$"ct", $1, K, x\rangle), \mathbf{Hash}(\langle$"$ct$", $2, K, x\rangle, \ldots, \mathbf{Hash}(\langle$"$ct$", $k, K, x\rangle)))$. The attack is parametrized with the update and **Hash** query budget q_U and q_H.

keeps inserting *distinct* elements until the frequency estimate for x drops to 0, i.e, $A[i][p_i].\text{fp} \neq \text{fp}_x$ for all rows i.

Let $\mathcal{I}_1$ be the set of inserted elements $\neq x$ at the moment that this happens, and the last inserted element was z_1. Then, z_1 must share at least one counter with x (the one that changed $A[i][p_i].\text{fp}$ from fp_x most recently). So, we set our round-one candidate cover set $C_1 \leftarrow \{z_1\}$ and insert t times to the HK. Now we are at the point when all $c_1, \ldots, c_k$ are owned by elements $\neq x$, and, under the NFC condition, all but one are of value one. Note that inserting $\mathcal{I}_1$ increased the estimate error by one.

CoverAttack$^{\mathbf{Up},\mathbf{Qry}}(x, \perp, \perp)$

```
1 : FindInsertCover^{Up,Qry}(x)
2 : until q_U Up-queries made:
3 :     Up(x)
4 : return done
```

Reintro$^{\mathbf{Up},\mathbf{Qry}}(x)$

```
1 : / reintroduce target x
2 : while True
3 :     if q_U Up- or q_Q Qry-queries made:
4 :         return
5 :     Up(x); a ← Qry(x)
6 :     if a > 0 : return
7 : endwhile
8 :
```

FindInsertCover$^{\mathbf{Up},\mathbf{Qry}}(x)$

```
1 : / insert ≤ k elements t times in a row
2 : cover ← ∅
3 : t ← Get-t()
4 : I ← ∅
5 : for i ∈ [1, 2, ..., k]k − priv − priv − a − yi
6 :     Reintro^{Up,Qry}(x)
7 :     while True
8 :         if q_U Up- or q_Q Qry-queries made
9 :             return
10 :        y ↞ U \ (I ∪ {x})
11 :        I ← I ∪ {y}
12 :        Up(y); a ← Qry(x)
13 :        if a = 0 :
14 :            cover ← cover ∪ {y}
15 :            for j ∈ [t] : Up(y)
16 :            break
17 : return
```

Figure 3-8. Cover Set Attack for the HK in private hash function and representation setting. The attack is parametrised with the update and query query budget q_U and q_Q. The attack uses the function Get-t(.) from Figure 3-7.

The adaptive portion of our attack proceeds as follows. In each round $i = 2, \ldots$ we first keep reinserting x until $\mathrm{HK}(x)$ reaches 1. Let d_i be the number of these reinsertions. Hence, these reinsertions increased the estimate error by $d_i - 1$. At this point, at least one counter $c_1, \ldots, c_k$ is owned by x and all counters owned by x are set to 1. Then, we search for a new element to create our round-i cover set candidate C_i, by inserting *new distinct* elements, until we find a z_i that drops $\mathrm{HK}(x)$ to 0. We set $C_i \leftarrow C_{i-1} \cup \{z_i\}$ and insert z_i t times. At this point, all counters $c_1, \ldots, c_k$ are owned by elements $\neq x$ again, and all are of value one, but the ones covered by C_i which (very likely) hold a value strictly greater than 1 and (very) close to t.

The procedure ensures that after some $\ell \leq k$ rounds we have found a complete 1-cover with (very) high probability. Each round i adds maximally one new element to the incomplete cover C_{i-1}. The added element covers whatever x is owning at the beginning of the round. Thus, with (very) high

probability, counters owned by x in the round are not covered by C_{i-1}. This is because all the counters covered by C_{i-1} were set to value t (or a value close to t with very high probability[3]) at some point, and the selection of t makes the probability of later overtaking one such counter (very) small. There are only k counters to cover and so with (very high) probability having only k rounds suffices to find a cover.

Let $\mathcal{I}_i$ be the set of inserted elements $\neq x$ in each round. We get the number of **Up**-queries required to complete k rounds is

$$q'_U \leq \sum_{i=1}^{k}(d_i + |\mathcal{I}_i| + t) = \sum_{i=1}^{k}(d_i + |\mathcal{I}_i|) + tk. \tag{3-5}$$

So, x can be potentially inserted $q_U - q'_U$ times, accumulating some additional error[4]. Let us assume that q_Q is not the limiting resource in the attack. Say C is the attack's maximal round candidate cover. Whenever $\neg\left(D^t\right)$, adding z_i t times to the HK incremented one of the x's counters, not yet set to value t by elements in C_{i-1}. If, in addition, we have $|C| = k$, k different elements set k different counters of x (i.e. all of x's counters) to t making them impossible to decrement later. Therefore, after the rounds to reach C are completed every further insertion of x ($q_U - q'_U$ of them) increased the error by 1. Note that $|C| = k$ implies the attack completed exactly k rounds and

$$\begin{aligned}
&\left[\mathsf{Err} \mid \neg\left(D^t\right), |C| = k\right] \\
&\geq \sum_{i=1}^{k}(d_i) + q_U - \sum_{i=1}^{k}\left(d_i + \left[|\mathcal{I}_i| \mid \neg\left(D^t\right), |C| = k\right]\right) - tk \\
&\geq q_U - \sum_{i=1}^{k}\left[|\mathcal{I}_i| \mid \neg\left(D^t\right), |C| = k\right] - tk.
\end{aligned}$$

[3] We could have z_i simultaneously covering more not yet covered counters. Then, adding z_i t times fixes one counter to t, and the others to t with the probability ≥ 0.9 – the other counters might have been owned by some others elements but are definitely of value one, so each of them gets "taken" by z_i in the first insertion with probability 0.9.

[4] We say potentially as the **Qry**-query budget might be a limiting factor.

Let D_i be the set of rows j with $A[j][p_j].\text{fp} = \text{fp}_x$ (i.e. x owning the counter), and let $c_{i,j}$ be the values of $A[j][p_j].\text{cnt}$ after the i-th round reinsertion step. Say $Y_{i,j}$ counts the minimal number of distinct element insertions to "overtake" the counter from x in row $j \in D_i$ after the i-th round reinsertion step, i.e., the minimal number of distinct evaluations of $R(K.\cdot)$ to set $A[j][p_j].\text{fp} \neq \text{fp}_x$. Then, $Y_{i,j}$ is a geometric random variable with $p = \frac{d^{c_{i,j}}}{m}$, $d^{c_{i,j}}$ coming from the probabilistic decay mechanism. Moreover, $c_{i,j} = 1$ for all $j \in D_i$ – that is counters owned by x equal 1 after every reinsertion step. As $|D_1| = k$ we have that $|\mathcal{I}_1| = \max_{j \in D_1} \{Y_{1,j}\}$ is essentially L^1 with $p = \frac{d}{m}$. Since $|D_i| \leq k$ and all $Y_{i,j}$ are positive and i.i.d. geometric variables with $p = \frac{d}{m}$, we have that $\mathbb{E}[|\mathcal{I}_i|] \leq \mathbb{E}[|\mathcal{I}_1|]$. So, $\mathbb{E}[|\mathcal{I}_i|] \leq \frac{m}{d} H_k$. This implies that

$$
\begin{aligned}
\mathbb{E}[\mathsf{Err}] &= \sum_{s=0}^{k} \mathbb{E}\left[\mathsf{Err} \mid \left(D^t\right), |C| = s\right] \Pr\left[\left(D^t\right) \wedge |C| = s\right] \\
&\quad + \sum_{s=0}^{k} \mathbb{E}\left[\mathsf{Err} \mid \neg\left(D^t\right), |C| = s\right] \Pr\left[\neg\left(D^t\right) \wedge |C| = s\right] \\
&\geq \mathbb{E}\left[\mathsf{Err} \mid \neg\left(D^t\right), |C| = k\right] \Pr\left[\neg\left(D^t\right) \wedge |C| = k\right] \\
&\geq (q_U - tk) \Pr\left[\neg\left(D^t\right) \wedge |C| = k\right] - \sum_{i=1}^{k} \mathbb{E}[|\mathcal{I}_i|] \\
&\geq (q_U - tk) \Pr\left[\neg\left(D^t\right) \wedge |C| = k\right] - \frac{km}{d} H_k.
\end{aligned}
$$

We expect $\Pr\left[\neg\left(D^t\right) \wedge |C| = k\right] \approx 1$ and $\mathbb{E}[\mathsf{Err}] \approx q_U - tk - \frac{km}{d} H_k$. We confirmed this experimentally as seen in Table 3-2.

Public hash and private representation setting. As with the CMS, the same attack and analysis applies from the public hash and public representation setting.

Private hash and public representation setting. The public representation allows us to design an attack similar to the attack for the public hash settings, but, as with the CMS attack in the setting, we need to find the cover using the **Up** oracle. Starting with an empty filter, the attack first inserts x, such that x is guaranteed to own all of its counters. Then, we keep adding *distinct*

CoverAttack$^{\mathbf{Up},\mathbf{Qry}}(x, \perp, \mathsf{repr})$

```
cover ← FindCover^Up(x, repr)
t ← Get-t()
for e ∈ cover
    for i ∈ [t]: Up(e)
until q_U Up-queries made:
    Up(x)
return done
```

FindCover$^{\mathbf{Up}}(x, \mathsf{repr})$

```
cover ← {}; found ← False
I ← ∅; tracker ← zeros(k)
repr′ ← Up(x)
/ compute x's indices
for i ∈ [k]
    for j ∈ [m]
        if repr′[i][j].fp ≠ repr[i][j].fp
            p_i ← j; break;
while not found
    if q_U Up-queries made : return ∅
    y ↞ U \ (I ∪ {x})
    I ← I ∪ {y}
    repr ← repr′
    repr′ ← Up(y)
    / compute y's indices
    for i ∈ [k]
        q_i ← False
        for j ∈ [m]
            if repr′[i][j].fp ≠ repr[i][j].fp
                q_i ← j; break;
    for i ∈ [k]
        / compare x's and y's indices row by row
        if q_i ≠ False and p_i = q_i
            / remove duplicates
            cover[i] ← y
            if tracker[i] < 1
                tracker[i] ← 1
    if sum(tracker) = k
        found ← True
/ cover elements own the target's counters
return cover.values() / all counters set to 1
```

Figure 3-9. Cover Set Attack for the HK in private hash function and public representation setting. The attack is parametrized with the update query budget q_U. The attack uses the function Get-t(.) from Figure 3-7.

elements, until all the $A[i][p_i]$.fp that once belonged to x has changed, in turn signaling the cover for x has been found. We give a pseudocode description of this attack in Figure 3-9.

Adding any $y \neq x$ has $\frac{d}{m}$ probability to change $A[i][p_i]$.fp after the single initial insertion of x. Let Y_i be the minimal number of distinct element $\neq x$ insertions before $A[i][p_i]$.fp changes from fp_x. We observe that Y_i is a geometric random variable with success probability $p = \frac{d}{m}$. Set $Y = \max_{i \in [k]}\{Y_i\}$. So, our cover-finding step requires $(q'_U = 1 + Y)$ **Up**-queries to complete - 1 query to insert x, and then Y to find a cover. Say q_U is the total **Up**-query budget. After the cover finding step, we insert cover C t times, to lock-down the counters followed by $q_U - q'_U - t|C|$ insertions of x. Each x-insertion added one to the error if $\neg\left(D^t\right)$ and

$$\begin{aligned}
\mathbb{E}[\mathsf{Err}] &\geq \mathbb{E}\left[\mathsf{Err} \mid \neg\left(D^t\right)\right] \Pr\left[\neg\left(D^t\right)\right] \\
&\geq (q_U - 1 - \mathbb{E}[Y] - tk) \Pr\left[\neg\left(D^t\right)\right] \\
&\approx q_U - \frac{m}{d} H_k - tk.
\end{aligned}$$

The last approximation comes from assuming t is set such that $\Pr\left[\neg\left(D^t\right)\right] \approx 1$, and observing that Y is essentially L^1 with $p = \frac{d}{m}$ (i.e. m replaced with $\frac{m}{d}$).

3.5 Count-Keeper

In Figure 3-10 we present the Count-Keeper (CK) data structure. At a high level, CK uses information from both CMS and HK (with $d = 1$) to create frequency estimates that are more accurate than either CMS or HK (alone) when the stream is "honest", and that are more robust in the presence of adversarial streams. After describing the structure, we will provide analytical support for its design, i.e., why it is more accurate and robust. To summarize this very briefly and informally: CK is more accurate because its HK component can decrease the effect of "collision noise" that drives up the values held at the relevant $M[i][p_i]$ in the CMS component; and it is more robust because a 1-cover no longer suffices to create estimation errors (minimally, a 2-cover is needed) and, unlike either CMS or HK alone, CK can detect when the state of M, A is "abnormal" and prone to producing spurious estimates.

3.5.1 Structure

At initialization, the CK initializes a standard CMS (initialized in the structure as M) and a HK with the decay parameter $d = 1$ (initialized in the structure as A) in their usual way. We set the substructures to be of the same number of rows and buckets and let the elements hash to the same counters' positions in each substructure using the same row hash functions.

To insert a stream element x arrives, we run the CMS and HK update procedures $M \leftarrow \text{UP}_K^{\text{CMS}}(M, \mathsf{up}_x)$ and $A \leftarrow \text{UP}_K^{\text{HK}}(M, \mathsf{up}_x)$, respectively. We note that the same positions $(p_1, \ldots, p_k) \leftarrow R(K, x)$ are visited in both procedures; thus the same elements are observed by $M[i][p_i]$ and $A[i][p_i]$. By "observed", we mean that both $M[i][p_i]$ and $A[i][p_i]$ maintain summary information about the same substream, namely the substream of elements z such that $p_i = R(K, z)[i]$.

When queried for the frequency estimate of an element $x \in \mathcal{U}$, CK first computes the CMS and HK estimates, which we will write as CMS(x) and HK(x) for brevity. If CMS(x)=HK(x), then we return their shared response. We will see precisely why this is the correct thing to do, but loosely, it is because (under the NFC assumption) $\text{HK}(x) \leq n_x \leq \text{CMS}(x)$. If $\text{CMS}(x) \neq \text{HK}(x)$ then CK proceeds row-by-row, using the information held at $A[i][p_i]$ to refine the summary information held at $M[i][p_i]$. If any of the $A[i][p_i]$.fp are uninitialized, then we are certain that $n_x = 0$; had *any* stream element been mapped to this position, the fingerprint would no longer be uninitialized. In this case, CK(x) returns 0.

Now assume that none of the $A[i][p_i]$ have uninitialized fingerprints, and $\text{CMS}(x) \neq \text{HK}(x)$. To explain our row-by-row refinements, let us define two sets $I_x = \{i \in [k] \mid A[i][p_i].\text{fp} = \text{fp}_x\}$ and $\hat{I}_x = \{i \in [k] \mid A[i][p_i].\text{fp} \neq \text{fp}_x\}$, i.e., the subset of rows in M (and A) that are "owned" and not "owned" (resp.) by x. Observe that we can write the CMS estimate for x as

$$\text{CMS}(x) = \min \left\{ \min_{i \in I_x} \{M[i][p_i]\}, \min_{i \in \hat{I}_x} \{M[i][p_i]\} \right\}$$

$\textsc{Rep}_K(\mathcal{S})$

1 : $M \leftarrow \text{zeros}(k, m)$
2 : **for** $i \in [k]$
3 : $\quad A[i] \leftarrow [(\star, 0)] \times m$
4 : repr $\leftarrow \langle M, A \rangle$
5 : **for** $x \in \mathcal{S}$
6 : $\quad$ repr $\twoheadleftarrow \textsc{Up}_K(\text{repr}, \text{up}_x)$
7 : **return** repr

$\textsc{Up}_K(\text{repr}, \text{up}_x)$

1 : $\langle M, A \rangle \leftarrow$ repr
2 : $M \twoheadleftarrow \textsc{Up}_K^{\text{CMS}}(M, \text{up}_x)$
3 : $A \twoheadleftarrow \textsc{Up}_K^{\text{HK}}(A, \text{up}_x)$
4 : **return** repr$\leftarrow \langle M, A \rangle$

$\textsc{Qry}_K(\text{repr}, \text{qry}_x)$

1 : $\langle M, A \rangle \leftarrow$ repr
2 : $(p_1, \ldots, p_k) \leftarrow R(K, x)$, $\text{fp}_x \leftarrow T(K, x)$
3 : $\Theta_1, \Theta_2 \leftarrow \infty$
4 : / CMS only overestimates
5 : $\text{cnt}_{\text{UB},x} \leftarrow \textsc{Qry}_K^{\text{CMS}}(M, \text{qry}_x)$
6 : / HK only underestimates
7 : $\text{cnt}_{\text{LB},x} \leftarrow \textsc{Qry}_K^{\text{HK}}(A, \text{qry}_x)$
8 : / return upperbound if equal to lowerbound
9 : **if** $\text{cnt}_{\text{UB},x} = \text{cnt}_{\text{LB},x}$
10 : $\quad$ **return** $\text{cnt}_{\text{UB},x}$
11 : **for** $i \in [k]$
12 : $\quad$ / if never observed
13 : $\quad$ **if** $A[i][p_i].\text{fp} = \star$
14 : $\quad\quad \text{cnt}_{\text{UB},x} \leftarrow 0$
15 : $\quad\quad$ **return** 0
16 : $\quad$ / upper bound adjustment
17 : $\quad$ / x does not own counter
18 : $\quad$ **else if** $A[i][p_i].\text{fp} \neq \text{fp}_x$
19 : $\quad\quad \Theta \leftarrow \frac{M[i][p_i] - A[i][p_i].\text{cnt}+1}{2}$
20 : $\quad\quad \Theta_1 \leftarrow \min\{\Theta_1, \Theta\}$
21 : $\quad$ / x owns counter
22 : $\quad$ **else if** $A[i][p_i].\text{fp} = \text{fp}_x$
23 : $\quad\quad \Theta \leftarrow \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2}$
24 : $\quad\quad \Theta_2 \leftarrow \min\{\Theta_2, \Theta\}$
25 : $\text{cnt}_{\text{UB},x} \leftarrow \lfloor \min\{\Theta_1, \Theta_2\} \rfloor$
26 : **return** $\text{cnt}_{\text{UB},x}$

Figure 3-10. Keyed structure $\text{CK}[R, T, m, k]$ supporting point-queries for any potential stream element x (qry_x). $\textsc{Qry}_K^{\text{CMS}}$, $\textsc{Up}_K^{\text{CMS}}$, resp. $\textsc{Qry}_K^{\text{HK}}$, $\textsc{Up}_K^{\text{HK}}$, denote query and update algorithms of keyed structure $\text{CMS}[R, T, m, k]$ (Figure 3-2), resp. $\text{HK}[R, T, m, k, 1]$ (Figure 3-3, but note $d = 1$). The parameters are a function $R : \mathcal{K} \times \{0,1\}^* \to [m]^k$, a function $T : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^n$ for some desired fingerprint length n, and integers $m, k \geq 0$. A concrete scheme is given by a particular choice of parameters.

so for each row $i \in [k]$, we have two cases to consider. For each case, CK maintains an internal estimator: when $i \in \hat{I}_x$ the estimator is Θ_1^i, and when $i \in I_x$ the estimator is Θ_2^i. We will talk about each of these, next. The upshot of this discussion is that CK defines $\Theta_1 = \min_{i \in \hat{I}_x}\{\Theta_1^i\}$, $\Theta_2 = \min_{i \in I_x}\{\Theta_2^i\}$, and its return value $\lfloor \min\{\Theta_1, \Theta_2\} \rfloor$ is always at least as good as $\mathrm{CMS}(x)$.

3.5.2 Correcting CMS and Correctness of CK

In what follows, we will assume the NFC condition. For sufficiently large fingerprints (e.g., τ-bit fingerprints where 2^τ is much larger than the number of distinct elements in the stream) this is reasonable. Under this assumption, CK may only overestimate the value of n_x.

Correcting $M[i][p_i]$ when x does not "own" $A[i][p_i]$. By its design as a count-all structure, the value of $M[i][p_i] = n_x + \sum_{y \in V_x^i} n_y$. When $i \in \hat{I}_x$, we claim that $n_x \leq \sum_{y \in V_x^i} n_y$. To see this, observe that if $n_x > \sum_{y \in V_x^i} n_y$ then x *would* own $A[i][p_i]$: we can pair up appearances of x with appearances of elements in $y \in V_x^i$, and because no element of V_x^i has the same fingerprint as x, each pair (x, y) effectively contributes 0 to the value of $A[i][p_i].\mathrm{cnt}$. So if $n_x > \sum_{y \in V_x^i} n_y$, the fingerprint held at $A[i][p_i]$ would be fp_x. Note that if $n_x = \sum_{y \in V_x^i} n_y$ and $i \in \hat{I}_x$, then $A[i][p_i].\mathrm{cnt} = 1$ and some $y \neq x$ was the last insertion. Thus, $A[i][p_i] - 1$ is a lowerbound on the difference $\sum_{y \in V_x^i} n_y - n_x$, i.e., the number of occurrences of $y \in V_x^i$ that are not canceled out by an occurrence of x. Thus, $n_x + A[i][p_i] - 1 \leq \sum_{y \in V_x^i} n_y$, which implies that $M[i][p_i] = n_x + \sum_{y \in V_x^i} n_y \leq 2n_x + A[i][p_i] - 1$.

Lemma 3-1. *Let $\vec{S}$ satisfy the NFC condition, and let $x \in \mathcal{U}$. Then for any $i \in \hat{I}_x$ we have* $n_x \leq \frac{M[i][p_i] - A[i][p_i].\mathrm{cnt}+1}{2} = \Theta_1^i$. ♦

Proof of Lemma 3-1. We can think of the counter $A[i][p_i].\mathrm{cnt}$ as counting the depth of a stack of fingerprint-labeled plates. The rules of the stack are as follows. Upon insertion of x into the CK structure:

1. if $A[i][p_i].\mathrm{cnt} = 0$ then the stack is empty; then push an fp_x-labeled plate and set $A[i][p_i].\mathrm{cnt} \leftarrow 1, A[i][p_i].\mathrm{fp} \leftarrow \mathrm{fp}_x$.

2(a). if $A[i][p_i].\text{cnt} = c > 0$ and $A[i][p_i].\text{fp} = \text{fp}_x$, then push an fp_x-labeled plate on to the stack and increment $A[i][p_i].\text{cnt} \leftarrow c + 1$.

2(b). if $A[i][p_i].\text{cnt} = c > 0$ and $A[i][p_i].\text{fp} \neq \text{fp}_x$, then pop the top (fp-labeled) plate and decrement $A[i][p_i].\text{cnt} \leftarrow c - 1$. If this causes $A[i][p_i].\text{cnt} = 0$, then push an fp_x-labeled plate and set $A[i][p_i].\text{cnt} \leftarrow 1$, $A[i][p_i].\text{fp} \leftarrow \text{fp}_x$.

These stack rules are precisely the CK rules for handling insertions. Now, upon the first insertion to CK, by rule 1 it is clear that all plates on the stack (there is only one of them) have label $A[i][p_i].\text{fp}$, and $A[i][p_i].\text{cnt}$ is the number (1) of plates on the stack. Inductively, assume that $A[i][p_i].\text{cnt} = c > 0$ and all c of the plates on the stack have the same label $A[i][p_i].\text{fp}$. Say that the next insertion is z and $A[i][p_i].\text{fp} = \text{fp}_z$. By rule 2(a), we push an fp_z-plate on to the stack and increment $A[i][p_i].\text{cnt} \leftarrow c + 1$. In this case, by assumption, it remains the case that all plates have the same label equal to $A[i][p_i].\text{fp}$, and there are $c + 1$ of them. Alternatively, if $A[i][p_i].\text{fp} \neq \text{fp}_z$ then by rule 2(b) we pop the top plate and decrement $A[i][p_i].\text{cnt} \leftarrow c - 1$. At this point, either the stack is empty and $A[i][p_i].\text{cnt} = 0$, so by 2(b) we push an fp_z-plate and set $A[i][p_i].\text{cnt} \leftarrow 1$ and $A[i][p_i].\text{fp} \leftarrow \text{fp}_z$; or the stack is not empty, and we take no further action. In the first case, the stack contains a single plate labeled with $A[i][p_i].\text{fp}$ and the counter is 1; in the second, by assumption all plates on the stack are still labeled with $A[i][p_i].\text{fp}$, and $A[i][p_i].\text{cnt}$ still gives the number of plates on the stack.

Having shown the invariant of the stack, we make the following observation. Let $\tilde{n} = \sum_{y \in V_x^i} n_y$. Then $M[i][p_i] = n_x + \tilde{n}$. By the statement of the lemma $i \in \hat{I}_x$, implying that $A[i][p_i].\text{fp} \neq \text{fp}_x$. We claim that $A[i][p_i].\text{cnt} = c > 0$ implies $\tilde{n} - n_x \geq A[i][p_i].\text{cnt} - 1$. To see this, note that $A[i][p_i].\text{cnt} = c > 0$ means that there are c plates labeled with $A[i][p_i].\text{fp} \neq \text{fp}_x$ on the stack associated to c insertions of elements in V_x^i with fingerprint $A[i][p_i].\text{fp}$. If there ever were any fp_x-labeled plates on the stack (i.e., $n_x > 0$), they were subsequently popped off by insertions of elements with their fingerprints not equal to fp_x. On the other hand, if an insertion of x did not place a plate on to the stack, then it popped off a plate corresponding to an insertion of an element

in V_x^i. Thus, at most $\tilde{n} - n_x$ insertions of elements in V_x^i have never popped off a plate of x, or had their plate popped off by an insertion of x. For $\tilde{n} - n_x = 0$ we have that $A[i][p_i].\text{cnt} = 1$, and $\tilde{n} - n_x \geq A[i][p_i].\text{cnt} - 1$. Similarly, if $\tilde{n} - n_x = d > 0$ then $\tilde{n} - n_x \geq A[i][p_i].\text{cnt} - 1$ as there are still $A[i][p_i].\text{cnt}$ plates associated with insertions of elements in V_x^i that have never been popped off and at least $A[i][p_i].\text{cnt} - 1$ of them correspond to insertions not popping off a plate of x.

We conclude that $M[i][p_i] = n_x + \tilde{n} \geq n_x + (n_x + A[i][p_i].\text{cnt} - 1) = 2n_x + A[i][p_i].\text{cnt} - 1$. Or, by rearranging,

$$n_x \leq \frac{M[i][p_i] - A[i][p_i].\text{cnt} + 1}{2}$$

which proves the lemma. □

As this lemma holds for every $i \in \hat{I}_x$, we conclude that
$n_x \leq \Theta_1 = \min_{i \in \hat{I}_x}\{\Theta_1^i\} \leq \min_{i \in \hat{I}_x}\{M[i][p_i]\}$.

Correcting $M[i][p_i]$ when x does "own" $A[i][p_i]$. Now, say that row $i \in I_x$. Under the NFC condition $A[i][p_i].\text{cnt}$ stores the number of occurrences of x that are *not* canceled out by occurrences of $y \in V_x^i$. So, we must have had at least $\sum_{y \in V_x^i} n_y \geq n_x - A[i][p_i].\text{cnt}$ occurrences of $y \in V_x^i$. This implies $M[i][p_i] \geq 2n_x - A[i][p_i].\text{cnt}$, and, by rearranging,
$n_x \leq \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2}$.

Lemma 3-2. *Let $\vec{S}$ satisfy the NFC condition, and let $x \in \mathcal{U}$. Then for any $i \in I_x$ we have*
$n_x \leq \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2} = \Theta_2^i$. ♦

Proof of Lemma 3-2. We can think of the counter $A[i][p_i].\text{cnt}$ as counting the depth of a stack of fingerprint-labeled plates as for the proof of Lemma 3-1. View an insertion of x being associated with either an insertion of $y \in V_x^i$ that pops off its fp_x-labelled plate from the stack or an insertion of $y \in V_x^i$ of the plate it pops off.

By the statement of the lemma $i \in I_x$, $A[i][p_i].\text{fp} = \text{fp}_x$ and under the NFC condition all plates on the stack are of x. Out of the insertions having plates on the stack, only the bottom plate one could

have popped off a plate of $y \in V_x^i$. Thus, at least $n_x - A[i][p_i].\text{cnt}$ insertions of x are associated with an (unique) insertion of $y \in V_x^i$ and

$$\tilde{n} \geq n_x - A[i][p_i].\text{cnt}.$$

From $M[i][p_i] = n_x + \tilde{n}$ we thus obtain $M[i][p_i] \geq 2n_x - A[i][p_i].\text{cnt}$ and

$$n_x \leq \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2}.$$

□

As this lemma holds for every $i \in I_x$, we conclude that $n_x \leq \Theta_2 = \min_{i \in I_x}\{\Theta_2^i\} \leq \min_{i \in I_x}\{M[i][p_i]\}$. Combined with the conclusion of Lemma 3-1, we have $n_x \leq \text{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\} \rfloor \leq \text{CMS}(x)$.

Precise estimation when some $|V_x^i| \in \{0, 1\}$. If there exists an i such that $\left|V_x^i\right| = 0$, then $M[i][p_i] = A[i][p_i] = n_x$. Hence, in this special case, both $\text{CMS}(x) = n_x$ and $\text{HK}(x) = n_x$. When this is not the case, $n_x < M[i][p_i]$ for all $i \in [k]$, so $n_x < \text{CMS}(x)$. For CK, on the other hand, if there exists a row i such that $|V_x^i| = 1$, we still have $\text{CK}(x) = n_x$. Our next result, which is a corollary of Lemmas 3-1 and 3-2, shows that either one of Θ_1^i or Θ_2^i is precisely n_x, or the smaller of the two is $n_x \pm 1/2$. Thus $\text{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\} \rfloor = n_x$.

Corollary 1. *Let $i \in [k]$ be such that $|V_x^i| = 1$. If the stream satisfies the NFC condition, then*

$$\begin{aligned} i \in \hat{I}_x \Rightarrow \quad & n_x = \frac{M[i][p_i] - A[i][p_i].\text{cnt}}{2} + c \text{ with } c \in \{1/2, 0\}, \\ i \in I_x \Rightarrow \quad & n_x = \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2} + c \text{ with } c \in \{-1/2, 0\}. \end{aligned}$$ ♦

Proof of corollary 1. We think of the counter $A[i][p_i].\text{cnt}$ as counting the depth of a stack of fingerprint-labeled plates as for the proof of Lemma 3-1 and associate occurrences of x and $y \in V_x^i$ in the similar way.

Moreover, $|V_x^i| = 1$ implies $M[i][p_i] = n_x + n_z$, or equivalently, $n_z = M[i][p_i] - n_x$ for $z \neq x$.

We start by focusing on the case $i \in \hat{I}_x$ ($A[i][p_i].\text{fp} \neq \text{fp}_z$). Say x at some point owned the counter. Then, the plate at the bottom of the stack (labeled with fp_z) corresponds to a occurrence of z that popped off a plate of x. So, only $A[i][p_i].\text{cnt} - 1$ occurrences of z are not associated with x implying $n_z = n_x + A[i][p_i].\text{cnt} - 1$. Hence, $M[i][p_i] - n_x = n_x + A[i][p_i].\text{cnt} - 1$, or equivalently, $n_x = \frac{M[i][p_i] - A[i][p_i].\text{cnt}+1}{2}$. Say x never owned the counter. Then, none of the occurrences of z with a plate on the stack popped an x-plate from the stack. This implies that $n_z = n_x + A[i][p_i].\text{cnt}$, and $n_x = \frac{M[i][p_i] - A[i][p_i].\text{cnt}}{2}$.

Let now $i \in I_x$ ($A[i][p_i].\text{fp} = \text{fp}_x$). Say x was the only owner of the counter. Then, none of the occurrences of x with a plate on the stack popped an z-plate from the stack. Thus, $n_x = n_z + A[i][p_i].\text{cnt}$ and, adding n_x to both sides and rearranging, $n_x = \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2}$. Say z at some point owned the counter. Then, the plate at the bottom of the stack (labeled with fp_x) corresponds to the occurrence of x that popped off a plate of z, and $n_x = n_z + A[i][p_i].\text{cnt} - 1$ and $n_x = \frac{M[i][p_i] + A[i][p_i].\text{cnt}-1}{2}$. □

Finally, we note one more case when $\text{CK}(x) = n_x$. If one of the x's buckets holds uninitialized fingerprint, i.e. $i \in [k]$ such that $A[i][p_i].\text{fp} = \star$, then $|\hat{n}_x - n_x| = 0$. This is because 1) the HK has the property that if x maps to a position in A with an uninitialized fingerprint, then x was never inserted (i.e., $n_x = 0$); and 2) we define CK to return $\hat{n}_x = 0$ if any of x's positions in A holds an uninitialized fingerprint.

3.5.3 Frequency estimate errors

In this section we extend the frequency estimation error analysis of CMS to CK. We have already seen that the CK estimate is never worse than the CMS estimate; in this section, we explore how much better it can be.

We begin with a simple theorem about the relationship between Θ_1 and the plain CMS estimate.

Theorem 3-1. *Fix an $x \in \mathcal{U}$, and let i^* be any row index such that* $\mathrm{CMS}(x) = M[i^*][p_{i^*}]$. *If* $i^* \in \hat{I}_x$ *then either* $\mathrm{CK}(x) = n_x$, *or* $\left(\Theta_1 \leq \frac{\mathrm{CMS}(x)}{2}\right)$. ♦

Proof. If any $A[i][p_i], i \in [k]$ has an uninitialized fingerprint, then $\mathrm{CK}(x) = n_x = 0$. Now assume this is not the case, so that $A[i][p_i].\mathrm{cnt} \geq 1$ for all the counters associated to x. By definition $\Theta_1 = \min_{i \in \hat{I}_x} \Theta_1^i \leq \Theta_1^{i^*}$, and so $\Theta_1 \leq \frac{M[i^*][p_{i^*}] - A[i^*][p_{i^*}].\mathrm{cnt}+1}{2} \leq \frac{\mathrm{CMS}(x)}{2}$. □

Next, a similar theorem relating Θ_2, the plain CMS estimate, and the HK estimate (when $d = 1$).

Theorem 3-2. *Fix an $x \in \mathcal{U}$, and let i^* be any row index such that* $\mathrm{CMS}(x) = M[i^*][p_{i^*}]$. *If* $i^* \in I_x$ *then either* $\mathrm{CK}(x) = n_x$ *or* $\left(\Theta_2 \leq \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}\right)$. ♦

Proof. If any $A[i][p_i], i \in [k]$ has an uninitialized fingerprint, then $\mathrm{CK}(x) = n_x = 0$. Now assume this is not the case, so $A[i^*][p_i^*].\mathrm{cnt} \leq \max_{i \in \mathcal{I}_x} A[i][p_i] = \mathrm{HK}(x)$. We have,
$\Theta_2 = \min_{i \in I_x} \Theta_2^i \leq \Theta_2^{i^*} = \frac{M[i^*][p_i^*] + A[i^*][p_i^*].\mathrm{cnt}}{2} \leq \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$. □

Now, if $\mathrm{CK}(x)$ is determined by line 10 of Figure 3-10, then $\mathrm{CK}(x) = \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$. On the other hand, if $\mathrm{CK}(x)$ is determined by line 15, then $\mathrm{CK}(x) = 0 \leq \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$. If neither of these holds, $\mathrm{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\} \rfloor$. Thus, Theorem 3-1 and 3-2 imply $\lfloor \min\{\Theta_1, \Theta_2\} \rfloor \leq \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$, giving us the following lemma.

Lemma 3-3. *For any $x \in \mathcal{U}$,* $\mathrm{CK}(x) \leq \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$. ♦

From here, it is straightforward to bound the CK estimation error, giving us the main result of this section.

Corollary 2. *Let $x \in \mathcal{U}$. If the stream satifies the NFC condition, then* $\mathrm{CK}(x) - n_x \leq \frac{\mathrm{CMS}(x)-\mathrm{HK}(x)}{2}$. ♦

Proof of corollary 2. The NFC condition gives $\text{CK}(x) \geq n_x \geq \text{HK}(x)$, and $\text{CK}(x) - n_x \leq \text{CK}(x) - \text{HK}(x)$. So, by Lemma 3-3 we arrive at

$$\begin{aligned}
\text{CK}(x) - n_x &\leq \left(\frac{\text{CMS}(x) + \text{HK}(x)}{2}\right) - n_x \\
&\leq \left(\frac{\text{CMS}(x) + \text{HK}(x)}{2}\right) - \text{HK}(x) \\
&\leq \frac{\text{CMS}(x) - \text{HK}(x)}{2}.
\end{aligned}$$

□

Consequences of Corollary 2. First, as $\text{CMS}(x)$ and $\text{HK}(x)$ approach each other — even if both are large numbers (e.g. when the stream is long and x is relatively frequent) — the error in $\text{CK}(x)$ approaches *zero*.

Next, because CMS is a count-all structure, the *worst case* guarantee is that the error $\text{CK}(x) - n_x \leq \text{CMS}(x)/2$, i.e., when $\text{HK}(x) = 0$. This occurs iff x does not own any of its counters, which implies that x is not the majority element in *any* of the substreams observed by the positions $A[i][p_i]$.cnt to which x maps. As $M[i][p_i]$ observes the same substream as $A[i][p_i]$, and $\text{CMS}(x) = \min_{i \in [k]}\{M[i][p_i]\}$, for practical values of k, m it is unlikely that all k of the V_x^i have unexpectedly large numbers of elements. Moreover, for typical distributions seen in practice (e.g., power-law distributions that have few true heavy elements), it is even less likely that *all* of the V_x^i contain a heavy hitter. Thus under "honest" conditions, we do not expect $\text{CMS}(x)$ be very large when $\text{HK}(x)$ is very small.

This last observation surfaces something that CK can provide, and neither CMS nor HK can: the ability to signal when the incoming stream is atypical. We explore this in detail in Section 3.5.6.

3.5.4 Experimental Results

We will now compare non-adversarial performance of the compact frequency estimators (CFEs) by measuring the ability of these structures in identifying the most frequent (heavy) elements of a

stream. Finding the heavy elements of a stream is the typical use case of CFEs and as such these structures are used for that purpose in many systems level applications [17, 26, 27, 28, 4, 29, 30]. The ability to accurately identify these heavy elements is based on a CFE's ability to accurately make frequency estimations on these heavy elements, while maintaining the ability to make accurate frequency estimations on the non-heavy elements, such that one would be able to distinguish between the two classes of elements. Therefore, we experimentally measure the non-adversarial performance of these structure by comparing a number of performance metrics in identifying heavy elements across three different streams.

Data Streams. We have three different streams we experiment with. We sourced two streams from a frequent item mining dataset repository[5]. We also sourced an additional stream by processing a large English language novel from Project Gutenberg.

We summarize each of these three streams and why they are of particular interest to experiment on below.

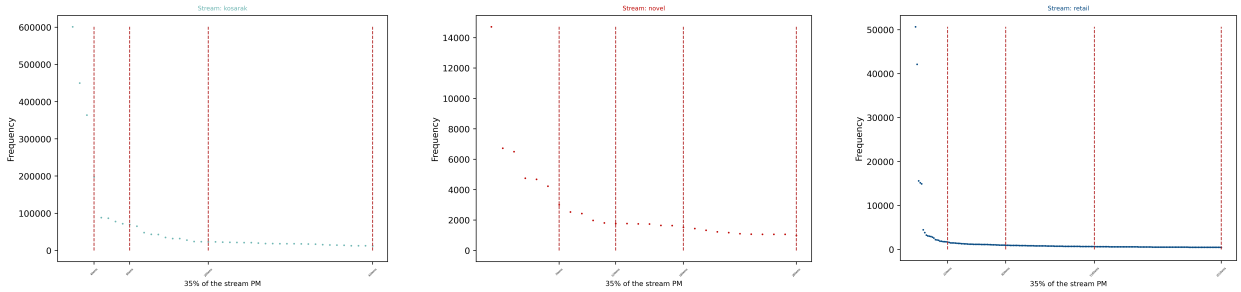
1. **Kosarak Stream:** This data collection contained anonymized click-rate data collected from visits to an online Hungarian news site. The resultant stream is of total length $8,019,015$ with $41,270$ distinct elements. As aforementioned, we sourced this stream from a frequent item mining dataset repository which is a collection of data sets meant to test frequent item finding algorithms on – the very task which we are doing. We flattened the raw collection of data such that it would resemble a stream that could be processed item-by-item.

2. **Novel Stream:** We created a stream by processing the individual words sequentially of The Project Gutenberg eBook plaintext edition of the 1851 English-language novel *Moby-Dick; or, The Whale* by Herman Melville (ignoring capitalization and non-alphabetical characters) [31]. Long bodies of natural language obey an approximate Zipf distribution as the frequency of any word is inversely proportional to its rank in an ordered frequency

[5] http://fimi.uantwerpen.be/data/

list [32]. It is of interest to measure compact frequency estimators performance against data following a Zipf distribution [29, 33, 17, 27, 28, 30]. The stream is of total length $2,174,111$ with $19,215$ distinct elements.

3. **Retail Stream:** This data collection contained anonymized shopping data from a Belgian retail store. The resultant stream is of total length $908,576$ and contains $16,740$ distinct elements. This data set is also from the frequent items mining dataset repository. As with the Kosarak stream we flattened the raw data such that it would resemble a stream after processing.



Kosarak stream Novel stream Retail stream

Figure 3-11. We plot the top 35% probability mass for each stream. That is the most frequent elements that make up 35% of the total weight of the stream (i.e. the fewest number of elements in each stream whose frequencies sum to such that when divided by the total length of the stream equal 35%). The first vertical red line in each plot is the top 20% probability mass, the second the top 25%, the third the top 30%, and the last the top 35%. From visual inspection we decided to make the top-K cut-off at, 20 for Kosarak stream, 22 for the Novel stream, and 22 for the Retail stream.

Measures and Metrics. We want to measure the performance of the CFEs of interest in the non-adversarial setting by determining how well they are able to identify and characterize the heavy elements in the streams above.

This problem, with varying but related definitions, is referred to in the literature as the heavy-hitters problem, the hot-items problem, or the top-K problem.

The simplest of these definitions to apply is that of the top-K problem, which is to simply report the set of elements with the K highest frequencies (for some K) for a given stream. That is given

elements of a stream $\vec{S} \subseteq \{e_1, e_2, \ldots, e_M\}$ with associated frequencies $(n_{e_1}, n_{e_2}, \ldots, n_{e_M})$ we can order the elements $\{e_1^*, e_2^*, \ldots, e_M^*\}$ such that $(n_{e_1}^* \geq n_{e_2}^* \geq \ldots \geq n_{e_M}^*)$. Then for some $K \in \mathbb{Z}^+$ we output the set of elements $\{e_1^*, e_2^*, \ldots, e_K^*\}$ with the K highest frequencies $(n_{e_1}^* \geq n_{e_2}^* \geq \ldots \geq n_{e_K}^*)$.

The top-K problem can be solved exactly given space linear to that of the stream by keeping an individual counter for each distinct element in the stream. It is not possible to solve exactly with space less than linear (see [34] for a formal impossibility argument), but it is a common technique to place a small data structure such as a min-heap restricted to size K on top of a CFE and by updating this small structure on each insertion once, one is able to approximate this top-K set [17, 26, 28].

For our purposes we simply compute the approximate top-K by processing the stream with a compact frequency estimator, querying on every distinct element in the stream, and ordering elements by approximated frequency. Likewise, we compute a true top-K for each stream by processing said stream with a map linear in the size of the stream, computing a frequency for each element, and ordering by true frequency. We note that we would have achieved identical results by putting a min-heap on top of each structure with fixed sized K, updating as described in [17] and outputting its contents once the entire stream has been processed. However, for experimental purposes our approach is more extensible than the one that would be used in practice.

The number of heavy elements, or perhaps the number of heavy elements one would care about, varies depending on the stream and the application. For instance, it is noted that in a telecommunications scenario when monitoring the top outgoing call destinations of a customer typically a value of K in the range of $10 - 20$ is appropriate [35]. Moreover, when identifying the most frequent elements of interest of Zipfian distribution it is often of interest to vary K based on the parameters of the underlying distribution [29].

We select K for each stream by observing the number of clearly identifiable outliers in the underlying stream. We do this by visually inspecting the selected streams' frequency plots. We set the x-axis to enumerate all distinct elements in a stream, ordered from most to least frequent and the y-axis as those distinct elements' corresponding frequencies. We make a cut-off around the point where the frequencies went from very peaked (distinct with prominent frequency jumps from element to element) to flat (many elements with about the same frequency – the point at which the frequency differences decline less sharply). These frequency plots can be seen in Figure 3-11. We set $K = 20$ for the Kosarak stream, $K = 22$ for the novel stream, and $K = 22$ for the retail stream.

We measure the accuracy of the non-adversarial performance according to four different metrics.

1. **Set Intersection Size (SIS):** This measures the size of the set intersection of the true top-K set $\mathcal{K}$ of the stream and the estimated top-K set $\tilde{\mathcal{K}}$ as reported by the CFE: SIS = $|\mathcal{K} \cap \tilde{\mathcal{K}}|$. This is measure of precision on the estimated top-K set as compared to the true top-K set. A SIS of K would imply perfect precision.

2. **Jaccard Index (JI):** The JI is a statistic that measures the similarity of two sets [36]. We use the statistic to determine the similarity of the true top-K set $\mathcal{K}$ of the stream and the estimated top-K set $\tilde{\mathcal{K}}$ as reported by the CFE. It is defined as JI = $\frac{|\mathcal{K} \cap \tilde{\mathcal{K}}|}{|\mathcal{K} \cup \tilde{\mathcal{K}}|}$. A JI can be in the range $[0, 1]$, with a JI of 1 implying a perfect characterization of the true top-K set by the CFE in its top-K estimation.

3. **Minimal Top-$\tilde{K}$ to Capture True Top-K (MCT):** This measures determines the minimal size $L \geq K$ the estimated top-k set $\tilde{\mathcal{K}}$ would need to be to capture all elements contained in the true top-K set $\mathcal{K}$. That is if one were to order the frequency estimates of all items made by a particular CFE, we would determine the number of items one would need to examine (starting from the most-frequent going down to the least-frequent) until all the elements from $\mathcal{K}$ were contained in that ordered set. Thus, $L - K$ indicates the number of elements

that fall out of $\mathcal{K}$ that are incorrectly being individually estimated to be greater than at least one element that is truly in $\mathcal{K}$.

4. **Average Relative Error on Top-k elements (ARE)**: Average Relative Error is a standard measure to use when comparing CFEs [17]. It is defined as $\text{ARE} = \frac{1}{K}\sum_{i=1}^{K}\frac{|\hat{n_i}-n_i|}{n_i}$ where $i \in [K]$ indexes the true top-K elements for a particular stream.

Results. We crafted reference implementations for all three CFEs of interest: CMS, HK, and CK[6]. They are implemented in Python3 and use the BLAKE2b cryptographic hash function for independent row hash functions and for a fingerprint hash function in the case of CK and HK.

We are interested in comparing performance when the space used by the structures is held constant. Observe that CK is three times as large as CMS, and HK is twice as large as CMS assuming the same space is used for a counter bucket and a fingerprint bucket (in the CK and HK) across all structures. In practice these buckets could be (say) 32-bits. We picked two sets of parameters, a *standard* set and a *constrained* set to test.

The standard set of parameters set $m = 2048, k = 4$ for CMS, $m = 1024, k = 4$ for HK, and $m = 910, k = 3$ for CK. This corresponds to 32.76 kB of space when using a 32-bit bucket sizes. We experimentally show that at this size all the structures are able to identify the heavy elements of the streams we test upon with minimal to no error.

The constrained set of parameters sets $m = 512, k = 4$ for CMS, $m = 256, k = 4$ for HK, and $m = 341, k = 2$ for CK. This corresponds to just 8.19 kB of space when using a 32-bit counter and fingerprint bucket sizes. In this space constrained setting the structures are still able to identify the heavy elements of the streams we test upon, but with some degree of moderate error.

For HK, we set $d = 0.9$ for all experiments, as this is the default chosen by Redis [16] and satisfies the desired properties of the exponential decay function stated in [17].

[6]Source code is available at: https://github.com/smarky7CD/cfe-in-adv-envs

We ran 1000 trials for each structure, stream, and parameter triplet using our reference implementations. We randomize each trial on the particular choice of hash functions used for the rows (by selecting a random per-trial seed), as well as the order in which the items in the stream are processed. The latter simulates an item being randomly drawn from the underlying distribution of the stream. We averaged our four metrics for each structure, stream and parameter triplet over the 1000 trials.

Structure	**Parameters (m,k)**	**Stream**	**SIS**	**JI**	**MCT**	**ARE**
Standard						
CK	(910,3)	Kosarak ($K = 20$)	20	1	20	≈ 0
		Novel ($K = 22$)	22	1	22	≈ 0
		Retail ($K = 22$)	22	1	22	≈ 0
CMS	(2048,4)		19.303	0.934	20.901	0.017
			22.999	0.999	22.001	0.009
			21.643	0.997	22.405	0.040
HK	(1024,4)		20	1	20	≈ 0
			22	1	22	≈ 0
			22	1	22	≈ 0
Constrained						
CK	(341,2)	Kosarak ($K = 20$)	17.189	0.757	28.695	≈ 0
		Novel ($K = 22$)	21.617	0.967	22.451	≈ 0
		Retail ($K = 22$)	13.442	0.441	209.439	0.021
CMS	(512,4)		18.241	0.841	24.567	0.125
			21.638	0.969	22.473	0.062
			18.745	0.745	41.609	0.296
HK	(256,4)		20	1	20	≈ 0
			22	1	22	0.001
			21.976	0.998	55.008	0.005

Table 3-1. A summary of non-adversarial setting results between the CK, CMS, and HK compact frequency estimators.

We present a summary of the results in Table 3-1. For the *standard* parameter set we see that CK and HK perform best, being able to perfectly capture the true top-K set for each stream with their outputted estimated top-K set in *every* trial. This is indicated by the SIS and MCT being equal to K and the JI being equal to 1 for each stream. Moreover, the estimates on these top-K elements for both of these structures were very tight. The ARE over all trials and streams was 0 (ignoring a

small rounding error). This indicates that CK and HK nearly perfectly individually estimated every single element in the true top-K across all trials.

CMS with the standard parameter sizing performs almost as well. Only failing to capture the true top-K set with its estimated set a few number of times over the 1000 trials. This is indicated by the SIS and MCT being very close to K and the JI being very close to 1 for each stream. However, CMS, as it is prone to overestimation on every element, has slightly higher ARE than the other structures.

The *constrained* set of parameters presents a challenge for all the CFEs in computing individual frequency estimations on elements in the streams, and as a result computing an accurate estimated top-K. This setting only allocates CK a measly 642 individual counters to compactly represent streams that all have over $19,000$ distinct elements. Under these conditions, HK performs best according to our metrics. It perfectly captures the true top-K in both the Kosarak and Novel stream, while only failing to do so in a handful of trials with the Retail stream. Moreover, the ARE is small across all streams – comparatively less than CMS with the standard parameters. HK by design prioritizes providing accurate estimates on the most frequent elements, by way of its probabilistic decay mechanism. So while it performs well on this task, it severely underestimates middling and low frequency elements at this sizing, reporting an individual frequency estimate very near 0 for any element that is not heavy.

CMS and CK perform less well in this small space allocation setting. While CMS performs slightly better in capturing the true top-K set within its estimated top-K set, CK continues to give better accuracy on individual point estimations of the true top-K elements across streams due to its internal sub-estimators that provide tighter estimations than CMS.

We observe in this constrained space setting across the structures measured performance is the worst on the Retail stream. This is because the Retail stream has a flatter distribution as compared to the other streams. That is to say, it has very few clearly identifiable heavy elements before

containing a large collection of elements of about the same frequency. This can be seen in the frequency plot in Figure 3-11. The Retail element with frequency rank 22 has a true frequency of 1715 while the Retail element of frequency rank 56 has a true frequency of 1005. Comparing this to $n_{22} = 22631, n_{56} = 9559$ and $n_{22} = 1176, n_{56} = 474$, respectively for the Kosarak and Novel stream, one can see that the relative fall off in true frequency is far less pronounced within this region of the Retail stream. This in turn leads to small errors in the individual frequency estimations of elements near (but outside) the true top-K of the Retail stream propagating to the top-K estimation – by making it challenging for the CFEs to draw a clear distinction between the truly heavy elements and the nearly heavy elements. The upshot being, one needs larger structures to accurately estimate these flatter streams.

In sum, CK performs comparatively well to both CMS and HK in this particular task. In fact, CK performed better than CMS when not burdened with *very* tiny space constraints. It is able to perfectly estimate the true top-K for all streams over all trials with only 2730 individual counters in the standard parameter setting, while also being adversarial robust where the others structures are not.

3.5.5 Attacks Against the CK

Our attacks against CK are almost one-to-one with those we present against the CMS with one major difference. Recall from Corollary 1 that if at least one counter in some row i of the element x we are querying on maps to has $|V_x^i| \leq 1$ then CK returns estimate $\hat{n}_x$ such that $\hat{n}_x = n_x$, i.e. CK(x) is a perfect estimate of x. This implies that for an error to exist in a frequency estimation of x it must be that $\forall i \in [k]$ it is necessary that $|V_x^i| \geq 2$. In the attack setting this means we need to find a 2-cover (specifically a $(\mathcal{FP}_x, x, 2)$-cover) on x to create error.

A 2-cover C for x contains elements $\{y_1, y_2, \ldots, y_t\}$ such that for every counter x maps to in positions $(p_1, p_2, \ldots, p_k) \leftarrow R(K, x)$ it is such that at least two distinct elements in C cover each counter. In our attack model we assume an initially empty representation and we never insert x in

any of our attacks (except for once to discover its counter positions in the public representation, private hash setting).

We attack CK in a two-step process, as with CMS and HK. We first find a 2-cover for our target element x and then repeatedly insert the 2-cover to create error. Under the assumption that x does not own any of its counters in the A substructure of the CK (which is guaranteed in our attack model[7]), then the Θ_1 sub-estimator will be used to make the final error evaluation **Qry** query on x. Say that after some process of finding a 2-cover for x (which will be of size $\leq 2k$ – for this discussion we will assume the size of the 2-cover is exactly $2k$) we have ω insertions to repeatedly insert the elements in the cover. Repeated and equal insertions of each of the elements in the 2-cover for x will cause the values in all of x's counters in the M substructure of the CK to be of value $\frac{\omega}{k}$. In the A substructure the value in the counters that x maps to will have value 1 and be owned by some element in the 2-cover. This is because (under the no-fingerprint collision assumption) in the initially empty structure, ownership of said counters will flip-flop on each iteration of the insertions of the 2-cover between the two distinct elements that map to these counters in accordance with the Up algorithm of the HK with $d = 1$.

Then applying the estimation from Θ_1 we see that we will generate error on x equal to $\frac{\omega}{2k}$. If we hold k constant and assume that we are attacking a CMS under the same conditions (we have found a 1-cover for a target x through some process and have ω insertions to accrue error) we will have an error of $\frac{\omega}{k}$, which is twice that of the CK under the same conditions. Under the same assumptions for HK, in addition to the assumption that we have already locked-down the counters of the target with initial insertions of the cover in the structure, we will achieve an error on the target of ω – which is $\omega - \frac{\omega}{2k}$ greater than that of the CK. We will see this pattern holds when giving concrete experimental attack error results at the conclusion of this section.

Public hash and representation setting. As our other attacks (for CMS and HK) in this setting, the CK attack (Figure 3-12) can be viewed as a two-step process. In this setting, we find a 2-cover

[7] Save for the trivial case in the public representation, private hash setting when no cover is able to be found.

$\text{CoverAttack}^{\textbf{Hash},\textbf{Up},\textbf{Qry}}(x, K, \text{repr})$

1 : cover $\leftarrow$ $\text{FindCover}^{\textbf{Hash}}(2, x, K)$
2 : **until** q_U **Up**-queries made:
3 : **for** $e \in$ cover: **Up**(e)
4 : **return** done

$\text{FindCover}^{\textbf{Hash}}(r, x, K)$

1 : cover $\leftarrow \emptyset$; found $\leftarrow$ False
2 : $\mathcal{I} \leftarrow \emptyset$; tracker $\leftarrow$ zeros(k)
3 : // $R(K, x)[i] = \textbf{Hash}(\langle i, K, x \rangle)$
4 : $(p_1, p_2, \ldots, p_k) \leftarrow R(K, x)$
5 : **while** not found
6 : **if** q_H **Hash**-queries made
7 : **return** $\emptyset$
8 : $y \twoheadleftarrow \mathcal{U} \setminus (\mathcal{I} \cup \{x\})$
9 : $\mathcal{I} \leftarrow \mathcal{I} \cup \{y\}$
10 : $(q_1, q_2, \ldots, q_k) \leftarrow R(K, y)$
11 : **for** $i \in [k]$
12 : **if** $p_i = q_i$ **and** tracker$[i] < r$
13 : cover $\leftarrow$ cover $\cup \{y\}$
14 : tracker$[i]$ + = 1
15 : **if** sum(tracker) $= rk$
16 : found $\leftarrow$ True
17 : **return** cover

Figure 3-12. Cover Set Attack for the CK in public hash function setting. The attack is parametrized with the update and **Hash** query budget q_U and q_H.

for target x using the **Hash** oracle only, and then accumulate error for the target by repeatedly inserting the 2-cover. Each insertion of the 2-cover increases the error by one. The two cover can be inserted at least $\frac{q_U}{2k}$ as the size of the cover is $\leq 2k$. We apply the same analysis used for the CMS attack, but replace $k(1 + L^1)$ with $k(1 + L^2)$ as the number of **Hash**-queries to complete the cover-finding step, as again, we now find a 2-cover. Assuming $q_U > 2k$ (so that any found C can be inserted at least once) we arrive at $\mathbb{E}[\text{Err}] \geq \left\lfloor \frac{q_U}{2k} \right\rfloor \Pr\left[L^2 \leq \frac{q_H - k}{k}\right]$ Using results from Section 3.4.1 we can further obtain a concrete expression for $\Pr\left[L^2 \leq \frac{q_H - k}{k}\right]$.

Private hash and representation setting.

Our CK attack for the setting (Figure 3-13) is essentially the same as the CMS attack, except a 2-cover (as opposed to a 1-cover) is detected and repeatedly inserted to build up the error. Using analysis similar to the CMS case and assuming q_Q is not the limiting factor,

$\text{CoverAttack}^{\mathbf{Up},\mathbf{Qry}}(x, \perp, \perp)$

```
1 : cover ← FindCover^{Up,Qry}(x)
2 : until q_U Up-queries made:
3 :     for e ∈ cover: Up(e)
4 : return done
```

$\text{MinUncover}^{\mathbf{Up},\mathbf{Qry}}(x, a', \text{cover})$

```
1 : b' ← a' − 1
2 : while a' ≠ b'
3 :     if (q_U − |cover| + 1)Up-
4 :         or q_Q Qry-queries made:
5 :             return cover
6 :     b' ← a'
7 :     for y ∈ cover : Up(y)
8 :     a' ← Qry(x)
9 : return a'
```

$\text{FindCover}^{\mathbf{Up},\mathbf{Qry}}(x)$

```
1 : / find 2- cover for x
2 : cover ← ∅
3 : found ← False
4 : I ← ∅; a ← Qry(x)
5 : while not found
6 :     if q_U Up- or q_Q Qry-queries made
7 :         return cover
8 :     y ↞ U \ (I ∪ {x})
9 :     I ← I ∪ {y}
10 :    Up(y); a' ← Qry(x)
11 :    if a' ≠ a :
12 :        cover ← {y}
13 :        found ← True
14 : for i ∈ [2, 3, ..., 2 · k]
15 :    a ← MinUncover^{Up,Qry}(x, a', cover)
16 :    if a = cover : return cover
17 :    for y ∈ I / in order of insertion to I
18 :        if q_U Up- or q_Q Qry-queries made
19 :            return cover
20 :        Up(y); a' ← Qry(x)
21 :        if a' ≠ a :
22 :            cover ← cover ∪ {y}
23 :            I ← I \ {y}
24 :            break
25 : return cover / cover is inserted at least once
```

Figure 3-13. Cover Set Attack for the CK in private hash function and representation setting. The attack is parametrized with the update query and query query budget – q_U and q_Q.

$$\mathsf{Err} \geq \left\lfloor \left(\frac{\ell+1}{2} + \frac{1}{\ell}\left(q_U + \sum_{i=1}^{\ell-1} (\ell - i)\delta_i \right) - L^2 \right) \right\rfloor$$

with $\ell \leq 2k$ rounds to find a 2-cover. The error bound is similar to the one for the CMS attack, but with L^1 replaced with L^2 as now $|\vec{I}|$ is precisely L^2.

For reasonable sizes of the CK we mainly expect $\ell = 2k$ (for the CMS case we expected ℓ=k) and that $\mathbb{E}[\delta_1]$ are bounded by a constant that is small relative to $m, q_U/k$. Given that $k \ll m$, we expect the following to approximate $\mathbb{E}[\mathsf{Err}]$:

$$\mathbb{E}\left[\left\lfloor\left(\frac{2k+1}{2} + \frac{1}{2k}\left(q_U + \sum_{i=1}^{2k-1}(2k-i)\delta_i\right) - L^2\right)\right\rfloor\right] \approx \frac{q_u}{2k} - \mathbb{E}[L^2].$$

Public hash and private representation setting. As with the CMS, the attack and analysis from the public hash and representation setting applies.

Private hash and public representation setting. This attack (Figure 3-14) is one-to-one with the CMS attack in the same setting, but again we find 2-cover as opposed to a 1-cover. Hence, $\mathbb{E}[\mathsf{Err}] \geq \frac{q_U - 1 - \mathbb{E}[L^2]}{2k} \gtrapprox \frac{q_U - 1 - 2mH_k}{2k}$.

Attack Comparisons. We implemented our attacks against all structures in all settings to

	Public Hash Setting			**Private Hash, Private Rep Setting**		
Structure	$\lvert\text{cov}\rvert$	Exp. Err	$\mathbb{E}[\mathsf{Err}]$	$\lvert\text{cov}\rvert$	Exp. Err	$\mathbb{E}[\mathsf{Err}]$
CK, $(m = 682, k = 4)$	7.96	131821.00	131072.00	7.96	130796.69	127432.90
CMS, $(m = 2048, k = 4)$	3.99	263017.82	262144.00	3.99	261116.16	257877.34
HK, $(m = 1024, k = 4)$	3.99	1047502.69	1047500.00	4.0	1038804.55	1038018.54
CK, $(m = 1365, k = 8)$	15.97	65667.10	65536.00	15.93	63776.52	56618.28
CMS, $(m = 4096, k = 8)$	8.00	131072.00	131072.00	7.99	127029.66	119939.65
HK, $(m = 2048, k = 8)$	7.96	1046434.76	1046424.00	7.98	1007439.04	996946.87

Table 3-2. A comparison of Err accumulated by the different structures during attacks in the public hash setting and the private hash, private representation setting. We give the average size of the cover set and average error accumulated in each structure, setting pair over the 100 experiment trials. We also give the $\mathbb{E}[\mathsf{Err}]$ according to our analysis.

experimentally verify their correctness and our analysis. In Table 3-2 we present a summary of results for the public hash setting (our least restrictive setting) and the private hash, private representation setting (our most restrictive setting.). We experiment on two sets of parameters, one fixing $k = 4$ and the other $k = 8$. We then select a reasonable value of m for CMS and then half it for HK and third it for CK so that the same space is used in each structure. We fix

$\text{CoverAttack}^{\mathbf{Up},\mathbf{Qry}}(x, \perp, \mathsf{repr})$

```
cover ← FindCover^Up(2, x, repr)
until q_U Up-queries made:
    for e ∈ cover: Up(e)
return done
```

$\text{FindCover}^{\mathbf{Up}}(r, x, \mathsf{repr})$

```
⟨M, A⟩ ← repr
cover ← ∅; found ← False
I ← ∅; tracker ← zeros(k)
⟨M', A'⟩ ← Up(x)
/ compute x's indices
for i ∈ [k]
    for j ∈ [m]
        if M'[i][j] ≠ M[i][j]
            p_i ← j; break;
while not found
    if q_U Up-queries made : return ∅
    y ↞ U \ (I ∪ {x})
    I ← I ∪ {y}
    ⟨M, A⟩ ← ⟨M', A'⟩
    ⟨M', A'⟩ ← Up(y)
    / compute y's indices
    for i ∈ [k]
        for j ∈ [m]
            if M'[i][j] ≠ M[i][j]
                q_i ← j; break;
    for i ∈ [k]
        / compare x's and y's indices row by row
        if p_i = q_i and tracker[i] < r
            cover ← cover ∪ {y}
            tracker[i] + = 1
    if sum(tracker) = rk
        found ← True
return cover
```

Figure 3-14. Cover Set Attack for the CK in private hash function and public representation setting. The attack is parametrized with the update query budget q_U.

adversarial resources such that $q_H, q_U, q_Q = 2^{20}$. In practice this ensures that the number of **Hash** queries or **Qry** queries will not be the bottleneck in our attacks and that we are able to generate sufficient error in each attack to showcase overall trends. We run each attack setting and structure pairing over 100 trials, selecting a random target in each trial, and average the results.

Observe the pattern that when holding k constant and setting reasonable m values, adjusting such that CMS, CK, and HK use the same space, attacks against CK generate the least amount of error. The attacks against CK produce about half of the amount of error as opposed to the CMS attacks, and about $q_U - \frac{q_U}{2k}$ less the amount of error as opposed to the HK attacks. Moreover, observe that our analytical results closely match those of our experimental results.

3.5.6 Adversarial Robustness

Corollary 2 shows that the error in $\mathrm{CK}(x)$ is largest when $\mathrm{HK}(x) \ll \mathrm{CMS}(x)$. In particular, when x does not own any of its counters $\mathrm{HK}(x)$ takes on its minimal value of zero. But we can say something a bit more refined, by examining what is computed on the way to the returned value $\mathrm{CK}(x)$.

Specifically, recall that $\mathrm{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\} \rfloor$, where Θ_1 is the smallest upperbound on n_x that we can determine by looking only at the rows that x does not own, and Θ_2 is the smallest upperbound on n_x that we can determine by looking only at the rows that x does own. Let $\Delta = |\mathrm{CK}(x) - n_x|$ be the potential error in the estimate $\mathrm{CK}(x)$. Dropping the floor for brevity, if $\mathrm{CK}(x) = \Theta_1$ then Lemma 3-2 tells us that $\Delta \leq (M[i^*][p_{i^*}] - A[i^*][p_{i^*}].\mathrm{cnt} + 1)/2$, where $i^* \in \{j \mid \Theta_1^j = \min_{i \in \hat{I}_x}\{\Theta_1^i\}\}$.

Likewise, if $\mathrm{CK}(x) = \Theta_2$ then by Lemma 3-2 we have $n_x \leq (M[i^*][p_{i^*}] + A[i^*][p_{i^*}].\mathrm{cnt})/2$, where now $i^* \in \{j \mid \Theta_2^j = \min_{i \in I_x}\{\Theta_2^i\}\}$. In this case $A[i^*][p_{i^*}].\mathrm{cnt} \leq n_x$, so we know that $\Delta \leq (M[i^*][p_{i^*}] + A[i^*][p_{i^*}].\mathrm{cnt})/2 - A[i^*][p_{i^*}].\mathrm{cnt} = (M[i^*][p_{i^*}] - A[i^*][p_{i^*}].\mathrm{cnt})/2$. Adding 1/2 to this upperbound gives the same expression as in the previous case.

Thus, we can augment the basic version of CK so that $\textsc{Qry}(\mathsf{qry}_x)$ computes Δ, and returns a boolean value flag along with the estimate of n_x. The value of flag would be set to 1 iff $\Delta \geq \psi N$, where N is the length of currently inserted stream and ψ is a parameter. We choose this condition because the non-adaptive correctness guarantees of CMS have a similar form: with k rows and m counters per row, the estimate $\mathrm{CMS}(x)$ is such that $\Pr[\,\mathrm{CMS}(x) - n_x \leq \epsilon N\,] \geq 1 - \delta$ when $\epsilon = e/m$, $\delta = e^{-k}$.

Observe that when the frequency estimation error on an element x is large, then row i^* will be such that $M[i^*][p_{i^*}]$ will have a large value and $A[i^*][p_{i^*}]$.cnt will have a value very small relative to the value in $M[i^*][p_{i^*}]$. In the worst case $A[i][p_{i^*}]$.cnt = 1 – in our attacks we force this to be the case. Taking $A[i^*][p_{i^*}]$.cnt ≈ 0, observe that whether $\text{CK}(x)$ is determined by Θ_1 or Θ_2, we see $\text{CK}(x) \approx (1/2)M[i^*][p_{i^*}] \approx (1/2)\text{CMS}(x)$ in this high error case. Then rolling in the non-adaptive CMS correctness guarantee we see $\Pr[\Delta > (1/2)(\epsilon N) - (1/2)n_x] \leq \delta$ and certainly $\Pr[\Delta > 1/2(\epsilon)N] \leq \Pr[\Delta > 1/2(\epsilon)N - (1/2)n_x]$, thus setting $\psi = (1/2)\epsilon$ (where we can derive ϵ from parameter m) can be a useful starting point for setting ψ. As a caveat, however, as N becomes large, an adversarial stream may be able to induce significant error by setting ψ in this way (due to the looseness of the CMS bound). Depending on the deployment scenario, smaller values of ψ, or even sublinear functions of N, may be more appropriate for detecting abnormal streams.

Nonetheless, we implemented an version of CK with flag-raising (see Figure 3-15), and set $m = 1024$, $k = 4$. This corresponds to $\epsilon = 0.00265$, $\delta = 0.0183$. We then set $\psi = 0.0012 < \frac{1}{2}\epsilon$. Against it, we ran 100 trials of the public hash, public representation attack with $q_U = 2^{16}$, and with per-trial random target elements x. The average error was 8203.71, and in *every* trial the warning flag was raised on the frequency estimation of the target element.

For comparison, we also ran 100 trials, with the same parameters, using the non-adversarial streams from Section 3.5.4. In each trial, the entire stream was processed, and then we queried for the frequency of *every* element in the stream, counting the number of estimates that raised the flag. Over all 100 trials, or nearly 7.7 million estimates in total, only *three* flags were raised. These initial findings suggest that the potential for CK to flag suspicious estimates may be of significant benefit to systems employing compact frequency estimators.

$\textsc{Rep}_K(\mathcal{S})$

$M \leftarrow \text{zeros}(k, m)$
for $i \in [k]$
  $A[i] \leftarrow [(\star, 0)] \times m$
$\text{repr} \leftarrow \langle M, A \rangle$
for $x \in \mathcal{S}$
  $\text{repr} \twoheadleftarrow \textsc{Up}_K(\text{repr}, \text{up}_x)$
return repr

$\textsc{Up}_K(\text{repr}, \text{up}_x)$

$\langle M, A \rangle \leftarrow \text{repr}$
$M \twoheadleftarrow \textsc{Up}_K^{\text{CMS}}(M, \text{up}_x)$
$A \twoheadleftarrow \textsc{Up}_K^{\text{HK}}(A, \text{up}_x)$
return $\text{repr} \leftarrow \langle M, A \rangle$

$\textsc{Qry}_K(\text{repr}, \text{qry}_x)$

$\langle M, A \rangle \leftarrow \text{repr}$
$(p_1, \ldots, p_k) \leftarrow R(K, x)$, $\text{fp}_x \leftarrow T(K, x)$
$\Theta_1, \Theta_2, \Delta \leftarrow \infty$
$\text{flag} \leftarrow \text{False}$
$N \leftarrow \sum_{j=1}^{m} M[1][j]$
$\text{cnt}_{\text{UB},x} \leftarrow \textsc{Qry}\text{CMS}_K(M, \text{qry}_x)$
$\text{cnt}_{\text{LB},x} \leftarrow \textsc{Qry}_K^{\text{HK}}(A, \text{qry}_x)$
if $\text{cnt}_{\text{UB},x} = \text{cnt}_{\text{LB},x}$
  return $\text{cnt}_{\text{UB},x}$, flag
for $i \in [k]$
  if $A[i][p_i].\text{fp} = \star$
    $\text{cnt}_{\text{UB},x} \leftarrow 0$
    return 0, flag
  else if $A[i][p_i].\text{fp} \neq \text{fp}_x$
    $\Theta \leftarrow \frac{M[i][p_i] - A[i][p_i].\text{cnt}+1}{2}$
    $\Theta_1 \leftarrow \min\{\Theta_1, \Theta\}$
    $\hat{\Delta} \leftarrow \frac{M[i][p_i] - A[i][p_i].\text{cnt}+1}{2}$
    $\Delta \leftarrow \min\{\Delta, \hat{\Delta}\}$
  else if $A[i][p_i].\text{fp} = \text{fp}_x$
    $\Theta \leftarrow \frac{M[i][p_i] + A[i][p_i].\text{cnt}}{2}$
    $\Theta_2 \leftarrow \min\{\Theta_2, \Theta\}$
    $\hat{\Delta} \leftarrow \frac{M[i][p_i] - A[i][p_i].\text{cnt}}{2}$
    $\Delta \leftarrow \min\{\Delta, \hat{\Delta}\}$
$\text{cnt}_{\text{UB},x} \leftarrow \lfloor \min\{\Theta_1, \Theta_2\} \rfloor$
if $\Delta \geq \psi N$
  $\text{flag} \leftarrow \text{True}$
return $\text{cnt}_{\text{UB},x}$, flag

Figure 3-15. Keyed structure $\text{CK}[R, T, m, k, \psi]$ supporting point-queries for any potential stream element x (qry_x) and the ability to raise a flag on "bad" frequency estimation. $\textsc{Qry}_K^{\text{CMS}}$, $\textsc{Up}_K^{\text{CMS}}$, resp. $\textsc{Qry}_K^{\text{HK}}$, $\textsc{Up}_K^{\text{HK}}$, denote query and update algorithms of keyed structure $\text{CMS}[R, T, m, k]$ (Figure 3-2), resp. $\text{HK}[R, T, m, k, 1]$ (Figure 3-3). The parameters are a function $R : \mathcal{K} \times \{0,1\}^* \to [m]^k$, a function $T : \mathcal{K} \times \{0,1\}^* \to \{0,1\}^n$ for some desired fingerprint length n, integers $m, k \geq 0$, and flag parameter $\psi \in (0, 1)$. A concrete scheme is given by a particular choice of parameters.

CHAPTER 4
PROBABILISTIC DATA STRUCTURES IN THE WILD: A SECURITY ANALYSIS OF REDIS

Probabilistic Data Structures (PDS) are becoming ubiquitous in modern computing applications that deal with large amounts of data, especially when the data is presented as a stream. Their key property in this setting is that they provide approximate answers to queries on data without needing to store all the data. For example, a user may wish to estimate the cardinality of a datastream (in which case the HyperLogLog cardinality estimator could be used), find the most frequent elements in the stream (in which case a so-called top-K PDS is available), or just ask whether a particular data item has been seen before in the stream (where Bloom and Cuckoo filters are the tool for the job). Many modern data warehousing and processing systems provide access to PDS as part of their functionality.

A prominent example of such a system is Redis, a general purpose, in-memory database. Redis is integrated into general data analytics and computing platforms offered by AWS, Google Cloud, IBM Cloud, and Microsoft Azure, amongst others. Redis supports a variety of PDS: HyperLogLog (HLL), Bloom filter, Cuckoo filter, t-digest, Top-K, and count-min sketch [37]. While Redis was mostly used as a cache in the past, it is now a fully general system, used by a companies like Adobe [38], Microsoft [39], Facebook [40] and Verizon [41] for a variety of purposes. These include security-related applications, such as traffic analysis and intrusion detection systems [42].

As the functionality of Redis has broadened, so has its maturity with respect to security. Initially, the Redis developers stated that no security should be expected from Redis: *The Redis security model is: "it's totally insecure to let untrusted clients access the system, please protect it from the outside world yourself"* [43]. In reality, users failed to comply with this [44]. Today, Redis has a number of security features, and has adopted a different model, with a protected mode as default, user authentication, use of TLS, and command block-listing amongst other features [45]. Redis now also recognize security and performance in the face of adversarially-chosen inputs as being a

valid concern, stating that *"an attacker might insert data into Redis that triggers pathological (worst case) algorithm complexity on data structures implemented inside Redis internals"* and then going on to discuss two potential issues, namely hash table exhaustion and worst-case sorting behavior triggered by crafted inputs [45]. The first issue is prevented in Redis by using hash function seeding; the second issue is not currently addressed. However, Redis' consideration of malicious inputs does not seem to extend to their PDS implementations.

Given its prominence in the marketplace and the many other systems that rely on it, we contend that the PDS used in Redis are deserving of detailed analysis. Moreover, in view of the broad set of use cases for these PDS, including those where adversarial interference is anticipated and would be damaging if successful, this analysis should be done in an adversarial setting. This approach follows a line of recent work on PDS analysis [46, 5, 47, 48, 49, 50]. In this paper, we make a comprehensive security analysis of the suite of PDS provided by Redis, with a view to understanding how its constituent PDS perform in adversarial settings. As argued in [51], we regard the observation, documentation, and analysis of such security phenomena "in the wild" as constituting scientific contributions in their own right.

Following prior work, we assume only that the adversary has access to the functionality provided by the PDS (eg. via the presented API). The adversary's aim is then to subvert the main goal of the specific PDS under study. We deliberately remain agnostic about precisely which application is running on top of Redis, since the relevant applications will change over time and are anyway largely proprietary. The real-world effects of a successful attack will vary across applications, but might include, for example, false statistical information being presented to users (in the case of frequency estimation), wrongly reporting the presence of certain data items in a cache (in the case of Bloom filters or Cuckoo filters) leading to performance degradation, or the evasion of network attack detection (in the case of cardinality estimation being used in network applications). Instead of making application-specific analyses, we focus on the core PDS functionalities in Redis and how their goals can be subverted in general. Naturally, our analyses are specific to each of the

different PDS supported in Redis, and depend on various low-level implementation choices made by Redis. These choices lead us to develop novel attacks that are more powerful than the known generic attacks against the different PDS in Redis.

Since HLL in Redis was already comprehensively studied in [49], we do not consider it further here. We note only that [49] showed how to manipulate data input to Redis HLL to distort cardinality estimates in severe ways, in a variety of adversarial settings. The t-digest is a data structure first introduced in [52]; it uses a k-means clustering technique [53] to estimate percentiles over a collection of measurements. The structure is an outlier in the Redis PDS suite as it does not work in the streaming setting, but necessitates the batching of data in memory, and it is not really probabilistic in the same sense as the other PDS in Redis (in particular it does not employ the "hash functions mapping to array positions" paradigm that the other PDS in Redis use). For these reasons, we omit a security evaluation of t-digest (both in general and in the case of the Redis implementation).

This leads us to focus on the remaining four PDS in Redis: Bloom filter, Cuckoo filter, Top-K, and count-min sketch. For each PDS, we discuss how the PDS was originally described in the literature and lay out how the Redis implementation differs from this "theoretical" description. We then develop attacks for each of these four PDS, with the attacks in most cases exploiting specific features of the Redis implementations and being more efficient for this reason (simultaneously, we have to deal with the many oddities of the Redis codebase in our attacks). In total, we present 10 different attacks across the four PDS. We compare our attacks with known attacks for these PDS from the literature. We also look at how the PDS in Redis can be protected against attacks, drawing on existing literature that considers this question for PDS more generally [5, 54, 49, 50]. For the purposes of this dissertation, we provide the structural descriptions and attacks for the compact frequency estimators implemented in Redis (Top-K and count-min sketch). Full details on the Bloom filter and Cuckoo filter are available in the full paper [55].

Further, we notified Redis of our findings on 29.04.2024. The full version of our paper [55] is identical to the document we sent to Redis on 29.04.2024 aside from changes made in this subsection. We offered to engage in a coordinated approach to vulnerability disclosure and suggested a 90-day period before any public distribution of our research paper. Redis acknowledged our findings immediately and then gave a detailed response on 16.05.2024. In this response, Redis disputed the validity of analyzing Redis' PDS in adversarial settings; naturally we disagree with their viewpoint. However, they also committed to consider changes to their implementation in future versions, including using random seeds instead of fixed seeds, considering alternative hash functions, and adding disclaimers to their documentation. They did not commit to a timeline for this consideration. They decided not to handle our disclosure as a "Redis vulnerability".

4.1 PDS in Redis

We start by (re)introducing the PDS that we consider in this chapter – the count-min sketch and the Top-K (HeavyKeeper). We will describe their original specification, the probabilistic guarantees they provide, and give a detailed description of their Redis implementation.

4.1.1 Count-min Sketches

CMS.setup(pp)

1 : $\varepsilon, \delta \leftarrow pp$
2 : $m \leftarrow \left\lceil \frac{e}{\varepsilon} \right\rceil$
3 : $k \leftarrow \left\lceil \ln(\frac{1}{\delta}) \right\rceil$
4 : $h(\circ) \leftarrow \mathit{MurmurHash2}(\circ) \bmod m$
5 : $\sigma \leftarrow \text{zeros}(k, m)$
6 : **return** $\top$

CMS.ins(x, σ, v)

1 : $(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
2 : **for** $i \in [k]$
3 : $\quad \sigma[i][p_i] += v$
4 : **return** $\min_{i \in [k]}\{\sigma[i][p_i]\}$

CMS.qry(x, σ)

1 : $(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
2 : **return** $\min_{i \in [k]}\{\sigma[i][p_i]\}$

Figure 4-1. Redis count-min sketch algorithms. The analogous functions in the Redis API are: CMS.setup is CMS.INITBYPROB, CMS.ins is CMS.INCRBY, and CMS.qry is CMS.QUERY. We refer to a Redis count-min sketch initialized with $pp = \varepsilon, \delta$ as CMS$[\varepsilon, \delta]$.

A count-min sketch supports frequency estimates, i.e. estimates of the number of times a particular element occurs in a data set. Originally introduced in [4], a count-min sketch consists of a $k \times m$ array σ of (initially zero) counters, and k pairwise independent hash functions $h_1, ..., h_k$ that map between the universe $\mathcal{U}$ of data items and $[m]$.

An element x is added to a count-min sketch by computing $(p_1, p_2, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$, then adding 1 to each of the counters at $\sigma[i][p_i]$ for $i \in [k]$. This extends in the obvious way to insertions of v instances of an element at a time. A frequency estimate for x is computed as $\hat{n}_x = \min_{i \in [k]}\{\sigma[i][p_i]\}$. A count-min sketch may produce overestimates of the true frequency, but never underestimates.

For any $\varepsilon, \delta \geq 0$, any $x \in \mathcal{U}$, and any collection of data C stored by the count-min sketch (over $\mathcal{U}$) of length N, it can be guaranteed by appropriate setting of parameters that $\Pr[\hat{n}_x - n_x > \varepsilon N] \leq \delta$, where n_x is the true frequency of x. Specifically, we can take $m \leftarrow \lceil e/\varepsilon \rceil$, $k \leftarrow \lceil \ln(1/\delta) \rceil$. This correctness bound holds when the individual hash functions are sampled from a pairwise-independent hash family H (see [4] for a proof). It further assumes that insertions are done in the honest setting. That is, C and the queried element x are independent of the internal randomness of the structure (the random choice of the hash functions).

In Redis, a count-min sketch is initialized by the user calling CMS.setup(ε, δ). We will refer to the resulting sketch as CMS$[\varepsilon, \delta]$. The dimensions m, k of the count-min sketch are then calculated as above, and a $k \times m$ array of zeros is initialized. We note that it is also possible to initialize the structure from the dimensional parameters m, k, rather than deriving them from ε, δ. Insertions and membership queries on any element x are carried out in the same way as in the original structure, using the commands CMS.ins(x, σ, v) and CMS.qry(x, σ); both return the frequency estimate of x. The analogous functions in Redis are called CMS.INITBYPROB, CMS.INCRBY and CMS.QUERY, respectively.

To instantiate the k pairwise independent hash functions, Redis uses *MurmurHash2* with a per row seed equal to the row index, i.e. $h_1(x) \leftarrow h(x, 1), ..., h_k(x) \leftarrow h(x, k)$, where the syntax $h(x, i)$ means *MurmurHash2* evaluated on input x with seed i. For full details of count-min sketches in Redis, see Figure 4-1.

We point out that using fixed hash functions violates the honest setting assumptions that are required for the guarantees on frequency estimation errors in [4]. We will leverage this and the properties of *MurmurHash2* in our attacks to cause large frequency overestimates.

4.1.2 Top-K

A Top-K structure, originally introduced as the HeavyKeeper in [17], solves the *approximate* top-K problem.

The exact version of the problem is defined as follows: given elements of a data collection $C \subseteq \{e_1, e_2, ..., e_m\}$ with associated frequencies $(n_{e_1}, n_{e_2}, ..., n_{e_m})$, we can order the elements $\{e_1^*, e_2^*, ..., e_M^*\}$ such that $(n_{e_1}^* \geq n_{e_2}^* \geq ... \geq n_{e_M}^*)$. Then, for some $K \in \mathbb{Z}^+$, we output the set of elements $\{e_1^*, e_2^*, ..., e_K^*\}$ with the K largest frequencies $(n_{e_1}^* \geq n_{e_2}^* \geq ... \geq n_{e_K}^*)$. Given space linear in the stream this is trivial to solve exactly. However, by the pigeonhole principle, it is not possible to find an exact solution with space less than linear (see [34] for a formal impossibility argument). A common technique is to place a small data structure of size $O(K)$, like a heap or list, on top of a compact frequency estimator. By updating this small structure at most once upon an insertion of each element, we can approximate this top-K set [26, 28]. Using this technique we will obtain the K elements with the largest *estimated* frequencies.

The Top-K structure is represented by a $k \times m$ matrix σ. Each entry in σ is an (fp, cnt) pair, where fp is a fingerprint of the element that "owns" the counter, and cnt is said element's recorded count. These entry pairs are initialised to the distinguished symbol $\star$ and zero, respectively. Associated with each row is a hash function that maps elements in $\mathcal{U}$ to $[m]$, i.e. k hash functions $h_1, ..., h_k$. The fingerprint hash function h_{fp} maps elements in $\mathcal{U}$ to $\{0, 1\}^{\lambda_{fp}}$, for some desired fingerprint length λ_{fp}. Further, we initialise a min-heap H of maximal size K to store the elements

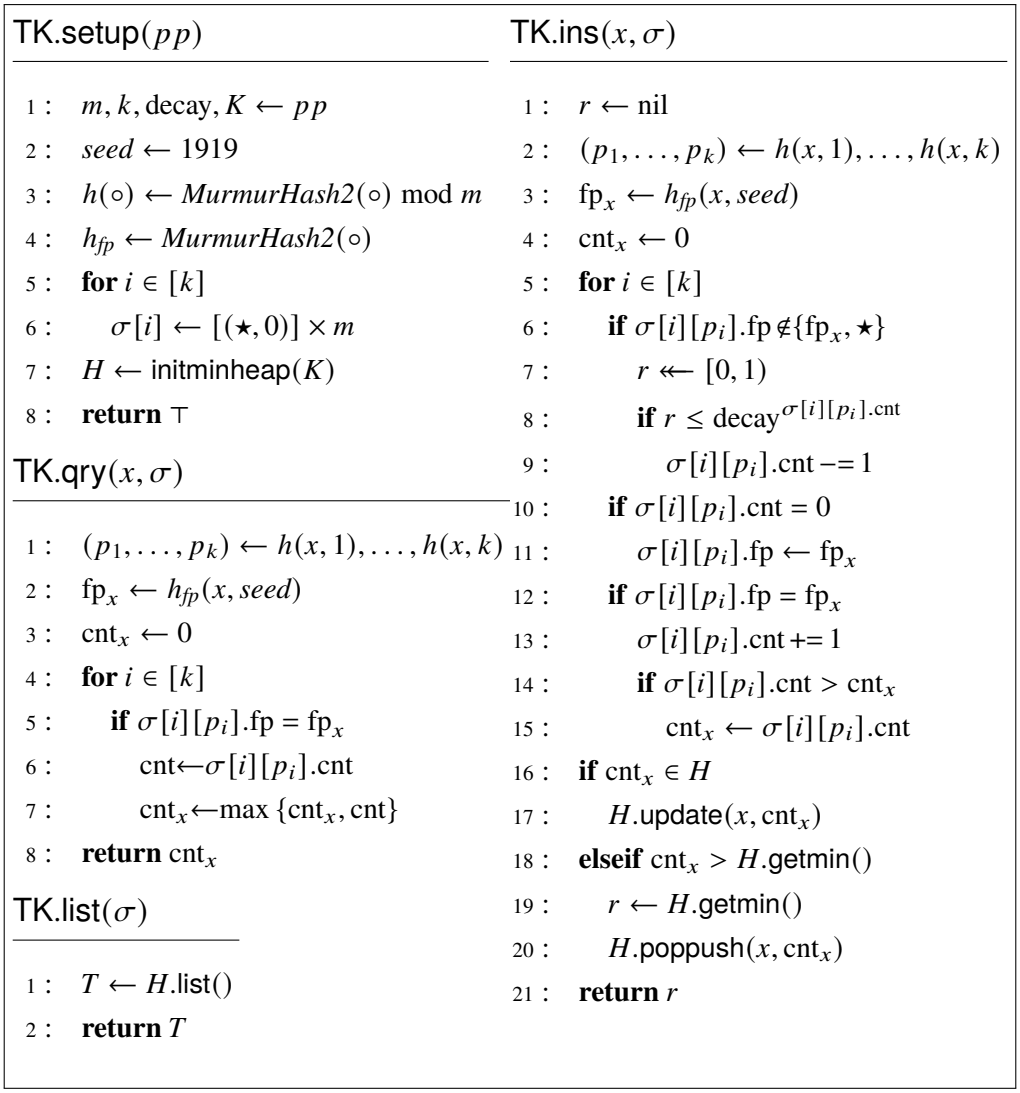


Figure 4-2. Redis Top-K structure algorithms. The analogous functions in the Redis API are: TK.setup is TOPK.RESERVE, TK.ins is TOPK.ADD, TK.qry is TOPK.COUNT, and TK.list is TOPK.LIST. We refer to a Redis Top-K structure initialised with $pp = m, k, \text{decay}, K$ as TK$[m, k, \text{decay}, K]$.

with the K largest estimated frequencies. Lastly, a decay value is set, which is used to decrement a counter when a specific condition is hit.

To insert an element x, we start by computing $(p_1, ..., p_k) \leftarrow (h_1(x), \ldots, h_k(x))$. We then compute the fingerprint fp_x associated with the element x as $h_{fp}(x)$. We also set a variable $\text{cnt}_x \leftarrow 0$. We then go row by row (indexed by $i \in [k]$), with the following cases:

1. **if** $\text{fp}^* = \star$, where fp^* is the current fingerprint value at matrix position (i, p_i), **then** we set the counter value to 1, the fingerprint to fp_x, and if $\text{cnt}_x < 1 : \text{cnt}_x \leftarrow 1$.

2. **else if** $\text{fp}_x = \text{fp}^*$, we add 1 to the counter value, and if $\text{cnt}_x < c : \text{cnt}_x \leftarrow c$, where c is the current counter value at matrix position (i, p_i).

3. **else** we select a random value $r \twoheadleftarrow [0, 1)$. If $r < \text{decay}^c$, where c is the current counter value at matrix position (i, p_i), we decrement the counter value stored at this position. If, after decrementing, this value is 0, we then set the counter value to 1, the fingerprint to fp_x, and if $\text{cnt}_x < 1 : \text{cnt}_x \leftarrow 1$. This is the so-called *probabilistic decay* process.

If, after this procedure, it is such that $x \in H$, we update the entry in the heap based on the current value of cnt_x. Else, we check that $\text{cnt}_x > H.\mathsf{min}$, and if so we remove the min entry in H and replace it with (x, cnt_x). This ensures that we are keeping an accurate account of the K highest estimated frequencies in H.

Top-K provides approximate answers to frequency queries for any element x, by computing $(p_1, \ldots, p_k) \leftarrow (h_1(x), \ldots, h_k(x))$ and $\text{fp}_x \leftarrow h_{fp}(x)$, and returning $\hat{n}_x = \max_{i \in [k]}\{\sigma[i][p_i]\}$ where $\sigma[i][p_i].\text{fp} = \text{fp}_x$. If none of the fingerprints in this set of buckets equals fp_x, then 0 is returned. Top-K returns the estimated top-K elements by returning all the pairs of items and estimated counts stored in H.

In [17], a probabilistic guarantee for estimation error magnitude is presented, assuming that each $\sigma[i][j]$ has a sole owner throughout the processing of the entire stream. However, the statement

lacks precision, and its proof is flawed, thus we will not restate it (see instead [50] for a meaningful result). Moreover, the results in [17] rely on a no-fingerprint collision (NFC) assumption, ensuring that all frequency estimates satisfy $\hat{n}_x \leq n_x$, where n_x is the true frequency of x, i.e. Top-K strictly underestimates frequencies. While not formally defined in the original paper, a rigorous definition is given in [50], characterizing NFC as the assumption that elements hashing to the same row position in any row do not share a fingerprint. This assumption is reasonable for practical sizes of $\mathcal{U}$ and a sufficiently large fingerprint space.

To initialize a Top-K structure in Redis, the user specifies k, m, decay, and K, by calling $\mathsf{TK.setup}(k, m, \text{decay}, K)$. (The analogous function in Redis is called TOPK.RESERVE.) We refer to the resulting structure as $\text{TK}[k, m, \text{decay}, K]$. The hash functions for each row are again computed as $h_1(x) \leftarrow h(x, 1), ..., h_k(x) \leftarrow h(x, k)$, with h set to *MurmurHash2* mod m. The fingerprint hash function is computed as $h_{fp} \leftarrow h(x, seed)$, with h_{fp} set to *MurmurHash2* ($\lambda_{fp} = 32$) with a fixed $seed = 1919$. The decay value is by default set to 0.9.

Insertions and frequency queries on an element x then proceed as described above, through the $\mathsf{TK.ins}(x, \sigma)$ and $\mathsf{TK.qry}(x, \sigma)$ functionalities. Similar to the count-min sketch, multiple instances of an element can be added to the Top-K, however this is implemented through repeated invocations of the insert algorithm described above. To return the top-K elements, one invokes $\mathsf{TK.list}(\sigma)$. (The analogous functions in Redis are called TOPK.ADD, TOPK.COUNT and TOPK.LIST, respectively.) For full details of the Redis Top-K structure, see Figure 4-2.

We will show that the specific implementation choices that Redis makes leads to security issues. Specifically, we give attacks that block the true K most frequent elements from being reported in the top-K estimation (with overwhelming probability) whether or not these elements are known to the attacker before the attack. Further, we show that one is able to trivially violate the NFC assumption and cause the Redis Top-K structure to allow for frequency *over*estimates.

4.2 Attacks Against PDS in Redis

In this section, we construct attacks against the Redis implementations of count-min sketches and Top-K structures (for attacks against Bloom filters and Cuckoo filters see [55]). While our attacks vary in their goals and complexity, at their core, they all exploit Redis' choice of weak hash functions (from the *MurmurHash* family) and their invertibility. By implementing our attacks and giving experimental results, we demonstrate that malicious Redis users can severely disrupt the performance of each PDS. Code for our attacks can be found at [56].

4.2.1 MurmurHash Inversion Attacks

The Redis PDS suite relies heavily on two different *MurmurHash* hash functions: *MurmurHash64A* and *MurmurHash2*. Both functions accept an element, a length parameter and a *seed* as input. The functions have, respectively, 64-bit and 32-bit outputs. In Redis, all inputs must have valid ASCII encoding, as the length field is set to the character length of the string representation of the input. Seeds are usually set to fixed values.

The *MurmurHash* family of hash functions are designed to be fast but are not cryptographically secure. Indeed, starting with a target hash value h and a given seed, it is easy to find one or many elements that hash to h under either *MurmurHash64A* or *MurmurHash2*, so these functions are not even one-way. We refer to these resulting elements as pre-images of h, and the algorithms that compute them as *inversion* algorithms. Our inversion algorithms for *MurmurHash64A* and *MurmurHash2* are about as fast as computing the hash functions in the forward direction. They are based on the deterministic approach in [57]. However, we adapt this method to make our algorithms randomized and to be able to produce many pre-images for the same target hash value h. For *MurmurHash64A*, our inversion algorithm outputs strings consisting of two 64-bit blocks B_1, B_2 in which B_2 is chosen arbitrarily and B_1 is then determined by B_2 and the seed. Similarly, for *MurmurHash2*, but with 32-bit blocks. In both cases, the algorithms can be modified to produce inversions that are t-block messages for any t; then any $t-1$ of the blocks can be freely chosen (with the remaining one then being determined). However, pre-images that comprise two 64-bit or 32-bit blocks suffice for our attacks.

For attacks on Redis, we must also further modify our algorithms to ensure the pre-images are valid ASCII-encoded strings. Meeting this additional requirement incurs extra cost. For *MurmurHash64A*, given a valid ASCII-encoded B_2, ensuring that B_1 has the correct format requires on average 2^8 trial inversions, hence costing roughly the same as 256 forward hash function computations. Here, the factor of 2^8 comes from a 64-bit string representing 8 ASCII characters, each of which must have a single bit set to zero. For *MurmurHash2*, an average of 16 trial inversions is needed to obtain a 2-block pre-image respecting the ASCII constraint. Additionally, we enforce the leading byte of B_1 to be non-zero to ensure that the length of the pre-image, when viewed as a string, is exactly 16 or 8 bytes. This is important as *MurmurHash64A* and *MurmurHash2* outputs depend on the input length. Overall, this results in an average number of an equivalent of $256 \cdot \frac{128}{127} \approx 258$ and $16 \cdot \frac{128}{127} \approx 16$ hash function calls to compute a correctly formatted 2-block pre-image for *MurmurHash64A* and *MurmurHash2*.

It is also possible to construct so-called universal multi-collisions for certain hash functions in the *MurmurHash* family [58]. These are large sets of input values that all hash to the same output, irrespective of the seed. For *MurmurHash64A*, such inputs could be useful in our targeted false positive attack on Redis' Bloom filter below; however, they seem to be difficult to construct while respecting the ASCII encoding requirement. We leave the construction and exploitation of such collisions to future work.

4.2.2 Count-Min Sketch Attack

We give an attack against Count-Min sketches in Redis that causes large frequency overestimates for any target element.

4.2.2.1 Overestimation attack

Consider a Count-Min sketch with parameters ε, δ. After initializing CMS$[\varepsilon, \delta]$ σ, an adversary $\mathcal{A}$ is given access to insertion and query oracles: $\mathbf{Ins}(\cdot) := \mathsf{CMS.ins}(\cdot, \sigma)$ and $\mathbf{Qry}(\cdot) := \mathsf{CMS.qry}(\cdot, \sigma)$. In a frequency overestimation attack, the adversary is given a target element x as input and is challenged with causing the frequency of x to be overestimated. A

metric for the adversary's success is the value $\mathsf{CMS.qry}(x, \sigma) - n_x$, where n_x is the number of times x was actually inserted into the Count-Min sketch.

We begin by recalling that, for a frequency estimation query on an element $x \in \mathcal{U}$, the response given by a Count-Min sketch has one-sided error, i.e. it only overestimates. In the honest setting, this error can be bounded according to the number of items inserted into the structure and the parameters of the structure (see Section 4.1.1). We will show that in an adversarial setting, we can exploit knowledge of the internal randomness of the structure to cause the sketch to make massive overestimates of the frequency of a target element x.

In Section 3.4.2 we present attacks against the general CMS structure. We could directly apply their "public hash" attack to the Redis implementation of the Count-Min sketch, as the seeds used for each row hash function are hard-coded. However, Redis' choice to use *MurmurHash2* for row position hash functions allows us to exploit the invertibility of the function to speed up the attack. As *MurmurHash2* is invertible, we can generate an arbitrary number of multicollisions for a fixed hash output and seed. This allows us to carry out the attack more efficiently.

To create an overestimation error on x, one must find a cover for x, which (with respect to the parameters of a given Count-Min sketch) is a set of elements $\{y_1,...,y_k\}$ such that $\forall i \in [k]: h(x, i) = h(y_i, i)$ and $\forall i \in [k]: y_i \neq x$. We use the fact that *MurmurHash2* is invertible to find our cover. Let p_i denote $h_i(x)$ for $i \in [k]$, where $h_i(\cdot)$ is instantiated using $\mathit{MurmurHash2}(\cdot, i)$ as in Redis. We then set y_i by inverting $\mathit{MurmurHash2}(\cdot, i)$ at x for $i \in [k]$. Respecting Redis' ASCII encoding constraint, we expect this to cost an equivalent of about 16 hash function evaluations for each i (as per Section 4.2.1). Therefore, we expect a total cost of about $16k$ *MurmurHash2* computations. Once the cover is found, we simply repeatedly insert it, using **Ins** calls on y_i for $i \in [k]$. Since we never insert x and our covers are always of size k, after I insertions we observe an error on x equal to $\lfloor \frac{I}{k} \rfloor$, i.e. $\mathsf{CMS.qry}(x, \sigma) - n_x \geq \lfloor \frac{I}{k} \rfloor$. For a full description of our attack, see Figure 4-3.

overestimation_attack$^{\mathbf{Ins}}(x, pp, I)$

cover $\leftarrow$ find_cover(x, pp)
until I insertions are made
  for $e \in$ cover: **Ins**(e)
return done

find_cover(x, pp)

$\varepsilon, \delta \leftarrow pp$
$k \leftarrow \left\lceil ln(\frac{1}{\delta}) \right\rceil$
cover $\leftarrow \emptyset$
$(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
for $i \in [k]$
  $y \leftarrow MurmurHash2Inverse(p_i, i)$
  cover $\leftarrow$ cover $\cup \{y\}$
return cover

Figure 4-3. The count-min sketch overestimation attack. We use the invertibility of *MurmurHash2* to find a cover. We then repeatedly insert the cover to create error. Note that we abuse notation and assume that *MurmurHash2Inverse* is run until a validly encoded pre-image is found.

We remark that the attack also works against structures that already have elements stored in them., as the Count-Min sketch is a linear structure.

We implemented the attack and measured the computation needed for a variety of ε, δ. We compare the error to the forward hash computation based attack of [50] given in Section 3.4.2 with the one we present here. The results are summarized in Table 4-1. As we can see our experimental results tightly match our analysis, and our attack is at least an order of magnitude less expensive than previous best attack in [50]. Further, to verify the correctness of our attack we mounted it against the Redis Count-Min sketch and selected a random target element. We found a cover for said element and verified that for a fixed number of insertions I we obtained the expected error on the target, i.e. achieved error $\lfloor \frac{I}{k} \rfloor$ in all trials.

ϵ, δ (m, k)	Ours	[50]
$2.7 \times 10^{-3}, 1.8 \times 10^{-2}$ $(1024, 4)$	66.85	8533.32
$6.6 \times 10^{-4}, 1.8 \times 10^{-2}$ $(4096, 4)$	61.11	34133.36
$2.7 \times 10^{-3}, 3.4 \times 10^{-4}$ $(1024, 8)$	124.22	22264.72
$6.6 \times 10^{-4}, 3.4 \times 10^{-4}$ $(4096, 8)$	128.8	89058.72

Table 4-1. Experimental number (average over 100 trials) of equivalent *MurmurHash2* calls needed to find a cover for a random target x. We compare the average to the expected number of *MurmurHash2* calls needed in the attack of [50] given in Section 3.4.2, namely kmH_k.

4.2.3 Top-K

We present three attacks on the Top-K structure in Redis. The first two attacks suppress the reporting of the true top-K elements, while the third attack causes frequency overestimates by violating the no-fingerprint collision assumption.

4.2.3.1 Known top-K hiding attack

Consider a Top-K structure with parameters m, k, decay, K. After initializing TK$[m, k,$ decay, $K]$ σ, a collection of data C with true top-K elements F is generated from some honest distribution (that is, a distribution that does not depend on the internal randomness of the structure). In practice, we can take this to be some collection of network traffic or a collection of items in a large database.

Then, an adversary $\mathcal{A}$ is given access to insertion and query oracles $\mathbf{Ins}(\cdot) := \mathsf{TK.ins}(\cdot, \sigma)$ and $\mathbf{Qry}(\cdot) := \mathsf{TK.qry}(\cdot, \sigma)$. In a known top-$K$ hiding attack, the adversary receives F as input and wins if it suppresses the reporting of the true top-K elements F. The adversary's success can be checked by inserting C and checking whether $[f \notin \mathsf{TK.list}(\sigma)]$ for all $f \in F$. Due to the probabilistic decay mechanism, we need the adversary to be able to insert elements into the structure before the honest collection is processed. In practice this is reasonable, as adversaries can time their attacks to ensure they have early access to the structure.

To carry out this attack, we adapt the strategy from [50] given in Section 3.4.3. We begin by computing a cover using the inversion strategy for every element in F. We then insert every element in the cover t times through **Ins**$(\cdot)$ calls, where t is computed such that there exists negligible probability that, after the cover is inserted, any element from F will ever own any of their counters. The algorithm to compute t takes inputs p, n, where p is the probability that a cover element will relinquish ownership of its counters and n is the number of colliding insertions we expect. We set $p = 2^{-128}$ and n to the frequency of the maximum $f \in F$ for this attack. Once C is inserted after the attack phase, all elements in F will have estimated frequency equal to zero, and will in turn not be reported in the top-K list as they should.

In practice, t will be quite small compared to the frequencies of the elements in F for a real-world data collection C. The frequency of all $f \in F$ is often of the order of 10^5 or greater, yielding t of the order of 10^3 for $p = 2^{-128}$. Thus, the true top-K of C equals the top-K of the new stream consisting of our attack elements concatenated with C. For more details on this attack (including the calculation of t), see Figure 4-4.

We expect an equivalent of $16k|F|$ calls to *MurmurHash2* to find a cover for known true top-K list F. To test our attack, we initialized a TK[4096, 20, 0.9, 20], selected our data collection C as the individual words in the English language version of *War and Peace*, and computed F for K=20 for C. Our choice of C was inspired by Redis' blog post introducing the structure [59]. We then computed a cover on F using our technique described above. Averaged over 100 trials, we made an equivalent of 2580 calls to *MurmurHash2*, matching our analysis. We then inserted each element in the cover t times for t=206 based on input parameters p=2^{-128}, n=34577 (the frequency of the most frequent element). After this, the entirety of C was inserted. In every trial, the reported top-K and F were disjoint as desired.

4.2.3.2 Hidden top-K hiding attack

We consider a similar attack model to Section 4.2.3.1 with the modification that the adversary $\mathcal{A}$ receives no input. Since $\mathcal{A}$ does not know F, it must compute a cover for the entire structure, i.e.

known_F_attack$^{\mathbf{Ins}}(F, n, p, pp)$

1 : $t \leftarrow$ get_t(n, p, pp)
2 : F_cover $\leftarrow$ find_F_cover(F, pp)
3 : **for** $e \in$ F_cover
4 : **for** $i \in [t]$
5 : **Ins**(e)
6 : **return** done

get_t(n, p, pp)

1 : $m, k, \text{decay}, K \leftarrow pp$
2 : $g(t) \leftarrow \log_2(k \cdot n^t \cdot \text{decay}^{t(t+1)/2}) - \log_2(p)$
3 : $t_1, t_2 \leftarrow$ FindRootsOf(g)
4 : **if** $t_1 > 1$ **or** $t_2 < 1 : t \leftarrow 1$
5 : **if** $t_2 > 1 : t \leftarrow \lceil t_2 \rceil$
6 : **if** $t_2 = 1 : t \leftarrow 2$
7 : **return** t

find_F_cover(F, pp)

1 : $m, k, \text{decay}, K \leftarrow pp$
2 : F_cover $\leftarrow \emptyset$
3 : **for** $f \in F$
4 : $(p_1, \ldots, p_k) \leftarrow h(f, 1), \ldots, h(f, k)$
5 : **for** $i \in [k]$
6 : $y \leftarrow$ *MurmurHash2Inverse*(p_i, i)
7 : F_cover $\leftarrow$ F_cover $\cup \{y\}$
8 : **return** F_cover

Figure 4-4. The Top-K known top-K hiding attack.

all $k \times m$ counters. We go counter-by-counter and use hash inversion to compute a cover element for each counter. Note, however, that when computing a cover element for a particular counter, we collect additional positions in other rows that the element touches (if we have not yet covered said positions). In this way, we actually do less work than the expected equivalent of $16mk$ calls to *MurmurHash2*.

After computing this cover for the entire structure, $\mathcal{A}$ then inserts each element in the cover t times through $\mathbf{Ins}(\cdot)$ calls, with t=500 (corresponding to p=2^{-128}, n=10^{11} from the previous method of computing t). In practice, setting t=500 means that with overwhelming probability no true top-K element will ever own its counters for any realistic data collection. Then, for any subsequent items inserted that are not part of the cover, their estimated frequency will be zero. In practice, this blocks any F from any realistic data collection C from being reported in the top-K list. This attack can be seen as a denial-of-service attack, as after the attack phase the structure is prevented from making accurate frequency estimates for any elements that are subsequently inserted into the Top-K. Our full attack is given in Figure 4-5.

We verified the correctness of the attack as in Section 4.2.3.1, except again now setting t=500.

4.2.3.3 NFC assumption violation attack

Consider a Top-K structure with parameters $m, k,$ decay, K. After initializing TK[$m, k,$ decay, K] σ, an adversary $\mathcal{A}$ is given access to insertion and query oracles: $\mathbf{Ins}(\cdot) := \mathsf{TK.ins}(\cdot, \sigma)$ and $\mathbf{Qry}(\cdot) := \mathsf{TK.qry}(\cdot, \sigma)$. The adversary's goal in an NFC assumption violation attack equates to the same goal as of that in Section 4.2.2.1. That is, $\mathcal{A}$ receives x as input and is challenged with causing the frequency of x to be overestimated. Again we can use $\mathsf{TK.qry}(x, \sigma) - n_x$ as a metric of success, where n_x is the number of times x was actually inserted into the Top-K structure.

Recall that under the no-fingerprint collision assumption, the Top-K structure only underestimates frequencies of elements. We will show that, with the Redis implementation of Top-K, it is trivial to violate this assumption, and thus create large frequency overestimation errors.

```
hidden_F_attack^Ins(n, p, pp)
 1 :  t ← 500
 2 :  S_cover ← find_S_cover(pp)
 3 :  for e ∈ S_cover
 4 :     for i ∈ [t]
 5 :        Ins(e)
 6 :  return done
find_S_cover(pp)
 1 :  m, k, decay, K ← pp
 2 :  η ← zeros(k, m)
 3 :  S_cover ← ∅
 4 :  for i ∈ [k]
 5 :     for j ∈ [m]
 6 :        if η[i][j] = 0
 7 :           y ← MurmurHash2Inverse(j, i)
 8 :           S_cover ← S_cover ∪ {y}
 9 :           (p_1, ..., p_k) ← h(y, 1), ..., h(y, k)
10 :           for r ∈ [k]
11 :              η[r][p_r] ← 1
12 :  return S_cover
```

Figure 4-5. The Top-K hidden top-K attack.

(m, k)	*MurmurHash2* inversions	*MurmurHash2* calls
$(1024, 4)$	4296.69	1072.52
$(4096, 4)$	18489.68	4602.56
$(1024, 8)$	1849.71	905.44
$(4096, 8)$	10058.16	5031.52

Table 4-2. Experimental number (averaged over 100 trials) of *MurmurHash2* inversion trials and *MurmurHash2* calls needed to find a cover element for a randomly selected target x. Recall that the cost of each is about the same.

To create large error on a given target x, we compute multicollisions on the fingerprint of x, stopping when we find a collision such that it shares one row position with x. Unlike the attack against the Count-Min sketch, we only need to find such a collision in one row, as the Top-K takes the maximum count over all owned counters. Therefore, we are now finding a single cover element y. Then, by inserting the cover element I times using $\mathbf{Ins}(y)$, $\mathcal{A}$ can expect to create error I on the frequency estimation of x, i.e. $\mathsf{TK.qry}(x, \sigma) - n_x \geq I$. Experimental results measuring the cost for this attack are given in Table 4-2. We need *MurmurHash2* computations (k per successful inversion) to check if the collision element we found matches any of the row positions to which our target maps.

We verified the correctness of the attack in the same way as in Section 4.2.2.1, obtaining the expected error I on the randomly select target x over all trials. For more details of our attack, see Figure 4-6.

4.3 Potential Countermeasures and Concluding Remarks

In this section, we outline some countermeasures that limit the effectiveness of our attacks. For remarks on the Bloom filter and Cuckoo filter we again refer the reader to [55].

Protecting the count-min sketch and Top-K against frequency estimation attacks is challenging. Recall, that both the Count-Min sketch and the Top-K are a class of probabilistic data structures called *compact frequency estimators* (CFE). In Chapter 3 we explore both of these structures in detail, and show that even when switching the hash functions to a keyed primitive (e.g. a PRF) and keeping the internal state of the structure efficient attacks that cause massive frequency estimation errors are still possible [50]. That is the leakage from insertions and queries to a black-boxed structure is sufficient to carry out the style of attacks we present in this paper. The choices of Redis make these attacks easier to carry out, but findings are negative in any case.

It is of great interest to explore secure PDS for frequency estimate queries that are tenable for real world applications. One could of course disallow queries to the structure, or use some public-key

nfc_violation_attack$^{\mathbf{Ins}}(x, pp, I)$

```
1 :  y ← find_cover_element(x, pp)
2 :  until I insertions are made
3 :     Ins(y)
4 :  return done
```

find_cover_element(x, pp)

```
1 :  m, k, decay, K ← pp
2 :  seed ← 1919
3 :  done ← ⊥
4 :  P ← (h(x, 1), ..., h(x, k))
5 :  fp_x ← h_fp(x)
6 :  while done = ⊥
7 :     y ← MurmurHash2Inverse(fp_x, seed)
8 :     C ← (h(y, 1), ..., h(y, k))
9 :     for i ∈ [k]
10 :       if P[i] = C[i]
11 :          done ← ⊤
12 :  return y
```

Figure 4-6. The Top-K no-fingerprint collision violation attack. We use the invertibility of *MurmurHash2* to find a single fingerprint collision and row pair element for the target x. We then repeatedly insert the element to create error.

infrastructure to only allow insertions from authenticated parties. However, this clearly limits both the usability and performance. Another possibility is to explore new ways of constructing frequency estimation PDS, such as the Count-Keeper introduced in [50]. While this structure remains susceptible to the types of attacks we present here, they are less effective, and the Count-Keeper has a native ability to flag suspicious frequency estimates.

4.3.1 Concluding Remarks

We made a comprehensive security analysis of the Redis PDS suite, developing 10 different attacks across four PDS. Our attacks can be used to cause severe disruptions to the performance of systems relying on these PDS, ranging from mis-estimation of data statistics to triggering denial-of-service attacks. Our work illustrates the importance of low-level algorithmic choices and the dangers of using weak hash functions in PDS.

Our work opens up interesting directions for future work. Various other PDS suites exist in the wild, such as in Google BigQuery and Apache Spark, and could also be subjected to detailed security analysis as we have done for Redis here. Methods to provably protect PDS against attacks have been proposed in [60, 5, 54, 49]. However, these analyses tend to focus on textbook versions of the PDS. Adapting these analyses to cater to the specifics of different implementations would help improve confidence in the deployed variants.

At a higher level, there still seems to be a lack of understanding in the broader developer community about the risks of using PDS in potentially adversarial settings. Work is needed to educate developers about these risks; we hope this paper can play a part in this effort. As an alternative, in an effort to shield developers from these risks, one could develop new PDS implementations that are secure by default and package them in the form of easily consumed libraries with safe APIs. Such an effort could leverage the experience that the research community has gained from developing "safe by default" cryptographic libraries.

CHAPTER 5
PROVABLY ROBUST SKIPPING-BASED PROBABILISTIC DATA STRUCTURES

TODO!

CHAPTER 6
CONCLUSION AND FUTURE WORK

The following needs to be re-organized and re-written.

LIST OF REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2022.

[2] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *DMTCS Conference on Analysis of Algorithms*, 2007.

[4] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[5] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic data structures in adversarial environments," in *ACM SIGSAC CCS*, 2019.

[6] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM CCR*, vol. 45, p. 52–66, July 2015.

[7] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] B. Goodwin, M. Hopcroft, D. Luu, A. Clemmer, M. Curmei, S. Elnikety, and Y. He, "Bitfunnel: Revisiting signatures for search," in *ACM SIGIR Conference on Research and Development in Information Retrieval*, 2017.

[9] "Bip 37." https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki.

[10] M. Naor and E. Yogev, "Bloom filters in adversarial environments," in *Annual Cryptology Conference*, 2015.

[11] M. Filić, K. Paterson, A. Unnikrishnan, and F. Virdia, "Adversarial correctness and privacy for probabilistic data structures," in *ACM SIGSAC CCS*, 2022.

[12] K. G. Paterson and M. Raynal, "Hyperloglog: Exponentially bad in adversarial settings," in *EuroS&P*, 2022.

[13] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, and J. Rexford, "Netscope: Traffic engineering for ip networks," *IEEE Network*, vol. 14, no. 2, pp. 11–19, 2000.

[14] A. Lakhina, M. Crovella, and C. Diot, "Characterization of network-wide anomalies in traffic flows," in *ACM SIGCOMM Conference on Internet Measurement*, 2004.

[15] L. Melis, G. Danezis, and E. De Cristofaro, "Efficient private statistics with succinct sketches," *arXiv preprint arXiv:1508.06110*, 2015.

[16] "Redisbloom: Probabilistic data structures for redis." https://oss.redis.com/redisbloom/.

[17] T. Yang, H. Zhang, J. Li, J. Gong, S. Uhlig, S. Chen, and X. Li, "Heavykeeper: An accurate algorithm for finding top-k elephant flows," *IEEE/ACM Transactions on Networking*, vol. 27, no. 5, pp. 1845–1858, 2019.

[18] B. Sigurleifsson, A. Anbarasu, and K. Kangur, "An overview of count-min sketch and its applications." https://easychair.org/publications/preprint/gNlw, 2019.

[19] "Redis is an open source (BSD licensed), in-memory data structure store, used as a database, cache, and message broker.." https://redis.io/.

[20] A. Hassidim, H. Kaplan, Y. Mansour, Y. Matias, and U. Stemmer, "Adversarially robust streaming algorithms via differential privacy," in *NeurIPS*, 2020.

[21] M. Hardt and D. P. Woodruff, "How robust are linear sketches to adaptive inputs?," in *ACM STOC*, 2013.

[22] E. Cohen, X. Lyu, J. Nelson, T. Sarlós, M. Shechner, and U. Stemmer, "On the robustness of countsketch to adaptive inputs." https://arxiv.org/abs/2202.13736, 2022.

[23] O. Ben-Eliezer, R. Jayaram, D. P. Woodruff, and E. Yogev, "A framework for adversarially robust streaming algorithms," *Journal of the ACM*, vol. 69, no. 2, 2022.

[24] S. A. Markelon, M. Filić, and T. Shrimpton, "Compact frequency estimators in adversarial environments." Cryptology ePrint Archive, Paper 2023/1366, 2023.

[25] H. Liu, Y. Sun, and M. S. Kim, "Fine-grained ddos detection scheme based on bidirectional count sketch," in *International Conference on Computer Communications and Networks*, 2011.

[26] A. Mandal, H. Jiang, A. Shrivastava, and V. Sarkar, "Topkapi: parallel and fast sketches for finding top-k frequent elements," *NeurIPS*, 2018.

[27] R. Berinde, P. Indyk, G. Cormode, and M. J. Strauss, "Space-optimal heavy hitters with strong error bounds," *ACM Transactions on Database Systems*, vol. 35, no. 4, 2010.

[28] A. Metwally, D. Agrawal, and A. E. Abbadi, "An integrated efficient solution for computing frequent and top-k elements in data streams," *ACM Transactions on Database Systems*, vol. 31, p. 1095–1133, sep 2006.

[29] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," in *International Colloquium on Automata, Languages, and Programming*, pp. 693–703, 2002.

[30] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams," in *International Conference on Very Large Databases*, 2002.

[31] H. Melville, *Moby Dick; Or, The Whale*. Project Gutenberg, 1851.

[32] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet.," *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[33] G. Cormode and S. Muthukrishnan, "What's hot and what's not: tracking most frequent items dynamically," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 249–278, 2005.

[34] T. Roughgarden and G. Valiant, "Cs168: The modern algorithmic toolbox lecture #2: Approximate heavy hitters and the count-min sketch," p. 15.

[35] N. Homem and J. P. Carvalho, "Finding top-k elements in data streams," *Information Sciences*, vol. 180, no. 24, pp. 4958–4974, 2010.

[36] R. Real and J. M. Vargas, "The probabilistic basis of jaccard's index of similarity," *Systematic biology*, vol. 45, no. 3, pp. 380–385, 1996.

[37] "Probabilistic: Probabilistic data structures in Redis.." https://redis.io/docs/data-types/probabilistic/.

[38] R. T. RedisConf 2021, "How Adobe uses the Enterprise tier of Azure Cache for Redis to serve push notifications, Adobe," 2024. https://www.youtube.com/watch?v=OslaeJEXW5k.

[39] C. M. RedisDays New York 2022, "Using AI to Reveal Trading Signals Buried in Corporate Filings," 2024. https://www.youtube.com/watch?v=_Lrbesg4DhY.

[40] G. Y. Redis Day TLV 2016, "Redis @ Facebook," 2024. https://www.youtube.com/watch?v=XGxntWcjI24.

[41] R. B. RedisConf 2021, "Redis on the 5G Edge: Practical advice for mobile edge computing, Verizon," 2024. https://www.youtube.com/watch?v=NwQwE2JAIXc.

[42] K. J. The Data Economy Podcast, "Using Real-Time Data and Digital Twins to Improve Cyber Security," 2024. hhttps://www.youtube.com/watch?v=TycylT0J6cc.

[43] S. Sanfilippo, "A few things about redis security.." http://antirez.com/news/96.

[44] T. Fiebig, A. Feldmann, and M. Petschick, "A one-year perspective on exposed in-memory key-value stores," in *ACM Workshop on Automated Decision Making for Active Cyber Defense*, 2016.

[45] "Redis security: Security model and features in Redis.." https://redis.io/docs/latest/operate/oss_and_stack/management/security/.

[46] T. Gerbet, A. Kumar, and C. Lauradoux, "The power of evil choices in bloom filters," in *DSN*, 2015.

[47] D. Desfontaines, A. Lochbihler, and D. Basin, "Cardinality Estimators do not Preserve Privacy," in *PETs*, 2019.

[48] P. Reviriego and D. Ting, "Security of hyperloglog (HLL) cardinality estimation: Vulnerabilities and protection," *IEEE Commun. Lett.*, vol. 24, no. 5, pp. 976–980, 2020.

[49] K. G. Paterson and M. Raynal, "Hyperloglog: Exponentially bad in adversarial settings," in *EuroS&P*, 2022.

[50] S. A. Markelon, M. Filić, and T. Shrimpton, "Compact frequency estimators in adversarial environments," in *ACM SIGSAC CCS*, 2023.

[51] M. R. Albrecht and K. G. Paterson, "Analysing cryptography in the wild - a retrospective." Cryptology ePrint Archive, Paper 2024/532, 2024. https://eprint.iacr.org/2024/532.

[52] T. Dunning, "The t-digest: Efficient estimates of distributions," *Software Impacts*, vol. 7, p. 100049, 2021.

[53] T. M. Kodinariya, P. R. Makwana, *et al.*, "Review on determining number of cluster in k-means clustering," *International Journal*, vol. 1, no. 6, pp. 90–95, 2013.

[54] M. Filić, K. G. Paterson, A. Unnikrishnan, and F. Virdia, "Adversarial correctness and privacy for probabilistic data structures," in *ACM SIGSAC CCS*, 2022.

[55] M. Filić, J. Hofmann, S. A. Markelon, K. G. Paterson, and A. Unnikrishnan, "Probabilistic data structures in the wild: A security analysis of redis." Cryptology ePrint Archive, Paper 2024/1312, 2024.

[56] "PDS in the Wild GitHub Repository." https://anonymous.4open.science/r/PDS-in-the-Wild-A-Security-Analysis-of-Redis-5365.

[57] "Coding blog." https://bitsquid.blogspot.com/2011/08/code-snippet-murmur-hash-inverse-pre.html.

[58] J.-P. Aumasson, D. J. Bernstein, and M. Boßlet, "Hash-flooding DoS reloaded: attacks and defenses." https://web.archive.org/web/20130913185247/https://131002.net/siphash/siphashdos_appsec12_slides.pdf.

[59] "Redis blog post on top-k." https://redis.com/blog/meet-top-k-awesome-probabilistic-addition-redis/.

[60] M. Naor and E. Yogev, "Bloom filters in adversarial environments," in *CRYPTO*, Lecture Notes in Computer Science, 2015.

BIOGRAPHICAL SKETCH

- Nmae plus Research interests
- PhD at UF
- UConn – BS + Goldwater Scholar
- Proof trading