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

For my dearest Julia, without whom this would mean less

ACKNOWLEDGEMENTS

This is the last section of my dissertation, and somehow, it also feels like the toughest (I would be remiss to forget anyone!). While defending a dissertation and earning a doctorate may seem like a solitary achievement, I can unequivocally assure you that it is not. This accomplishment (and my life trajectory in general) would not have been possible without a wonderful and unwavering support system, both personal and professional.

I would like to begin by thanking my parents, Dawn and Tim. They have been nothing but loving and encouraging since the day I was born and gave me an almost idyllic childhood in the wonderful small town of Burlington, Connecticut. Instilling in me and my siblings, Jack and Hannah (whom I also owe a lot to), the values of education and hard work was a gift beyond measure. I also want to thank the entirety of my (very large) extended family – you have forever impressed upon me the importance of family and gathering. I especially want to acknowledge my maternal grandparents, my Nonni and Boppi. They are my only local relatives in north-central Florida, and I have greatly enjoyed the many dinners and visits over the past five years. It is with great sadness that my Boppi is not here to witness the end of my doctoral journey, but his memory is a blessing, and I am comforted by the thought that he is at peace. Lastly, to my soon-to-be in-laws, Kelly and Brian: thank you for all of your support over the last five years (especially the plane tickets to visit home), for making me a part of your family, and above all, for giving me the greatest gift of all – my fiancée Julia.

Professionally, I have had many mentors who encouraged my intellectual curiosity and guided my growth as a scientist. At the University of Connecticut, I am grateful to Dr. Kyungseon Joo for introducing me to research and offering incredible opportunities to a wide-eyed underclassman. Dr. Walter Krawec welcomed me to his quantum key distribution lab, redirecting me from physics toward computer science and cybersecurity. I also thank him for his mentorship on my first academic papers. Lastly, I want to thank Dr. Benjamin Fuller and Dr. Amir Herzberg for introducing me to cryptography and inspiring my passion for it.

I would not be who I am today without Dr. Thomas Shrimpton. Quite literally, meeting Tom at Real World Crypto 2020 convinced me to go to the University of Florida and work with him. I am deeply thankful for his guidance over the past five years and for suggesting (or perhaps imposing upon me) the research topic that forms this dissertation. I will never forget our sessions at the chalkboard, marathon Slack chats, and all-night paper pushes. Although Tom is not my advisor of record due to his bewildering decision to move to industry (I jest, of course), he deserves the most credit for my development as a researcher.

I also extend my thanks to my committee – Dr. Vincent Bindschaedler, Dr. Patrick Traynor, Dr. Sara Rampazzi, and Dr. Jeremy Booher – for their encouragement and helpful suggestions along the way. I am especially grateful to Vincent for stepping in as my advisor and helping (or dragging) me across the finish line. I also want to recognize Dr. Kevin Butler, who, although not on my committee, provided invaluable leadership at the Florida Institute for Cybersecurity (FICS). To all the faculty, administrators, and students (past and present) of FICS, as well as to the institute itself – my deepest gratitude. It is a truly wonderful place to work, and I could not have asked for a better lab to complete my doctorate in. While the volume and impact of research coming out of FICS is impressive, it is the people and the culture that make it exceptional. You will not find a group of smarter, more innovative, friendly, and honest individuals. I am honored to have been a small part of it.

During my PhD, I was fortunate to participate in two international collaborations. I am especially thankful to Dr. Kenneth Paterson (ETH Zurich, Applied Cryptography Group) and Dr. Marc Fischlin (TU Darmstadt, Cryptoplexity Group) for hosting me for research visits and collaborating on works that appear in this dissertation. Mia Filić (ETH Zurich) deserves special recognition – she is my longest collaborator and a dear friend. Without her, this work would not have been possible. I also wish to acknowledge Moritz Huppert (TU Darmstadt); he has been an insightful and tenacious collaborator and deserves credit for introducing me to many delicious German beers (and his more limited success at convincing me of the joys of German humor). I also thank all the students of these groups – they were incredibly welcoming during my visits, and

I am glad to call many of you friends. A special thanks to Nicholas-Philip Brandt (ETH Zurich); although Nico and I did not collaborate on work in this dissertation, he is one of the most meticulous researchers I know.

Beyond my family and professional mentors, this journey would not have been possible without a wide and geographically dispersed group of friends. To Tim, Jefferson, Logan, Gavin, and John – thank you for making Gainesville fun these past five years. To my UConn friends Jake, Andy, Owen, Mitchell, Kevin, Skippy, Stofko, Bucco, and Paul – thank you for all the shenanigans, for visiting me in Florida, and for Huskies basketball (back-to-back national champions!). To my childhood friends Cameron, Jack, Alex, and Tom – thank you for making it feel like no time has passed when we get together. To everyone who has ever played a round of bar trivia with me, and to all my other dear friends whom I could not possibly enumerate – thank you.

Lastly, and above all, thank you to my Julia. Your partnership over the past decade has kept me steady. Thank you for your unwavering support (even when I decided to move over a thousand miles away to pursue a PhD with some guy named Tom I met at a conference). You are everything I could have dreamed of in a partner. I am beyond excited to marry you and build our life together. I love you with all my heart.

TABLE OF CONTENTS

LIST OF TABLES

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 dissertation investigates the security of widely-used data structures under adversarial conditions, with a particular focus on compact frequency estimators (CFEs) and probabilistic skipping-based data structures (PSDS). These structures, though efficient and fundamental to many modern systems, have historically been analyzed primarily through the lens of performance and operational efficiency, with security considerations treated as an afterthought.

Compact probabilistic data structures (CPDS), such as Bloom filters, Count-min sketch (CMS), and HeavyKeeper (HK), are designed to provide compact representations of large datasets, supporting approximate queries with bounded error probabilities. However, existing correctness guarantees implicitly assume non-adaptive adversaries. This dissertation reveals that adaptive adversaries can severely degrade the accuracy of these structures. In particular, it presents both theoretical and experimental attacks against CMS and HK that exploits a mixture of fixed randomness and past query responses, leading to significant frequency estimation errors. To counteract these vulnerabilities, the dissertation introduces Count-Keeper, a novel CFE that offers improved accuracy for honest data streams, resilience against adaptive attacks, and a native mechanism for flagging suspicious estimates.

Beyond theoretical analysis, this work evaluates CFE implementations in Redis, a widely deployed in-memory database system. Redis's reliance on non-cryptographic hash functions and deviations from standard CFE designs enable novel attacks that surpass those possible against

generic versions. This analysis highlights the critical need for secure implementations in practice and proposes countermeasures to mitigate these threats.

Probabilistic skipping-based data structures (e.g., hash tables, skip lists, and treaps) also exhibit vulnerabilities under adversarial conditions. While these structures achieve efficient operations by leveraging randomness, adaptive adversaries can force worst-case performance, causing exponential degradation in their operational efficiency. This dissertation presents attacks on these structures and proposes robust, performant variants. The robustness of these variants is formalized through the Adaptive Adversary Property Conservation (AAPC) framework, which quantifies deviation from expected performance under adversarial influence. Analytical proofs and experimental validation confirm the efficacy of these designs.

Collectively, this dissertation advances the security analysis of data structures from theoretical constructs to real-world implementations, bridging the gap between performance and provable security.

CHAPTER 1
INTRODUCTION

Data structures define representations of (possibly dynamic) sets or multisets, along with the operations that can be performed on these representations. 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 treated as 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 manipulate the data they represent. Furthermore, complex, modern protocols with sophisticated security goals increasingly rely on bespoke data structures as fundamental components of their designs. It is therefore both timely and prudent to apply the provable security paradigm to data structures themselves.

Consider, the family of *compressing probabilistic data structures* (CPDS). The use of CPDS has grown rapidly in recent years in correlation with the rise of distributed applications producing and processing colossal amounts of data. These structures provide compact representations of (potentially massive) data collections, 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. A canonical example is the Bloom filter [2] which supports set-membership queries (*Does element x appear in the data?*). Bloom filters have found widespread use in applications such as caching [3], database query optimization [4], search engine indexing [5], and even Bitcoin wallet synchronization [6].

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. Recent research, however, has revealed that many data structures, including Bloom filters, perform poorly under adaptive adversaries, which tailor queries based on previous responses and knowledge of the underlying hash functions [7, 8, 9, 10]. Similar vulnerabilities have been demonstrated for HyperLogLog structures [11], which estimate the number of distinct elements in a collection [12]. Another important subclass of CPDS is the family of compact frequency estimators (CFEs), which includes structures such as Count-,in sketch and HeavyKeeper. Unlike Bloom filters, which answer membership queries, CFEs estimate the frequency of elements in a data stream. Despite their increasing adoption in network monitoring, stream processing, and heavy hitter detection, the formal analysis of CFEs has lagged behind their practical deployment. In particular, their robustness under adversarial conditions remains largely unexplored.

Another critical family of data structures is what we refer to as *probabilistic skipping-based data structures* (PSDS), including hash tables, skip lists, treaps, skip graphs, and randomized meldable heaps. Unlike CPDS, these structures are not space-efficient, but, in turn, offer exact answers to queries. Their design leverages randomness to achieve fast average-case performance – often constant or logarithmic time for search, insertion, and deletion – but with worst-case runtimes linear in the collection size. This randomness, while beneficial for expected performance, creates an attack surface: adversaries can exploit structural weaknesses to force worst-case behavior.

This is exemplified by complexity attacks against hash tables, where an adversary crafts input patterns that induce excessive hash collisions, resulting in performance degradation to linear time. While the underlying idea (exploiting structural dependencies in hashing) may seem conceptually simple, the real-world consequences are severe, including denial-of-service (DoS) vulnerabilities in many widely used applications [13, 14, 15, 16, 17]. Despite a range of proposed mitigations, formal security models for these attacks and provable guarantees for the mitigations are conspicuously lacking. Even less is known about how other PSDS structures, such as skip lists, treaps, zip trees, and randomized meldable heaps, behave in adversarial settings. Notably, the only substantive work addressing adversarial behavior in PSDS beyond hash tables is that of Nussbaum and Segal, who demonstrated a timing side-channel attack against skip lists [18].

The need for provably robust data structures is thus not merely theoretical, but essential for securing modern systems against adaptive adversaries. This dissertation advances this frontier by investigating attacks and constructing formally secure variants of CPDS and PSDS, with a particular emphasis on compact frequency estimators (a subclass of CPDS) and hash tables, skip lists, and treaps (PDSS). By doing so, we aim to provide both a rigorous foundation for secure data structures and practical tools for safeguarding real-world applications.

1.1 Thesis Statement

The goal of this dissertation is to rigorously analyze data structures in adversarial environments, specifically focusing on both compressing probabilistic data structures and probabilistic skipping-based data structures, and to construct performant and provably robust variants of these structures. The emphasis is on adaptive adversaries, who are capable of selecting queries and data inputs based on prior interactions with the structure, knowledge of the randomness used to initialize the structure, and the representation the structure has collection for a given data collection.

Therefore, the central thesis of this dissertation is:

For compact frequency estimators (a subclass of compressing probabilistic data structures) and probabilistic skipping-based data structures (including hash tables, skip lists, and treaps), formal adversarial models that capture the adaptive ability of adversaries can be defined under which these structures are provably insecure. Specifically, these models capture scenarios in which an adversary, with knowledge of the structure's parameters, query responses, and, in certain cases, initialization choices and representations, can degrade correctness or performance guarantees beyond acceptable thresholds. It is further claimed that, for these same adversarial models, it is possible to construct new variants of these data structures that are provably robust, with explicit, formal guarantees on their correctness, performance, and security under attack.

1.2 Contributions

In addressing the goals of formalizing the security of compact frequency estimators and probabilistic skipping-based data structures, and in providing provably robust versions, this dissertation makes the following contributions:

Compact Frequency Estimators in Adversarial Environments. Count-min sketch (CMS) and HeavyKeeper (HK) are two realizations of compact frequency estimators (CFEs), a subclass of compressing probabilistic data structures that maintain a compact summary of (typically) high-volume streaming data. CFEs provide approximate estimates of the number of times a particular element has appeared in the stream. They often serve as foundational structures in systems identifying the highest-frequency elements (e.g., top-K elements, heavy hitters, elephant flows). Traditionally, the probabilistic guarantees on the accuracy of CFEs are proved under the implicit assumption that stream elements are independent of the internal randomness of the structure—in other words, under the assumption of non-adaptive adversaries. However, in many practical scenarios, particularly those involving malicious actors incentivized to manipulate the data stream, this assumption does not hold. We demonstrate that both CMS and HK can be forced to make significant estimation errors via concrete attacks exploiting adaptivity. These attacks are analyzed both analytically and experimentally, with tight agreement between theory and practice. Unfortunately, our results suggest that such vulnerabilities may be inherent to sketch-based CFEs with parameters practical for real-world use. On a positive note, we introduce a new CFE, *Count-Keeper*, which can be viewed as a composition of CMS and HK. Count-Keeper yields estimates that are typically more accurate (by at least a factor of two) than CMS for ”honest” streams; our adaptive attacks are less effective (and more resource-intensive) against Count-Keeper; and Count-Keeper uniquely supports flagging suspicious estimates – an ability absent in CMS, HK, and, to our knowledge, any other known CFE.

Compact Frequency Estimators in the Wild: A Case Study of Redis. Redis (Remote Dictionary Server) is a general-purpose, in-memory database that supports a rich array of functionality, including various CPDS, and in particular, two CFEs: CMS and Top-K (based on

HK). As aforementioned, CFEs typically perform well on average-case inputs, their performance can degrade severely under adaptive adversaries. Given Redis's wide deployment across diverse applications, it is crucial to evaluate the resilience of these CFEs under worst-case scenarios, specifically adversarial inputs. We conduct a comprehensive analysis of Redis's CFE implementations, detailing deviations from the structures as described in the literature. We demonstrate that these deviations enable four novel attacks, each more severe – or outright impossible – compared to attacks on the generic versions of the CFEs. We highlight the critical role of Redis's choice to use non-cryptographic hash functions in the severity of these attacks. Finally, we discuss potential countermeasures to mitigate these vulnerabilities.

Provably Robust Probabilistic Skipping-Based Data Structures. Probabilistic skipping-based data structures – such as hash tables, skip lists, and treaps – support efficient operations through randomized hierarchies that enable "skipping" elements, achieving sublinear query complexity in the average case for perfectly correct responses. These structures are critical in performance-sensitive systems where correctness is essential and efficiency is highly desirable. While simpler than deterministic alternatives like balanced search trees, these structures traditionally assume that input data is independent of the structure's internal randomness and state – an assumption that breaks down in adversarial environments. Under adaptive adversaries, this can lead to significant degradation in query performance. We present adaptive attacks on hash tables, skip lists, and treaps that, in the case of hash tables and skip lists, induce exponential performance degradation compared to the input-independent setting. While attacks on hash tables have been well studied, our attacks on skip lists and treaps offer new insights into vulnerabilities in probabilistic skipping-based data structures. In response, we propose simple and efficient modifications to these structures, yielding provably secure variants under adaptive adversaries. Our approach is formalized through the *Adaptive Adversary Property Conservation* (AAPC) framework, a general security notion capturing deviation from expected efficiency guarantees in adversarial settings. Using this framework, we present rigorous robustness proofs for our proposed variants. Lastly, we conduct experiments whose empirical results align closely with our analytical predictions.

1.3 Outline and Publications

The remainder of this dissertation is organized as follows. Chapter 2 introduces the necessary notation and syntax used throughout the document, as well as key terms and a review of related work. Chapter 3 examines compact frequency estimators in adversarial environments. Chapter 4 presents our analysis of compact frequency estimator implementations in Redis and their limited robustness under adversarial conditions. Chapter 5 focuses on probabilistic skipping-based data structures, providing robust efficiency guarantees. Finally, Chapter 6 offers concluding remarks and discusses potential directions for future work.

This dissertation is based on the following publications[1]:

- Chapter 3 is based on:

  Sam A. Markelon, Mia Filić, and Thomas Shrimpton. 2023. Compact Frequency Estimators in Adversarial Environments. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS ’23).

- Chapter 4 is based on:

  Mia Filić, Jonas Hofmann, Sam A. Markelon, Kenneth G. Paterson, and Anupama Unnikrishnan. 2025. Probabilistic Data Structures in the Wild: A Security Analysis of Redis. In Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy (CODASPY ’25).

- Chapter 5 is based on:

  Marc Fischlin, Moritz Huppert, and Sam Markelon. 2025. Probabilistic Skipping-Based Data Structures with Robust Efficiency Guarantees. In submission to the 2025 ACM SIGSAC Conference on Computer and Communications Security (CCS ’25).

[1]Publications with a majority European collaboration use alphabetical author ordering.

CHAPTER 2
BACKGROUND

2.1 Notation

Bitstring and Set Operations.

Let $\{0,1\}^*$ denote the set of bitstrings and let ε denote the empty string. Let $X \parallel Y$ 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 $\parallel$ operator and write $\mathcal{S} \parallel e$.

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. Moreover, we denote by $\mathsf{U}(S)$ the uniform distribution on the (finite or uncountable) set $S \neq \emptyset$, and by $\mathsf{G}(p)$ be the geometric distribution for success probability p.

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 $\textsc{id}^f : \{\varepsilon\} \times \mathcal{X} \to \mathcal{Y}$ so that $\textsc{id}^f(\varepsilon, x) = f(x)$ for all x in the domain of f. This allows us to use unkeyed hash functions H in situations where, syntactically, a 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 syntax for data structures first provided by [8]. 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 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:

- $\textsc{Rep}\colon \mathcal{K} \times \mathcal{D} \to \{0,1\}^* \cup \{\bot\}$ 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 $\bot$ in the case of a failure. We write this as $\mathsf{repr} \leftarrow \textsc{Rep}_K(\mathcal{S})$.

- $\textsc{Qry}\colon \mathcal{K} \times \{0,1\}^* \times \mathcal{Q} \to \mathcal{R} \cup \{\bot\}$ 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 $\bot$ in the case of a failure. We write this as $a \leftarrow \textsc{Qry}_K(\mathsf{repr}, \mathsf{qry})$.

- $\textsc{Up}\colon \mathcal{K} \times \{0,1\}^* \times \mathcal{U} \to \{0,1\}^* \cup \{\bot\}$ 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 $\bot$ 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 Related Works

We now provide a summary of related work for the classes of data structures that we focus on in this dissertation.

2.4.1 Compact Probabilistic Data Structures and Compact Frequency Estimators

2.4.1.1 Approximate Set Membership Data Structures

The first works to explore CPDS in a provable security style focused on the Bloom filter [2]. The Bloom filter admits approximate set-membership queries. The structure is widely used in many computing contexts, such as databases [19], networking [20], distributed systems [21], and search [5].

Naor and Yogev were the first to consider settings in which inputs and queries may be chosen by an adaptive adversary and formally investigate attacks that can occur in such a setting [7]. Their results show that adversaries can find queries that are guaranteed to be false positive for a given instantiation of a filter and data collection. They formalized a notion of adversarial correctness for a modified Bloom filter structure of their own construction and provide a correctness bound for it. Clayton et al. [8] extend this work by considering stronger adversaries. They allow for the adversary to insert elements into the structure after the adversary has started to issue queries – that is, they consider a fully mutable setting. They find that the basic Bloom filter is vulnerable to adversarial manipulation, which can increase false positives to nearly 100%. To secure it, they recommend adding a unique salt in an immutable setup, or using a private representation, keyed hash functions, and insertion thresholds in a mutable setting. Further, they formalize a notion of adversarial correctness that extends past only Bloom filters, also concretely analyzing the counting filter [22] and the Count-min sketch [23]. Filić et al. [24, 10] further analyze the adversarial correctness of Bloom filters and Cuckoo filters (another approximate membership data structure) in a simulation style security notion. They reach similar conclusions to [8].

2.4.1.2 HyperLogLog

The HyperLogLog (HLL) [12] is a CPDS that provides a compact representation of a set and can accurately approximate the number of distinct elements in the set (i.e., the set's cardinality). Patterson and Raynal [11] provide a provable security treatment of the HLL. They first present attacks which exploit the use of fixed and publicly computable hash function in the HLL to cause large cardinality estimate errors. They then show that by switching these hash functions for a secretly keyed primitive that (even in the setting where an adversary has complete access to the internal state of the structure) the structure remains secure in terms of conserving the non-adversarial correctness guarantees of the structure. Prior to this, Revirigeo and Ting provide attacks against the HLL in a model where the adversary has access to a "shadow" device that mirrors the structure that is being attacked [25]. Patterson and Raynal point out this setting in unrealistic, but nonetheless improve the attack in this model.

2.4.1.3 Compact Frequency Estimators

Recall that compact frequency estimators are a class of CPDS that compactly represent a collection of streaming data (usually modeled as a multiset), and provide approximately correct frequency estimates (that is, the number of times any particular element has appeared in the stream). Alternately, compact frequency estimators can be viewed as providing a compact representation of the frequency distribution of a particular data stream.

As previously stated, Clayton et al. were the first to examine compact frequency estimators from a provable security perspective [8]. They specifically examined the Count-min sketch and presented attacks that could cause large frequency estimation error when the internals state of the structure or the hash functions used by the structure were made available to the adversary. They were able to prove security of the structure when the internal state of the structure is kept private and a secretly keyed primitive was used in place of the usual hash functions. However, their defined adversarial goal was very conservative. Any fixed amount of frequency estimation error was considered a win for the adversary, rather than an accumulated error that surpassed that of the non-adversarial correctness guarantee. Further, their construction relied on a thresholding technique, in which the structure would not accept any more updates after a bounded number of insertions – something that in practice is unrealistic.

2.4.2 Probabilistic Skipping-Based Data Structures

2.4.2.1 Self-Balancing and Self-Organizing Data Structures

Although PSDS share conceptual similarities with self-balancing and self-organizing data structures, they differ fundamentally in their guarantees and methodological approach. Notably, self-organizing data structures have been extensively analyzed under adversarial models where input sequences are deliberately constructed to degrade performance, whereas the corresponding analysis for PSDS against adaptive adversaries remains a significant open problem. Similarly, self-balancing data structures have been studied extensively under worst-case analyses that inherently account for adversarial strategies.

Self-organizing data structures [26], whether randomized or deterministic, dynamically adjust their internal ordering of elements to optimize performance based on a given (potentially adversarial) sequence of input requests. For instance, self-organizing lists may employ the move-to-front heuristic, where accessed elements are relocated to the front of the list, or the transpose method, where elements swap positions with their predecessors when accessed. Similarly, splay trees [27] rotate frequently accessed nodes closer to the root to reduce future access times. This approach has been shown to be challenging in adaptive adversarial settings, with (randomized) self-organizing lists incurring a cost at least three times that of the optimal reordering strategy [28].

Self-balancing data structures, such as Red-Black trees [29] and AVL trees [30], *deterministically* ensure an upper-bound on node depth, thereby providing worst-case performance guarantees for search operations. This deterministic approach is also exemplified by the deterministic skip list [31], which enforces an optimal structure by carefully promoting inserted nodes and their neighborhoods to appropriate levels. While these structures guarantee bounded search path lengths (even in adversarial settings), they require complex re-balancing mechanisms. In steep contrast, PSDS, such as the treap [32] and the original skip list [33], offer comparable expected performance, achieved through simple, probabilistic updating mechanisms. This presents a clear trade-off: deterministic structures provide absolute performance guarantees at the cost of implementation complexity, while probabilistic alternatives offer simplicity, albeit, with only probabilistic guarantees. In this work, we investigate whether we can maintain the implementation simplicity of probabilistic data structures while preserving their performance guarantees even in adversarial settings.

2.4.2.2 Complexity Attacks Against Probabilistic Skipping-Based Data Structures

This section provides a concise overview of so-called *complexity attacks* targeting PSDS. Previous research has identified clear vulnerabilities in hash tables and skip lists, but these works lack formal security analysis and rigorous proofs of security when potential mitigations are put forth. Hash tables have received the most attention, while skip lists have been addressed (to our knowledge) in only a single paper in this context. Further, to our knowledge, no prior work has examined complexity attacks against treaps. This absence is consistent with our finding that treaps possess inherent resistance to such attacks.

Hash Tables. Assuming a hash table's internal hash function has "good" collision-resistance properties, the amortized average-case complexity of insertions, deletions, and look-ups is $O(1)$. For these efficiency reasons, hash tables are widely used in many applications such as implementing associative arrays [34] and sets [35] in many programming languages, in cache systems [36], as well as for database indexing [37].

However, this average-case performance relies on a critical assumption: that the data inserted into a hash table is independent of the (potentially randomly selected) hash function used to map key-value pairs to buckets. This assumption fundamentally breaks down in adversarial scenarios where an attacker can deliberately craft insertions that exploit knowledge of the hash function or its outputs. Given the ubiquity of hash tables in modern computing systems, numerous researchers [38, 13, 39, 40, 14, 15, 17] have investigated techniques to compromise the data structure, forcing operations to degrade from expected $O(1)$ to worst-case $O(n)$ time complexity, where n represents the total number of elements in the structure. These adversarial approaches typically constitute complexity attacks that strategically engineer inputs causing multi-collisions –

deliberately exploiting hash function properties to force numerous distinct keys into identical buckets.

Crosby and Wallach [13] demonstrated denial-of-service attacks via complexity attacks in applications using hash tables, such as the Bro intrusion detection system [38], by forcing collisions with weak, fixed hash functions. They suggested universal hashing [41] as a mitigation, though without any formal guarantees. Klink and Walde [14] showed similar CPU exhaustion attacks on web servers (e.g., PHP, ASP.NET, Java), only using a single carefully crafted HTTP request. Aumasson et al.[15] further revealed vulnerabilities in hash tables using non-cryptographic hash functions (like MurmurHash and CityHash[42]), proposing SipHash [15] as a secure alternative – which is widely adopted but lacks a holistic formal analysis as it comes to security of hash tables in adversarial settings. Complexity attacks have also been shown effective in causing denial-of-service against flow-monitoring systems [40]. Further, the use of salting was undermined by remote timing attacks [39]. Recently, Bottinelli et al. [17] found nearly a third of QUIC implementations vulnerable to similar attacks. Despite these works and many proposed defenses, no formal framework exists for the provable security of (keyed) hash tables against adaptive adversaries. We address this gap by introducing the first rigorous security model for this setting, along with formal proofs establishing bounds on adversarial runtime degradation.

Skip Lists. In the original skip list paper [33], it is noted that it is imperative to keep the internal structure of the skip list hidden. Otherwise, adversarial users could observe the levels of individual elements and delete any element at a level greater than zero (the bottom layer). This would degenerate the structure to a simple linked list and force worst-case run time ($O(n)$) on subsequent operations after these deletions occur.

Nussbaum and Segal [18] demonstrate that private internal structure alone fails to protect skip lists against this style of attack. They present a (remote) timing attack that correlates query response times with element heights, ultimately allowing adversaries to force all elements in the structure to the lowest level. Their adversarial model is notably limited: the adversary cannot access the internal skip list structure, the initial data collection is non-adversarially selected, and the original data collection must be preserved during the attack. While they propose a structure called the *splay skip list* as a countermeasure, their solution lacks formal security analysis. Our work presents a significantly stronger adversarial model and provides a construction with formal security guarantees. We give an extensive commentary on [18] and vulnerabilities below.

Nussbaum and Segal [18] show that keeping the internal structure of the skip list private is insufficient to protect against complexity attacks. We discuss their attack in more detail because it is instructive in light of how to model attacks and prove the properties of robust alternatives. Nussbaum and Segal present a timing attack that allows an adversary to discover the levels at which specific elements reside through a series of queries and, in turn, correlate the time it takes to answer a query on a given element with the height of that element. After the heights of the elements are discovered, the simple deletion attack can be mounted.

The specific attack they present includes several assumptions.

- The size of the collection represented by the structure, n, is known to the adversary, $\mathcal{A}$.
- Each node in the structure holds a unique value.
- The well-ordered universe $\mathcal{U}$ is known and is of size $O(n)$.
- The runtime of the search algorithm in the structure is consistent. That is, a search for the same value will yield the same runtime each time the search is executed.

Further, their adversarial model is the following.

- $\mathcal{A}$ is given a skip list containing some collection of data, D that was selected by some (non-adversarial) process.
- The adversary, $\mathcal{A}$ does not have access to the internal structure of the skip list at any point. $\mathcal{A}$ can only interact with the structure through oracles that provide search, insertion, and deletion functionality to the structure that is under attack.

- After the completion of the attack, $\mathcal{A}$ is required to have altered the skip list it interacts with such that it contains the original D represented by the structure (before any adversarial interaction occurs) and the level that all (or nearly all) the elements reside at is the first.

The attack in this setting works by first running the timing attack to discover the level at which the elements in the structure exist (and, on the first iteration, which elements from $\mathcal{U}$ are present in the structure). Then all elements with a level greater than zero (exist at high level than the initial later) are removed. This set of removed elements are reinserted. These steps are repeated until (nearly) all the elements in the structure reside at level zero and the original collection represented by the structure is conserved – thereby, degrading the representation of this collection to (nearly) a flat singly-linked list.

As a countermeasure, the splay skip list structure is presented [18]. The approach is to swap the levels of certain elements during a search query, thereby preventing the adversary from discovering information about the level where any particular element resides (as they are not fixed). The structure is believed to prevent the timing attack from being effective, but no formal analysis of the security of the structure is given.

We again note that the adversarial setting that is given in [18] is rather limited. It assumes the adversary does not have access to the internal structure of the skip list, nor the ability to control the initial collection of data the skip represents. Further, it requires the adversary to conserve the initial data collection D that the skip list represents before any adversarial interaction occurs. We present a much stronger adversarial model in our work and a construction that satisfies this definition.

The authors propose a new structure that is believed to prevent the timing attack they present; however, as previously stated, no formal security analysis is given. Indeed, the splay skip list is still vulnerable to attacks, as demonstrated by the following scenario. Consider a collection D of elements represented by a splay skip list, where a total order is defined on the universe in which D resides. Suppose there exists an element d such that $x_1 \leq d \leq x_2$ for every pair of elements $x_1, x_2 \in D$, where $x_1 \neq x_2$. For a specific order, $x_1 \leq d_1 \leq x_2 \leq d_2 \leq \ldots$

for $x_i \in D$ and $d_i \notin D$, an adversary can exploit this by conducting search queries for the intermediary elements d_i.

Unlike searches for elements $x_i \in D$, which would trigger the splay mechanism, searches for these intermediary elements $d_i \notin D$ bypass the splay security mechanism. The runtimes required to (not) find these intermediate nodes, however, still uniquely determine the height of elements contained in D.[1] After the discovery of the heights of the elements contained in D, the trivial deletion attack could be carried out as before.

[1] Compared to searching for elements $x_1, x_2, \ldots$ as described in the original attack, the runtimes for searching $d_1, d_2, \ldots$ only change by a constant factor (one extra step to find that the $d_i \notin S$) .

CHAPTER 3
COMPACT FREQUENCY ESTIMATORS IN ADVERSARIAL ENVIRONMENTS

In this chapter, we focus on CPDS 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 [43], network monitoring [44], recommendation systems [45], and approximate database queries [46], to name a few applications.

The Count-min Sketch (CMS) [23] and HeavyKeeper (HK) [47] 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. [48]. The HK structure is the CFE of choice in the RedisBloom module [46], a component of the Redis database system [49].

Of particular interest to us is the 2019 ACM SIGSAC work of Clayton, Patton, and Shrimpton [8] 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 CPDS, 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 compressing 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 $(\mathrm{fp}, \mathrm{cnt})$. 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.

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 an insertion of e into the structure and $\mathbf{Qry}(e)$ to get a point query on e for some element $e \in \mathcal{U}$.

$\mathbf{Atk}^{\text{err-fe[u,v]}}_{\Pi,\mathcal{U}}(\mathcal{A})$

$\vec{S} \leftarrow \emptyset; K \twoheadleftarrow \mathcal{K}$

$\mathsf{repr} \twoheadleftarrow \textsc{Rep}_K(\vec{S})$

$\mathsf{kv} \leftarrow \top; \mathsf{rv} \leftarrow \top$

$\textbf{if } u = 1 : \mathsf{kv} \leftarrow K$

$\textbf{if } v = 1 : \mathsf{rv} \leftarrow \mathsf{repr}$

$x \twoheadleftarrow \mathcal{U}$

$\text{done} \twoheadleftarrow \mathcal{A}^{\mathbf{Hash},\mathbf{Up},\mathbf{Qry}}(x, \mathsf{kv}, \mathsf{rv})$

$n_x \leftarrow \mathsf{qry}_x(\vec{S})$

$\hat{n}_x \leftarrow \textsc{Qry}_K(\mathsf{repr}, \mathsf{qry}_x)$

$\textbf{return } |\hat{n}_x - n_x|$

$\mathbf{Up}(\mathsf{up})$

$\mathsf{repr}' \twoheadleftarrow \textsc{Up}_K(\mathsf{repr}, \mathsf{up})$

$\vec{S} \leftarrow \mathsf{up}(\vec{S})$

$\mathsf{repr} \leftarrow \mathsf{repr}'$

$\textbf{if } v = 0 : \textbf{return } \top$

$\textbf{return } \mathsf{repr}$

$\mathbf{Qry}(\mathsf{qry})$

$\textbf{return } \textsc{Qry}_K(\mathsf{repr}, \mathsf{qry})$

$\mathbf{Hash}(X)$

$\textbf{if } X \notin \mathcal{X} : \textbf{return } \bot$

$\textbf{if } H[X] = \bot$

$\quad H[X] \twoheadleftarrow \mathcal{Y}$

$\textbf{return } H[X]$

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.

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, $\textsc{Rep}, \textsc{Up}, \textsc{Qry}$) is provided oracle access to a random oracle $\mathbf{Hash}\colon \mathcal{X} \to \mathcal{Y}$, 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 $T\colon \mathbb{Z} \to \mathbb{Z}$, 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.

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

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

$M \leftarrow \text{zeros}(k, m)$
for $x \in \mathcal{S}$
  $M \leftarrow \textsc{Up}_K(M, \mathsf{up}_x)$
return M

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

$(p_1, \ldots, p_k) \leftarrow R(K, x)$
for $i \in [k]$
  $M[i][p_i] \mathrel{+}= 1$
return M

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

$(p_1, \ldots, p_k) \leftarrow R(K, x)$
return $\min_{i \in [k]} \{M[i][p_i]\}$

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.

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 $\textsc{Qry}(\mathsf{qry}_x)$ 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 [23]. 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

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

// initialise $k \times m$ (fp,cnt) 2-d array
for $i \in [k]$
  $A[i] \leftarrow [(\star, 0)] \times m$
for $x \in \mathcal{S}$
  $A \leftarrow \textsc{Up}_K(A, \mathsf{up}_x)$
return A

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

$(p_1, \ldots, p_k) \leftarrow R(K, x)$
$\mathrm{fp}_x \leftarrow T(K, x)$
$\mathrm{cnt}_x \leftarrow 0$
for $i \in [k]$
  if $A[i][p_i].\mathrm{fp} = \mathrm{fp}_x$
    $\mathrm{cnt} \leftarrow A[i][p_i].\mathrm{cnt}$
    $\mathrm{cnt}_x \leftarrow \max\{\mathrm{cnt}_x, \mathrm{cnt}\}$
return cnt_x

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

$(p_1, \ldots, p_k) \leftarrow R(K, x)$
$\mathrm{fp}_x \leftarrow T(K, x)$
for $i \in [k]$
  if $A[i][p_i].\mathrm{fp} \notin \{\mathrm{fp}_x, \star\}$
    $r \twoheadleftarrow [0, 1)$
    if $r \leq d^{A[i][p_i].\mathrm{cnt}}$
      $A[i][p_i].\mathrm{cnt} \mathrel{-}= 1$
// overtake the counter if 0
  if $A[i][p_i].\mathrm{cnt} = 0$
    $A[i][p_i].\mathrm{fp} \leftarrow \mathrm{fp}_x$
// increase the count if $\mathrm{fp} = \mathrm{fp}_x$
  if $A[i][p_i].\mathrm{fp} = \mathrm{fp}_x$
    $A[i][p_i].\mathrm{cnt} \mathrel{+}= 1$
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 $\mathrm{fp}_y = T(K, y)$, 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 $\mathrm{fp}_x \leftarrow T(K, x)$, a point-query for x returns $\max \left\{A[i][p_i].\mathrm{cnt} | A[i][p_i].\mathrm{fp}= \mathrm{fp}_x, i \in [k]\right\}$, 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.[47] 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 [50], 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 $R(K, x) = (\mathbf{Hash}(\langle 1, K, x\rangle), \mathbf{Hash}(\langle 2, K, x\rangle, \ldots, \mathbf{Hash}(\langle k, K, x\rangle)))$, and for HK, we set

$R(K, x)[i] = \mathbf{Hash}(\langle \text{"cnt"}, i, K, x\rangle)$ and $T(K, x) = \mathbf{Hash}(\langle \text{"fp"}, k + 1, K, x\rangle)$. 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. [47], 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 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

[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.

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

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

Note that relation (3-1) fully defines z for any fixed values of $\Pr[\,L^r \leq z\,], m, k, r$. 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 $L^r \leq f_{m,k,r}(q_H, q_U, q_Q)$ 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

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

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

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

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.

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.

[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.

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^1_x 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 $\mathbb{E}[\mathsf{Err} \mid Q] \geq \sum_{\xi=1}^{\lfloor (q_U - Q)/k \rfloor} \Pr\left[\mathsf{Err} \geq \xi \mid Q, \mathsf{Cover}^1_x\right] \Pr\left[\mathsf{Cover}^1_x \mid Q\right]$ 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

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

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; $\mathbb{E}[\mathsf{Err}] = \left\lfloor \frac{q_U}{k} \right\rfloor \Pr\left[\mathsf{Cover}^1_x \mid Q=0\right]$. 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 $\mathbb{E}[\mathsf{Err}] \geq \lfloor \frac{q_U}{k} \rfloor \left(1 - (1 - 1/m)^{\frac{q_H}{k} - 1}\right)^k$. 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.

$\mathrm{CoverAttack}^{\mathbf{Hash},\mathbf{Up},\mathbf{Qry}}(x, K, \mathsf{repr})$

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

$\mathrm{FindCover}^{\mathbf{Hash}}(r, x, K)$

cover $\leftarrow \emptyset$; found $\leftarrow$ False
$\mathcal{I} \leftarrow \emptyset$; tracker $\leftarrow$ zeros(k)
// $R(K, x)[i] = \mathbf{Hash}(\langle i, K, x\rangle)$
$(p_1, p_2, \ldots, p_k) \leftarrow R(K, x)$
while not found
    if q_H **Hash**-queries made
        return $\emptyset$
    $y \twoheadleftarrow \mathcal{U} \setminus (\mathcal{I} \cup \{x\})$
    $\mathcal{I} \leftarrow \mathcal{I} \cup \{y\}$
    $(q_1, q_2, \ldots, q_k) \leftarrow R(K, y)$
    for $i \in [k]$
        if $p_i = q_i$ **and** tracker$[i] < r$
            cover $\leftarrow$ cover $\cup \{y\}$
            tracker$[i] += 1$
    if sum(tracker) $= rk$
        found $\leftarrow$ True
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^{Up,Qry}(x, ⊥, ⊥)
─────────────────────────────
cover ← FindCover^{Up,Qry}(x)
until q_U Up-queries made:
   for e ∈ cover: Up(e)
return done
```

```
MinUncover^{Up,Qry}(x, a′, 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^{Up,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. $\circ$

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 $\mathbb{E}\left[\delta_1\right]$ 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 $\delta_2 \le \min_{j \notin \mathcal{M}(\{z_1,z_2\})}\{M[j][p_j]\} - \left(\min_{j \notin \mathcal{M}(\{z_1\})}\{M[j][p_j]\} + 1\right)$. 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 $\min_{j \notin \mathcal{M}(\{z_1,z_2\})}\{M[j][p_j]\} \le 2\alpha + (\delta_1 + 1)$: 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 $\delta_2 \le 2\alpha$ 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 $q'_Q \le 1 + \ell|\vec{I}| + \sum_{i=1}^{\ell-1}(\delta_i + 1)$, and the expected gap between q'_U and q'_Q is

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

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 $\mathbb{E}[\mathsf{Err}] \geq \frac{q_U - 1 - \mathbb{E}[L^1]}{k} \approx \frac{q_U - mH_k}{k}$.

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,

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

```
FindCover^Up(r, x, 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] ≠ repr[i][j]
         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]
      for j ∈ [m]
         if repr'[i][j] ≠ repr[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 3-6. 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.

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 [51]. 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 $A[i][p_i].\mathrm{fp} = \mathrm{fp}_x$ given that at some point during the attack we had $A[i][p_i].\mathrm{cnt} = t$ with $A[i][p_i].\mathrm{fp} = \mathrm{fp}_{z_i}, z_i \neq x$. 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 $A[i][p_i].\mathrm{cnt} = t$ with $A[i][p_i].\mathrm{fp} = \mathrm{fp}_{z_i}, z_i \neq x$ 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., $A[i][p_i].\mathrm{fp} = \mathrm{fp}_x$ 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 keeps inserting *distinct* elements until the frequency estimate for x drops to 0, i.e, $A[i][p_i].\mathrm{fp} \neq \mathrm{fp}_x$ 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 $A[i][p_i].\mathrm{fp}$ 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$^{\mathbf{Hash},\mathbf{Up},\mathbf{Qry}}(x, K, \mathsf{repr})$

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

Get-t()

```
g(t) ← log_2(k · (q_U)^t d^(t(t+1)/2)) − log_2(p)
// find the roots of the negative quadratic polynomial g
t_1, t_2 ← FindRootsOf(g) // t_1 ≤ t_2
// set t so t ≥ 1 and g(t) < 0
if t_1 > 1 or t_2 < 1 : t ← 1
if t_2 > 1 : t ← ⌈t_2⌉
if t_2 = 1 : t ← 2
return t
```

FindCover$^{\mathbf{Hash}}(x, K)$

```
cover ← {}; found ← False
I ← ∅; tracker ← zeros(k)
// R(K, x)[i]
// = Hash(⟨"ct", i, K, x⟩)
(p_1, p_2, ..., p_k) ← R(K, x)
while not found
    if q_H Hash-queries made
        return ∅
    y ←$ U \ (I ∪ {x})
    I ← I ∪ {y}
    (q_1, q_2, ..., q_k) ← R(K, y)
    for i ∈ [k]
        if p_i = q_i
            // remove duplicates
            cover[i] ← y
            if tracker[i] < 1
                tracker[i] ← 1
    if sum(tracker) = k
        found ← True
// return the cover
return cover.values()
```

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

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.

```
CoverAttack^{Up,Qry}(x, ⊥, ⊥)
  FindInsertCover^{Up,Qry}(x)
  until q_U Up-queries made:
    Up(x)
  return done

Reintro^{Up,Qry}(x)
  // reintroduce target x
  while True
    if q_U Up- or q_Q Qry-queries made:
        return
    Up(x); a ← Qry(x)
    if a > 0 : return
  endwhile

FindInsertCover^{Up,Qry}(x)
  // insert ≤ k elements t times in a row
  cover ← ∅
  t ← Get-t()
  I ← ∅
  for i ∈ [1, 2, ..., k]
    Reintro^{Up,Qry}(x)
    while True
      if q_U Up- or q_Q Qry-queries made
        return
      y ←$ U \ (I ∪ {x})
      I ← I ∪ {y}
      Up(y); a ← Qry(x)
      if a = 0 :
        cover ← cover ∪ {y}
        for j ∈ [t] : Up(y)
        break
  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 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.

[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.

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

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

Let D_i be the set of rows j with $A[j][p_j].\mathrm{fp} = \mathrm{fp}_x$ (i.e. x owning the counter), and let $c_{i,j}$ be the values of $A[j][p_j].\mathrm{cnt}$ 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 $A[j][p_j].\mathrm{fp} \neq \mathrm{fp}_x$. 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

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

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

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

We expect $\Pr\left[\neg (D^t) \wedge |C| = k\right] \approx 1$ 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* 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

```
CoverAttack^{Up,Qry}(x, ⊥, 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^{Up}(x, 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.

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

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

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 $M \leftarrow \textsc{Up}^{\text{CMS}}_K(M, \mathsf{up}_x)$ and $A \leftarrow \textsc{Up}^{\text{HK}}_K(M, \mathsf{up}_x)$, 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) $\mathrm{HK}(x) \leq n_x \leq \mathrm{CMS}(x)$. If $\mathrm{CMS}(x) \neq \mathrm{HK}(x)$ 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 $\mathrm{CMS}(x) \neq \mathrm{HK}(x)$. To explain our row-by-row refinements, let us define two sets $I_x = \{i \in [k] \mid A[i][p_i].\mathrm{fp} = \mathrm{fp}_x\}$ and $\hat{I}_x = \{i \in [k] \mid A[i][p_i].\mathrm{fp} \neq \mathrm{fp}_x\}$, 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

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

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.

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

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

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

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

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

$\langle M, A \rangle \leftarrow \mathsf{repr}$
$(p_1, \ldots, p_k) \leftarrow R(K, x)$, $\text{fp}_x \leftarrow T(K, x)$
$\Theta_1, \Theta_2 \leftarrow \infty$
/ CMS only overestimates
$\text{cnt}_{\text{UB},x} \leftarrow \text{Qry}_K^{\text{CMS}}(M, \mathsf{qry}_x)$
/ HK only underestimates
$\text{cnt}_{\text{LB},x} \leftarrow \text{Qry}_K^{\text{HK}}(A, \mathsf{qry}_x)$
/ return upperbound if equal to lowerbound
if $\text{cnt}_{\text{UB},x} = \text{cnt}_{\text{LB},x}$
  return $\text{cnt}_{\text{UB},x}$
for $i \in [k]$
  / if never observed
  if $A[i][p_i].\text{fp} = \star$
    $\text{cnt}_{\text{UB},x} \leftarrow 0$
    return 0
  / upper bound adjustment
  / x does not own counter
  else if $A[i][p_i].\text{fp} \neq \text{fp}_x$
    $\Theta \leftarrow \dfrac{M[i][p_i] - A[i][p_i].\text{cnt} + 1}{2}$
    $\Theta_1 \leftarrow \min\{\Theta_1, \Theta\}$
  / x owns counter
  else if $A[i][p_i].\text{fp} = \text{fp}_x$
    $\Theta \leftarrow \dfrac{M[i][p_i] + A[i][p_i].\text{cnt}}{2}$
    $\Theta_2 \leftarrow \min\{\Theta_2, \Theta\}$
$\text{cnt}_{\text{UB},x} \leftarrow \lfloor \min\{\Theta_1, \Theta_2\} \rfloor$
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). $\text{Qry}_K^{\text{CMS}}$, Up_K^{CMS}, resp. Qry_K^{HK}, Up_K^{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.

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

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

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

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 $A[i][p_i]$.fp, and $A[i][p_i]$.cnt is the number (1) of plates on the stack. Inductively, assume that

$A[i][p_i].\mathrm{cnt} = c > 0$ and all c of the plates on the stack have the same label $A[i][p_i].\mathrm{fp}$. Say that the next insertion is z and $A[i][p_i].\mathrm{fp} = \mathrm{fp}_z$. By rule 2(a), we push an fp_z-plate on to the stack and increment $A[i][p_i].\mathrm{cnt} \leftarrow c + 1$. In this case, by assumption, it remains the case that all plates have the same label equal to $A[i][p_i].\mathrm{fp}$, and there are $c + 1$ of them. Alternatively, if $A[i][p_i].\mathrm{fp} \neq \mathrm{fp}_z$ then by rule 2(b) we pop the top plate and decrement $A[i][p_i].\mathrm{cnt} \leftarrow c - 1$. At this point, either the stack is empty and $A[i][p_i].\mathrm{cnt} = 0$, so by 2(b) we push an fp_z-plate and set $A[i][p_i].\mathrm{cnt} \leftarrow 1$ and $A[i][p_i].\mathrm{fp} \leftarrow \mathrm{fp}_z$; or the stack is not empty, and we take no further action. In the first case, the stack contains a single plate labeled with $A[i][p_i].\mathrm{fp}$ and the counter is 1; in the second, by assumption all plates on the stack are still labeled with $A[i][p_i].\mathrm{fp}$, and $A[i][p_i].\mathrm{cnt}$ 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 $A[i][p_i].\mathrm{fp} \neq \mathrm{fp}_x$. We claim that $A[i][p_i].\mathrm{cnt} = c > 0$ implies $\tilde{n} - n_x \geq A[i][p_i].\mathrm{cnt} - 1$. To see this, note that $A[i][p_i].\mathrm{cnt} = c > 0$ means that there are c plates labeled with $A[i][p_i].\mathrm{fp} \neq \mathrm{fp}_x$ on the stack associated to c insertions of elements in V_x^i with fingerprint $A[i][p_i].\mathrm{fp}$. 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 $A[i][p_i].\mathrm{cnt} = 1$, and $\tilde{n} - n_x \geq A[i][p_i].\mathrm{cnt} - 1$. Similarly, if $\tilde{n} - n_x = d > 0$ then $\tilde{n} - n_x \geq A[i][p_i].\mathrm{cnt} - 1$ as there are still $A[i][p_i].\mathrm{cnt}$ plates associated with insertions of elements in V_x^i that have never been popped off and at least $A[i][p_i].\mathrm{cnt} - 1$ of them correspond to insertions not popping off a plate of x.

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

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

which proves the lemma. $\square$

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 $A[i][p_i]$.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 $\sum_{y \in V_x^i} n_y \geq n_x - A[i][p_i]$.cnt occurrences of $y \in V_x^i$. This implies $M[i][p_i] \geq 2n_x - A[i][p_i]$.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 $A[i][p_i]$.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 $i \in I_x$, $A[i][p_i]$.fp = 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 $n_x - A[i][p_i]$.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 $M[i][p_i] \geq 2n_x - A[i][p_i]$.cnt and

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

$\square$

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 $n_x \leq \mathrm{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\}\rfloor \leq \mathrm{CMS}(x)$.

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 $\mathrm{CMS}(x) = n_x$ and $\mathrm{HK}(x) = n_x$. When this is not the case, $n_x < M[i][p_i]$ for all $i \in [k]$, so $n_x < \mathrm{CMS}(x)$. For CK, on the other hand, if there exists a row i such that $|V_x^i| = 1$, we still have $\mathrm{CK}(x) = n_x$. 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 $\mathrm{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\}\rfloor = n_x$.

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

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

Proof of corollary 1. We think of the counter $A[i][p_i].\mathrm{cnt}$ 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 $i \in \hat{I}_x$ $(A[i][p_i].\mathrm{fp} \neq \mathrm{fp}_z)$. 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 $A[i][p_i].\mathrm{cnt} - 1$ occurrences of z are not associated with x implying $n_z = n_x + A[i][p_i].\mathrm{cnt} - 1$. Hence, $M[i][p_i] - n_x = n_x + A[i][p_i].\mathrm{cnt} - 1$, or equivalently, $n_x = \frac{M[i][p_i] - A[i][p_i].\mathrm{cnt}+1}{2}$. 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 $n_z = n_x + A[i][p_i].\mathrm{cnt}$, and $n_x = \frac{M[i][p_i] - A[i][p_i].\mathrm{cnt}}{2}$.

Let now $i \in I_x$ $(A[i][p_i].\mathrm{fp} = \mathrm{fp}_x)$. 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* $\text{CMS}(x) = M[i^*][p_{i^*}]$. *If $i^* \in \hat{I}_x$ then either* $\text{CK}(x) = n_x$, *or* $\left(\Theta_1 \le \frac{\text{CMS}(x)}{2}\right)$. ♦

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

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* $\text{CMS}(x) = M[i^*][p_{i^*}]$. *If $i^* \in I_x$ then either* $\text{CK}(x) = n_x$ *or* $\left(\Theta_2 \le \frac{\text{CMS}(x)+\text{HK}(x)}{2}\right)$. ♦

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

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 $\mathrm{CK}(x) = 0 \le \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$. If neither of these holds, $\mathrm{CK}(x) = \lfloor \min\{\Theta_1, \Theta_2\} \rfloor$. Thus, Theorem 3-1 and 3-2 imply $\lfloor \min\{\Theta_1, \Theta_2\} \rfloor \le \frac{\mathrm{CMS}(x)+\mathrm{HK}(x)}{2}$, giving us the following lemma.

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

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* $\mathrm{CK}(x) - n_x \le \frac{\mathrm{CMS}(x)-\mathrm{HK}(x)}{2}$. ♦

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

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

□

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

Next, because CMS is a count-all structure, the *worst case* guarantee is that the error $\mathrm{CK}(x) - n_x \le \mathrm{CMS}(x)/2$, i.e., when $\mathrm{HK}(x) = 0$. 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 $\mathrm{CMS}(x) = \min_{i \in [k]}\{M[i][p_i]\}$, 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 $\mathrm{CMS}(x)$ be very large when $\mathrm{HK}(x)$ 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 [47, 52, 53, 54, 23, 55, 56]. 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

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

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.

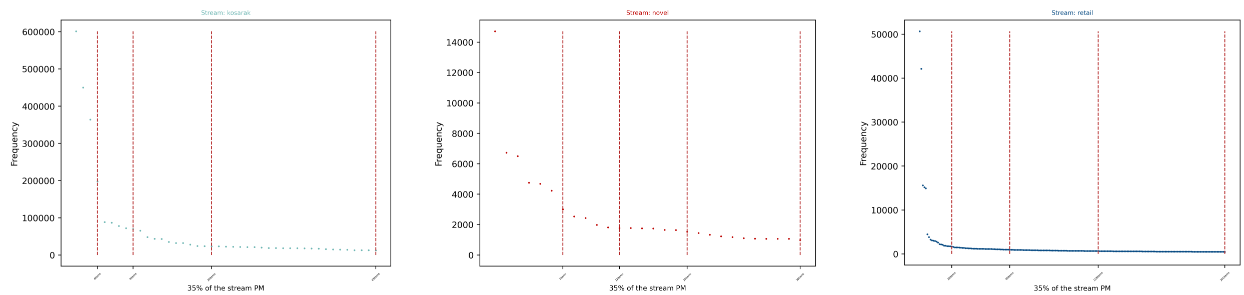
2. **Novel Stream:** We created a stream by processing the individual words sequentially of The Project Gutenberg eBook plaintext edition of the 1851 English-language novel *Moby-Dick; or, The Whale* by Herman Melville (ignoring capitalization and non-alphabetical characters) [57]. Long bodies of natural language obey an approximate Zipf distribution as the frequency of any word is inversely proportional to its rank in an ordered frequency list [58]. It is of interest to measure compact frequency estimators performance against data following a Zipf distribution [55, 59, 47, 53, 54, 56]. 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.

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



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.

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 [60] 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 [47, 52, 54].

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 [47] 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 [61]. 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 [55].

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 [62]. 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 [47]. It is defined as 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

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

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 [46] and satisfies the desired properties of the exponential decay function stated in [47].

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)	Kosarak ($K = 20$)	19.303	0.934	20.901	0.017
		Novel ($K = 22$)	22.999	0.999	22.001	0.009
		Retail ($K = 22$)	21.643	0.997	22.405	0.040
HK	(1024,4)	Kosarak ($K = 20$)	20	1	20	≈ 0
		Novel ($K = 22$)	22	1	22	≈ 0
		Retail ($K = 22$)	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)	Kosarak ($K = 20$)	18.241	0.841	24.567	0.125
		Novel ($K = 22$)	21.638	0.969	22.473	0.062
		Retail ($K = 22$)	18.745	0.745	41.609	0.296
HK	(256,4)	Kosarak ($K = 20$)	20	1	20	≈ 0
		Novel ($K = 22$)	22	1	22	0.001
		Retail ($K = 22$)	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 for target x using the **Hash** oracle only, and then accumulate error for the target by repeatedly

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

```
CoverAttack^{Hash,Up,Qry}(x, K, repr)
-------------------------------------
cover ←FindCover^{Hash}(2, x, K)
until q_U Up-queries made:
   for e ∈ cover: Up(e)
return done
```

```
FindCover^{Hash}(r, x, K)
-------------------------
cover ← ∅; found ← False
I ← ∅; tracker ← zeros(k)
// R(K, x)[i] = Hash(⟨i, K, x⟩)
(p_1, p_2, ..., p_k) ← R(K, x)
while not found
   if q_H Hash-queries made
      return ∅
   y ←$ U \ (I ∪ {x})
   I ← I ∪ {y}
   (q_1, q_2, ..., q_k) ← R(K, y)
   for i ∈ [k]
      if p_i = q_i and tracker[i] < r
         cover ← cover ∪ {y}
         tracker[i] + = 1
   if sum(tracker) = rk
      found ← True
return cover
```

Figure 3-12. 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.

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 $\mathbb{E}[\mathsf{Err}] \geq \left\lfloor \frac{q_U}{2k} \right\rfloor \Pr\left[L^2 \leq \frac{q_H - k}{k}\right]$ 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,

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

MinUncover^{Up,Qry}(x, a′, cover)
  b′ ← a′ − 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^{Up,Qry}(x)
  / find 2- cover for x
  cover ← ∅
  found ← False
  I ← ∅; a ← Qry(x)
while not found
    if q_U Up- or q_Q Qry-queries made
      return cover
    y ←$ U \ (I ∪ {x})
    I ← I ∪ {y}
    Up(y); a′ ← Qry(x)
    if a′ ≠ a :
      cover ← {y}
      found ← True
  for i ∈ [2, 3, . . . , 2 · k]
    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}
        I ← I \ {y}
        break
  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 $\mathbb{E}[\mathsf{Err}]$:

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

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	$\lvert$cov$\rvert$	Exp. Err	$\mathbb{E}[\mathsf{Err}]$	$\lvert$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 adversarial resources such that $q_H, q_U, q_Q = 2^{20}$. In practice this ensures that the number of **Hash**

```
CoverAttack^{Up,Qry}(x, ⊥, repr)

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

```
FindCover^{Up}(r, x, 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.

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 $A[i^*][p_{i^*}].\text{cnt} \approx 0$, observe that whether 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 $i \in [k]$
    $A[i] \leftarrow [(\star, 0)] \times m$
$\mathsf{repr} \leftarrow \langle M, A \rangle$
for $x \in \mathcal{S}$
    $\mathsf{repr} \twoheadleftarrow \textsc{Up}_K(\mathsf{repr}, \mathsf{up}_x)$
return repr

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

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

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

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

Figure 3-15. Keyed structure 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. $\textsc{Qry}^{\text{CMS}}_K, \textsc{Up}^{\text{CMS}}_K$, resp. $\textsc{Qry}^{\text{HK}}_K, \textsc{Up}^{\text{HK}}_K$, denote query and update algorithms of keyed structure CMS$[R, T, m, k]$ (Figure 3-2), resp. 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
COMPACT PROBABILISTIC DATA STRUCTURES IN THE WILD: A SECURITY ANALYSIS OF REDIS

As we have seen, compact probabilistic data structures are becoming ubiquitous in modern computing applications that deal with large amounts of data, especially when the data is presented as a stream. Many modern data warehousing and processing systems provide access to CPDS 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 CPDS: HyperLogLog (HLL), Bloom filter, Cuckoo filter, t-digest, Top-K, and count-min sketch [63]. While Redis was mostly used as a cache in the past, it is now a fully general system, used by a companies like Adobe [64], Microsoft [65], Facebook [66] and Verizon [67] for a variety of purposes. These include security-related applications, such as traffic analysis and intrusion detection systems [68].

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"* [69]. In reality, users failed to comply with this [70]. 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 [71]. 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 [71]. 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 CPDS implementations.

Given its prominence in the marketplace and the many other systems that rely on it, we contend that the CPDS used in Redis are deserving of detailed analysis. Moreover, in view of the broad set of use cases for these CPDS, 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 CPDS analysis [72, 8, 73, 74, 11, 75]. In this paper, we make a comprehensive security analysis of the suite of CPDS provided by Redis, with a view to understanding how its constituent CPDS perform in adversarial settings. As argued in [76], 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 CPDS (eg. via the presented API). The adversary's aim is then to subvert the main goal of the specific CPDS 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 CPDS functionalities in Redis and how their goals can be subverted in general. Naturally, our analyses are specific to each of the different CPDS 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 CPDS in Redis.

Since HLL in Redis was already comprehensively studied in [11], we do not consider it further here. We note only that [11] 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 [77]; it uses a k-means clustering technique [78] to estimate percentiles over a collection of measurements. The structure is an outlier in the Redis CPDS 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 CPDS in Redis (in particular it does not employ the “hash functions mapping to array positions” paradigm that the other CPDS 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 CPDS in Redis: Bloom filter, Cuckoo filter, Top-K, and count-min sketch. For each CPDS, we discuss how the CPDS 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 CPDS. We compare our attacks with known attacks for these CPDS from the literature. We also look at how the CPDS in Redis can be protected against attacks, drawing on existing literature that considers this question for CPDS more generally [8, 24, 11, 75]. 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). Recall, that in Chapter 3 we saw that count-min sketch and HeavyKeeper (called Top-K in Redis) are susceptible to devastating attacks in even limited adversarial settings. In this chapter we demonstrate how these attacks are augmented by the specific implementation choices that Redis makes.Full details on the Bloom filter and Cuckoo filter are available in the full paper [79].

Further, we notified Redis of our findings on 29.04.2024. The full version of our paper [79] 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' CPDS 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 theC 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. We depart from our use of the generic data structure's syntax of [8], instead using an ad-hoc syntax that better matches how the structures are defined and implemented in Redis.

4.1.1 Count-min Sketches

CMS.setup(pp)

$\varepsilon, \delta \leftarrow pp$

$m \leftarrow \left\lceil \frac{e}{\varepsilon} \right\rceil$

$k \leftarrow \left\lceil \ln(\frac{1}{\delta}) \right\rceil$

$h(\circ) \leftarrow \mathit{MurmurHash2}(\circ) \bmod m$

$\sigma \leftarrow \mathrm{zeros}(k, m)$

return $\top$

CMS.ins(x, σ, v)

$(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$

for $i \in [k]$

  $\sigma[i][p_i] += v$

return $\min_{i \in [k]}\{\sigma[i][p_i]\}$

CMS.qry(x, σ)

$(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$

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 $\varepsilon, \delta \in (0, 1)$ 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 [23], a count-min sketch consists of a $k \times m$ array σ of (initially zero) counters, and k pairwise independent hash functions $h_1, \ldots, h_k$ 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 [23] 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 $\mathsf{CMS.setup}(\varepsilon, \delta)$. 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 $\mathsf{CMS.ins}(x, \sigma, v)$ and $\mathsf{CMS.qry}(x, \sigma)$; 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. $h_1(x) \leftarrow h(x, 1), \ldots, h_k(x) \leftarrow h(x, k)$, 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 [23]. We will leverage this and the properties of *MurmurHash2* in our attacks to cause large frequency overestimates.

4.1.2 Top-K

```
TK.setup(pp)
  m, k, decay, K ← pp
  seed ← 1919
  h(∘) ← MurmurHash2(∘) mod m
  h_fp ← MurmurHash2(∘)
  for i ∈ [k]
    σ[i] ← [(★, 0)] × m
  H ← initminheap(K)
  return ⊤
```

```
TK.qry(x, σ)
  (p_1, ..., p_k) ← h(x, 1), ..., h(x, k)
  fp_x ← h_fp(x, seed)
  cnt_x ← 0
  for i ∈ [k]
    if σ[i][p_i].fp = fp_x
      cnt ← σ[i][p_i].cnt
      cnt_x ← max {cnt_x, cnt}
  return cnt_x
```

```
TK.list(σ)
  T ← H.list()
  return T
```

```
TK.ins(x, σ)
  r ← nil
  (p_1, ..., p_k) ← h(x, 1), ..., h(x, k)
  fp_x ← h_fp(x, seed)
  cnt_x ← 0
  for i ∈ [k]
    if σ[i][p_i].fp ∉ {fp_x, ★}
      r ←$ [0, 1)
      if r ≤ decay^(σ[i][p_i].cnt)
        σ[i][p_i].cnt −= 1
    if σ[i][p_i].cnt = 0
      σ[i][p_i].fp ← fp_x
    if σ[i][p_i].fp = fp_x
      σ[i][p_i].cnt += 1
      if σ[i][p_i].cnt > cnt_x
        cnt_x ← σ[i][p_i].cnt
  if cnt_x ∈ H
    H.update(x, cnt_x)
  elseif cnt_x > H.getmin()
    r ← H.getmin()
    H.poppush(x, cnt_x)
  return r
```

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 initialized with $pp = m, k, \text{decay}, K$ as TK[m, k, decay, K].

A Top-K structure, originally introduced as the HeavyKeeper in [47], 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 [60] 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 [52, 54]. 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 $(\mathrm{fp}, \mathrm{cnt})$ 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 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 $\mathrm{cnt}_x \leftarrow 0$. We then go row by row (indexed by $i \in [k]$), with the following cases:

1. **if** $\mathrm{fp}^* = \star$, 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 $\mathrm{cnt}_x < 1 : \mathrm{cnt}_x \leftarrow 1$.

2. **else if** $\mathrm{fp}_x = \mathrm{fp}^*$, we add 1 to the counter value, and if $\mathrm{cnt}_x < c : \mathrm{cnt}_x \leftarrow c$, 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 $r < \mathrm{decay}^c$, 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 $\mathrm{cnt}_x < 1 : \mathrm{cnt}_x \leftarrow 1$. 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 [47], 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 [75] for a meaningful result). Moreover, the results in [47] 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 [75], 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 TK.list(σ). (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 [79]). 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 [80].

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 [81]. 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 [82]. 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 $\{y_1,\ldots,y_k\}$ 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 *MurmurHash2*$(\cdot, i)$ as in Redis. We then set y_i by inverting *MurmurHash2*$(\cdot, i)$ at x for $i \in [k]$. Respecting Redis'

```
overestimation_attack^Ins(x, pp, I)

  cover ← find_cover(x, pp)
  until I insertions are made
    for e ∈ cover: Ins(e)
  return done

find_cover(x, pp)

  ε, δ ← pp
  k ← ⌈ ln(1/δ) ⌉
  cover ← ∅
  (p_1, ..., p_k) ← h(x, 1), ..., h(x, k)
  for i ∈ [k]
    y ← MurmurHash2Inverse(p_i, i)
    cover ← cover ∪ {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.

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.

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.

ϵ, δ (m, k)	Ours	[75]
$2.7 \times 10^{-3}, 1.8 \times 10^{-2}$ $(1024, 4)$	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 [75] given in Section 3.4.2, namely kmH_k.

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 [75] 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 [75]. 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.

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 $\text{TK}[m, k, \text{decay}, K]\ \sigma$, 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 $\textbf{Ins}(\cdot) := \mathsf{TK.ins}(\cdot, \sigma)$ and $\textbf{Qry}(\cdot) := \mathsf{TK.qry}(\cdot, \sigma)$. 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 [75] 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 $\textbf{Ins}(\cdot)$ 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 [83]. 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. 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 **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

$\text{known\_F\_attack}^{\mathbf{Ins}}(F, n, p, pp)$

$t \leftarrow \text{get\_t}(n, p, pp)$
$\text{F\_cover} \leftarrow \text{find\_F\_cover}(F, pp)$
for $e \in \text{F\_cover}$
  for $i \in [t]$
    $\mathbf{Ins}(e)$
return done

$\text{get\_t}(n, p, pp)$

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

$\text{find\_F\_cover}(F, pp)$

$m, k, \text{decay}, K \leftarrow pp$
$\text{F\_cover} \leftarrow \emptyset$
for $f \in F$
  $(p_1, \ldots, p_k) \leftarrow h(f, 1), \ldots, h(f, k)$
  for $i \in [k]$
    $y \leftarrow \mathit{MurmurHash2Inverse}(p_i, i)$
    $\text{F\_cover} \leftarrow \text{F\_cover} \cup \{y\}$
return F_cover

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

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.

```
hidden_F_attack^Ins(n, p, pp)
-----------------------------
t ← 500
S_cover ← find_S_cover(pp)
for e ∈ S_cover
    for i ∈ [t]
        Ins(e)
return done

find_S_cover(pp)
-----------------------------
m, k, decay, K ← pp
η ← zeros(k, m)
S_cover ← ∅
for i ∈ [k]
    for j ∈ [m]
        if η[i][j] = 0
            y ← MurmurHash2Inverse(j, i)
            S_cover ← S_cover ∪ {y}
            (p_1, ..., p_k) ← h(y, 1), ..., h(y, k)
            for r ∈ [k]
                η[r][p_r] ← 1
return S_cover
```

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

4.2.3.3 NFC assumption violation attack

Consider a Top-K structure with parameters m, k, decay, K. After initializing TK$[m, k, \text{decay}, K]$ σ, an adversary $\mathcal{A}$ is given access to insertion and query oracles: **Ins**$(\cdot)$:= $\mathsf{TK.ins}(\cdot, \sigma)$ and **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.

(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

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 [79].

```
nfc_violation_attack^Ins(x, pp, I)
-----------------------------------
y ← find_cover_element(x, pp)
until I insertions are made
    Ins(y)
return done

find_cover_element(x, pp)
-----------------------------------
m, k, decay, K ← pp
seed ← 1919
done ← ⊥
P ← (h(x, 1), . . . , h(x, k))
fp_x ← h_fp(x)
while done = ⊥
    y ← MurmurHash2Inverse(fp_x, seed)
    C ← (h(y, 1), . . . , h(y, k))
    for i ∈ [k]
        if P[i] = C[i]
            done ← ⊤
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.

Protecting the count-min sketch and Top-K against frequency estimation attacks is challenging. Recall, that both the count-,in 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 [75]. 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 CPDS 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 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 [75]. 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.

CHAPTER 5
PROVABLY ROBUST PROBABILISTIC SKIPPING-BASED DATA STRUCTURES

In this chapter we consider a distinct subset of probabilistic data structures, which ensures correctness (and hence are not compressing) while offering fast probabilistic runtime guarantees, have received considerably less attention in the literature. Existing security analyses, such as those addressing the robustness of hash tables [13, 15, 39, 40, 14, 17] and skip lists [18], provide valuable insights but lack formal adversarial models and rigorous security analyses. Due to their runtime properties, we refer to these as *probabilistic skipping-based data structures* (PSDS), as they inherently "skip" over parts of their internal structure to accelerate lookup operations. The lack of research in this area is particularly concerning given that the studies on hash tables have already uncovered practical attacks, including methods to mount denial-of-service attacks attacks against intrusion detection systems [39], web application servers [14], and the QUIC protocol [17].

5.1 Structures we Analyze

We give pseudocode and textual description of the probabilistic skipping-based data structures we consider in this work: hash tables, skip lists and treaps.

5.1.1 Hash Tables

We give a pseudocode description of a hash table (HT) in Figure 5-1. Elements consist of a pair of entries (x, v) of a unique index value x and the value v. An instance of HT consists of b buckets, each containing an (initially empty) linked list L, and a mapping $\textsc{Hash}(K, \cdot)$ from the index value x to the bucket number in $[b]$.

An index-value pair (x, v) is inserted into the HT representation by computing $\textsc{Hash}(K, x){=}i$ and traversing it to the i-th bucket. We then check if the pair is already in the linked list $\mathsf{L}[i]$ stored and delete the prior mapping if this is the case. This is necessary since we insert elements according to the index key x, and the value entry v may have changed in the new request. Finally, we insert the new pair into $\mathsf{L}[i]$. Likewise, a key is deleted by searching in the bucket to which it is assigned and removing the key and its associated value from the linked list $\mathsf{L}[i]$ in the bucket if this pair exists there. Traditionally, it is assumed that $i = \textsc{Hash}(K, x)$, where $\textsc{Hash}$ is a fast-to-compute hash function with good (enough) collision resistance properties. However, we

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

```
for i ← 1 to m do
    T[i] ← new L
for (x, v) ∈ S
    T ← Up_K(T, ins_(x,v))
return T
```

$\text{Up}_K(T, \mathsf{ins}_{(x,v)})$

```
v′ ← Qry_K(T, qry_x)
if v′ ≠ ⋆
    Up_K(T, del_x)
i ← Hash(K, x)
T[i].insert((x, v))
return T
```

$\text{Up}_K(T, \mathsf{del}_x)$

```
i ← Hash(K, x)
T[i].remove(x)
return T
```

$\text{Qry}_K(T, \mathsf{qry}_x)$

```
v ← ⋆
i ← Hash(K, x)
v′ ← T[i].find(x)
if v′ ≠ null
    v ← v′
return v
```

Figure 5-1. A possibly keyed hash-table structure $\mathrm{HT}[\text{Hash}_K, b]$ admitting insertions, deletions, and queries for any $k \in \mathcal{U}_\kappa$ and its associated value v. The parameters are an integer $b \geq 1$, and a keyed function $\text{Hash} : \mathcal{K} \times \mathcal{U}_\kappa \to [b]$ that maps the key part of key-value pair data-object elements (encoded as strings) to a position in the one of the table buckets $v.T$. A particular choice of parameters gives a concrete scheme. Each bucket contains a simple linked list L equipped with its usual operations insert, find, and remove for insertion, searching, and deletion. If an item is not contained in the map, the distinguished symbol $\star$ is returned.

generalize here to make the exposition cleaner and allow the mapping to depend upon secret randomness (i.e., a key K). To query a key for its associated value, the algorithm $\textsc{Qry}(\mathsf{qry}_x)$ searches the bucket x maps to and returns the index-value pair if it exists there; otherwise, we return the distinguished null symbol $\star$.

Hash tables are widely adopted for their $O(1)$ amortized average-case complexity for insertions, deletions, and look-ups, assuming "good" collision-resistance properties in the internal hash function. This efficiency has led to their extensive use across various applications, including implementations of associative arrays [34] and sets [35] in programming languages, cache systems [36], and database indexing [37]. However, despite their performance advantages, hash tables have inherent functional limitations—they cannot efficiently support operations that depend on order relationships between keys, such as range queries, predecessor/successor lookups, or sorted traversals, restricting their applicability in scenarios where such operations are essential.

5.1.2 Skip Lists

In Figure 5-2, we give a pseudocode description of the skip list (SL). SL maintains an ordered collection of data that allows for average-case runtime $O(\log n)$ for search, insertions, and deletions (where the size of the represented collection is n). The structure is maintained as a hierarchy of linked lists, with the first level containing all the elements of the collection and each higher level in the structure skipping over an increasing number of elements. Searching (as well as insertions and deletions) starts at the highest level, only moving down to lower levels as necessary. The specific elements that are skipped at each level are determined either probabilistically or deterministically (using (say) a PRF) at insertion time – we focus on the probabilistic version of this structure in this paper. For a full structure description, we point to the original paper [33].

Skip lists provide an elegant probabilistic alternative to balanced binary search trees. They are widely deployed in industry applications – managing millions of Discord server members [84], storing data in Apache Web Servers [85], and indexing SingleStore databases [86]. Unlike hash tables, skip lists efficiently support range queries, ordered traversals, and predecessor/successor operations, making them valuable for various applications [87, 88, 89].

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

```
h ← NewNode(m, ⋆)
L.header ← h,  L.level ← 1
for (x, v) ∈ S
   L ← Up_K(L, ins_(x,v))
return L
```

NewNode(ℓ, (x, v))

```
   / array position 0 is reserved for a key, value pair (x, v)
   / accessible via n.key and n.value
   / array positions 1 . . . ℓ are forward pointers
   / level is accessible via n.level
node ← new [0, .., ℓ]
node[0] ← (x, v)
for i ← ℓ downto 1 do
   node[i] ← null
return node
```

RandomLevel$_K$($\boxed{x}$)

```
[ℓ ← R(K, x, m, p)]
[return ℓ]
ℓ ← 1,  r ↞ [0, 1)
while r < p and ℓ < m do
   ℓ ← ℓ + 1,  r ↞ [0, 1)
return ℓ
```

Qry(L, qry$_x$)

```
c ← L.header
for i ← L.level downto 1 do
   while c[i] ≠ null and c[i][0].key < x do
      c ← c[i]
c ← c[1]
if c ≠ null and c[0].key = x then
   return c[0].value
else
   return ⋆
```

Up$_K$(L, ins$_{(x,v)}$)

```
u ← new [1, .., m]    / local array of pointers
c ← L.header
for i ← L.level downto 1 do
   while c[i] ≠ null and c[i][0].key < x do
      c ← c[i]
   u[i] ← c
c ← c[1]
if c ≠ null and c[0].key = x then
   c[0].value ← v
   return L
else
   ℓ ← RandomLevel_K([x])
   if ℓ > L.level then
      for i ← L.level + 1 upto ℓ do
         u[i] ← L.header
      L.level ← ℓ
n ← NewNode(ℓ, (x, v))
for i ← 1 upto ℓ do
   n[i] ← u[i][i],  u[i][i] ← n
return L
```

Up(L, del$_x$)

```
u ← new [1, .., m]    / local array of pointers
c ← L.header
for i ← L.level downto 1 do
   while c[i] ≠ null and c[i][0].key < x do
      c ← c[i]
   u[i] ← c
c ← c[1]
if c ≠ null and c[0].key = x then
   for i ← 1 upto c.level do
      u[i][i] ← c[i]    / free c
   while L.level > 1 and L.header[L.level] = null do
      L.level ← L.level − 1
return L
```

Figure 5-2. A possibly "deterministic" (and keyed) skip list structure SL[$\boxed{R}$, m, p] admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameters are an integer $m \geq 0$ representing the maximum level of the structure, a fraction $p \in (0, 1)$ used for determining an element's random level, and, if using the deterministic version of the structure, a keyed function $R : \mathcal{K} \times \mathcal{U} \times \mathbb{Z}^{+} \times (0, 1) \to [m]$ that maps an element to a level in accordance with the distribution imposed by m and p. A concrete scheme is given by a particular choice of parameters. Subroutines used by the deterministic version of the structure appear in the boxed environment.

5.1.3 Treaps

In Figure 5-3 , we give a pseudocode description of the treap (TR). A treap [90] combines the algorithms of a binary search tree (BST) and a heap and achieves an expected height of $O(\log n)$ [90]. Inserting a node into a treap works analogously to a BST, but the node gets assigned an additional random priority value. Subsequently, the algorithm rotates the tree to maintain a heap order amongst the priority values without affecting the key ordering. For instance, in a MIN heap, the parent nodes are guaranteed lower priority values than their children. Intuitively, when interpreting the priority values as timestamps, the resulting treap will correspond to a binary search tree in which all nodes have been inserted in random order (i.e., a randomized binary search tree). Deletion first rotates a node down the heap without affecting the key ordering and then removes it once it reaches a leaf position.

Treaps efficiently support the full spectrum of binary tree operations, including range queries, predecessor/successor lookups, in-order traversals, and advanced tree operations like join, split, and union. This versatility has made treaps valuable in applications where search efficiency and ordered operations are critical requirements, such as implementing retroactive data structures [91].

5.2 Unifying Probabilistic Skipping-Based Data Structures

Informally, one can think of a probabilistic skipping-based data structure as a data structure that uses some form of randomness (either fixed at initialization time or freshly sampled per operation) to distribute the underlying collection within its representation. This randomized representation is to (generally) allow for efficient search by "skipping" over some elements, such that the resulting expected runtime is sublinear with high probability.

For instance, hash tables employ a hash function to "randomly" map elements to buckets, and therefore, one only has to search in this bucket for a desired element. Likewise, skip lists randomly assign heights to elements to facilitate "skipping" over a sequence of elements while performing a search. While the treap randomly assigns priority values to maintain an (approximately) balanced tree representation. In turn, the hash table achieves non-adaptive

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

```
T.root ← null
for (x, v) ∈ S do
    T ← Up_K(T, ins_(x,v))
return T
```

$\textsc{RandomPriority}_K(\boxed{x})$

```
[ p ← R(K, x) ]
[ return p ]
p ↞ (0, 1)
return p
```

$\textsc{NewNode}((x, v), p)$

```
    / array position 0 is reserved for a key, value pair (x, v)
    / accessible via n.key and n.value
    / array positions 2, 3 are child pointers and 1 is priority
node ← [(x, v), p, null, null]
return node
```

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

```
T.root ← Qry^rec(T.root, qry_x)
return T
```

$\textsc{Qry}^{\mathrm{rec}}(c, \mathsf{qry}_x)$

```
if c = null then
    return ⋆
if c[0].key = x then
    return c[0].key
b ← (x > c[0].key)
return Qry^rec(c[2 + b], qry_x)
```

$\textsc{Rotate}(c, b)$

```
tmp ← c[2 + b][3 − b]
c[2 + b][3 − b] ← c
c[2 + b] ← tmp
return tmp
```

$\textsc{Up}_K(\mathsf{T}, \mathsf{ins}_{(x,v)})$

```
T.root ← Up_K^rec(T.root, ins_(x,v))
return T
```

$\textsc{Up}_K^{\mathrm{rec}}(c, \mathsf{ins}_{(x,v)})$

```
if c = null then
    p ← RandomPriority_K([x])
    return NewNode((x, v), p)
if c[0].key = x then
    c[0].value ← v
    return c
b ← (x > c[0].key)
c[2 + b] ← Up_K^rec(c[2 + b], ins_(x,v))
    / maintain MIN Heap property
if c[1] > c[2 + b][1] then
    c ← Rotate(c, b)
return c
```

$\textsc{Up}_K(\mathsf{T}, \mathsf{del}_x)$

```
T.root ← Up_K^rec(T.root, del_x)
return T
```

$\textsc{Up}^{\mathrm{rec}}(c, \mathsf{del}_x)$

```
if c = null then
    return null
if c[0].key = x then
    / Remove node
    if c[2] = null and c[3] = null then
        return null
    if c[2] = null then
        return c[3]
    if c[3] = null then
        return c[2]
    / Rotate node down before removing
    b ← c[3][1] > c[2][1] then
    c ← Rotate(c, b)
    c[3 − b] ← Up_K^rec(c[3 − b], del_x)
else
    b ← (x > c[0].key)
    c[2 + b] ← Up_K^rec(c[2 + b], del_x)
return c
```

Figure 5-3. A possibly "deterministic" (and keyed) MIN treap structure TR[$\boxed{R}$] admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameter is a keyed function $R : \mathcal{K} \times \mathcal{U} \to (0, 1)$ that assigns an element a random priority. Subroutines used by the deterministic version of the structure appear in the boxed environment. Let $\textsc{MinPrioChild}(c)$ denote the function that returns the child index (0 or 1) of node c with the minimum priority, or null if c has no children.

adversarial expected runtime $O(1)$ for insertions, deletions, and search; similarly, the skip list and treap achieve non-adversarial expected runtime $O(\log n)$ for these operations on an ordered collection (but has other advantages such as supporting range queries).

In contrast to compressing probabilistic data structures (e.g., Bloom filters, count-min sketches, HyperLogLogs, etc.), PSDS always return a correct QRY response. Further, unlike self-balancing data structures (e.g., splay trees, red-black trees, sorted arrays, etc.), skipping data structures do not require complex update mechanisms to maintain favorable representations. That is, under non-adversarial conditions, using randomness is sufficient to facilitate efficient operational runtimes (with high probability) without the overhead of complex and potentially expensive rebalancing algorithms.

While this provides an intuitive notion of a skipping-based data structure, it fails to provide a formal or constructive definition. Therefore, let us consider the following. Take a hash table, whose representations are built over a size n set of elements (index keys) from the domain $\{0, 1\}^n$ by running them each through a hash function and putting them into a bucket depending on the output of this hash function. Under the assumption that the hash function is uniform and the non-adversarial assumption that the set of elements is selected uniformly at random from the universe of all elements, then the elements in the table can be viewed as an (unordered) sequence of i.i.d. random variables. That is, we can decompose a hash table's representation as $B_1, B_2, \ldots, B_N$ where $\forall i \in [n] : B_i \sim \mathsf{U}(\{1, 2, \ldots, b\})$, where b is the number of buckets for the particular structure.

For a skip list, we can take a similar view. Here, we again assume that a skip list represents a size n set of elements from the domain $\{0, 1\}^n$. Additionally, we assumed that the set is well-ordered. Under the non-adversarial assumption that all updates are made uniformly at random from the universe of all elements, the representation can be viewed as a sequence of ordered i.i.d. random variables (again, in the adaptive adversarial setting independence of these random variables does not necessarily hold). We can decompose the skip list representation as $H_1, H_2, \ldots, H_N$ where $\forall i \in [n] : H_i \sim \mathsf{G}(p)$ for the geometric distribution, where p is the probability parameter of the structure. That is, a skip list can be viewed as the ordered sequence of its elements heights. The sequence of random variables $H_1, H_2, \ldots, H_N$ (heights) is sorted according to the order of the keys in the representation. Similarly, one can decompose the treap representation as $P_1, P_2, \ldots, P_N$ where $\forall i \in [n] : P_i \sim \mathsf{U}([0, 1])$. This sequence of random variables represents the priority of elements in the treap, and the sequence is again ordered by the keys in the representation. That is, treaps can be viewed as a binary search tree where the order of insertion is determined by the randomly sampled priorities [32].

With this intuition built, we arrive at our definition for probabilistic skipping-based data structures.

Definition 2 (Probabilistic Skipping-Based Data Structure)**.** *A probabilistic skipping-based data structure that represents a size n collection of elements from the domain* $\{0, 1\}^\lambda$ *is a data structure whose representation can be decomposed as a sequence of identically distributed random variables from some distributions* $\mathcal{X}$*. This sequence is either unordered (for data structures representing unordered data, like hash tables) or implicitly ordered by some well-defined ordering over the domain of the underlying collection (as is the case for ordered data structures, like skip lists and treaps).*

This definition offers a few key advantages. First, from an attack perspective, it helps us formally specify the necessary conditions for an adversary to succeed in our security game. For hash table attacks, this means forcing a large portion of the discrete uniform random variables $B_1, B_2, \ldots, B_n$ to be equal – a condition any successful attack strategy must achieve to degenerate the data structure. Additionally, it allows us to precisely differentiate between adaptive and non-adaptive adversarial capabilities. When decomposing a skip list into geometric random variables $H_1, H_2, \ldots, H_N$ (sorted according to key order), an adaptive adversary can observe previous outcomes and strategically insert a new H_i at any position in the sequence, thereby creating dependencies among the variables. In contrast, a non-adaptive adversary cannot observe previous geometric random variable outcomes, resulting in a final sequence $H_1, H_2, \ldots, H_N$ that maintains independence among the sequence of random variables.

Second, this stochastic formalization enables the application of well-established probabilistic techniques to derive tight bounds on adversarial success probabilities: balls-and-bins analysis for hash tables and martingale-based arguments for skip lists and treaps. Finally, for researchers looking at different PSDS from the ones we consider, it allows for generalization of our robust data structures: proving security for one structure characterized by a particular sequence of identically distributed random variables allows us to transfer robustness techniques to other structures of the same type. Though specific structural details may prevent exact technique transfer, this approach should inform effective general strategies.

5.2.1 Timing Side Channels

PSDS share a critical vulnerability: their runtime variation for distinct queries directly reveals information about their internal structure. This inherent timing side-channel has been successfully exploited in attacks against hash tables with (secret) salts [39] and skip lists [18]. For treaps, this vulnerability also manifests, as runtime correlates with node depth, potentially exposing the complete internal structure when combined with the ordering of the inserted elements. While remote attackers might face challenges like network latency in precisely measuring timing differences, recent research demonstrates that timing side-channels can be exploited with remarkable precision – as shown in [92], where researchers recovered an AES key from a Bluetooth chip's hardware accelerator.

Implementing enforced constant-time operations fails as a solution, as this would require the data structure to always operate at worst-case (linear) time, defeating the purpose of using these efficient structures. Similarly, making the data structure oblivious to prevent information leakage has significant limitations. Such approaches are inherently fragile – once an adversary learns anything about the internal structure, the security guarantees collapse entirely. As aforementioned, previous attempts to prevent information leakage in skip lists [18] by randomly swapping elements have proven unsuccessful.

Given these considerations, we adopt a more realistic approach by considering a very strong adversarial model. We grant the adversary full access to the internal structure of the PSDS, then prove that even with this knowledge, they cannot successfully degenerate the structure. This robust security model acknowledges that side channels inevitably exist in practical implementations and builds defenses that remain effective despite full information leakage. While this represents a strong adversarial capability, we argue it better reflects real-world threat scenarios than a model that assume perfect or partial information hiding.

5.2.2 Towards Robust PSDS

We observe that two abilities allow an adaptive adversary to shape the distribution of data in a PSDS such that subsequent operations on the structures are degraded with high probability. The first is the ability to *delete* elements. This allows an adversary to degenerate a structure after a series of insertions by deleting unfavorable (w.r.t. to the adversary's goal) elements. The second is the ability of the adversary to influence *where* a particular element gets placed in the structure upon insertion. This is akin to knowing in advance which bucket an element will be inserted into in a hash table, at what position and height an element will be inserted in a skip list, or the priority an element will receive upon insertion to a treap. Therefore, we propose two inexpensive and general modifications to the base PSDS to make them robust in an adversarial setting. We will later prove these modified structures secure.

Lazy Deletion. The first modification prevents the adversary from deleting (unfavorable) elements from the structure. This stultifies the ability of an adversary to perform a skip list degeneration-style attack, even with full access to the data structure's internal state.

Removing the deletion functionality entirely from our data structure would be undesirable. Instead, we use a simple scheme that allows for removing elements without modifying the underlying structure of a PSDS that previous insertions have imposed. We achieve this by simply labeling an element as "deleted". For the hash table, we replace the element's label (e.g., the key-value data) with a distinguished symbol $\diamond$ but do not modify the linked list in a hash table bucket by removing the node. For operational reasons, in the skip list and treap, we store a bit along with each node that indicates whether an element has been removed, but do not overwrite the originally inserted key with a distinguished symbol.

This change prevents the adversary from eliminating desired skip connections in a skip list, obtaining trivial wins in our security model against a hash table (when taking the represented set to be the collection of all empty and non-empty elements), or only allowing elements to persist solely on the longest path in a treap. However, this modified deletion functionality affects the space efficiency of the structures. In later sections, we discuss approaches to ameliorating such concerns and analyze the trade-offs of these approaches. Lastly, since "all bets are off" when deletions are allowed, we implicitly provide security bounds that would compare to insertion-only versions of these structures in the non-adaptive case. That is, since an adaptive adversary can pathologically degrade the base structures, we enforce that using our modified deletion mechanisms never helps the adversary achieve their goal (in fact, this point is the first step of all our security proofs).

Adversarial Robustness. The second modification eliminates (to the greatest extent possible) an adversary's ability to predict element placement within data structures. All analyzed data structures require distinct security approaches for adversarial robustness, which heavily depends on how randomness is used internally. Skip lists and treaps use per-insertion randomness while preserving key-based ordering. During queries, the element's key guides traversal, although specific paths vary based on insertion-time randomization. Hash tables function fundamentally differently – they determine bucket placement solely based on random experiment outcomes rather than element keys. This approach necessitates reproducing identical outcomes during search queries. Conventional implementations rely on public hash functions, creating a critical security vulnerability: adversaries can precalculate outcomes for elements and execute complexity attacks.

To provide adversarial robustness for hash tables, we replace public hash functions with secretly keyed primitives that effectively behave like truly random functions, preventing adversarial precalculation. Note that this approach necessitates secret key management, which presents potential implementation challenges. For skip lists, we develop an unkeyed, algorithmic approach to secure against adversarial manipulation. Despite an adversary's inability to have a priori knowledge of coin flip outcomes that determine the height of an element, skip lists remain vulnerable – an adversary can strategically shift unfavorable random outcomes to one side of the structure, effectively placing elements with specific heights at chosen positions. We counter this by enforcing a *local* balance in the internal representation through a constant overhead swap operation, making such attacks exponentially more difficult.

Note that simply pre-applying a (secretly-keyed) random function to the items a skip list stores, as in the hash table mechanism, would alter the structure's ordering, rendering these data structures incapable of performing range queries, join operations, and other order-dependent functions. We therefore developed security mechanisms that maintain fundamental ordering properties while enhancing attack resilience.

Treaps, by contrast,inherently rebalance their entire structure based on the priorities of all previously inserted elements. As we will demonstrate, this property already substantially reduces an adversary's ability to place elements at positions of their choosing.

5.3 A Security Model for Probabilistic Skipping-Based Structures

Informal Security for Probabilistic Skipping-based Structures.

Our goal is to capture the average-case run time of operations PSDS being *conserved* in the face of an adaptive adversary that can control the data represented by the structure. Loosely, the average-case run time of PSDS relates to how data is "distributed" in the representation. For instance, an ideal hash table would distribute the elements it represents equally among the buckets. Analogously, ideal ordered structures (e.g., a skip list or a treap) would be isomorphic to a balanced tree. If a data collection was fixed, and we ignored a desire for efficiency, one could always craft an ideal representation with respect to the runtime of queries. For a hash table, one could find a hash function that equally distributes the fixed collection to its buckets. For a fixed-ordered structure, one could simply assign the heights (depths) of elements such that the shortest possible search paths are guaranteed, as with a perfectly balanced tree structure.

However, PSDS are used in mutable settings. For this reason (and for efficiency), PSDS use some form of randomness to process updates dynamically and update their representation. Hash tables select a random hash function to map elements to buckets, and ordered PSDS employ per-operation randomization during insertion to determine an element's position in the structure

— typically through coin flips for skip lists or random priority assignments for treaps. These processes have been shown (with high probability) to yield representations of a dynamic data collection that are "close" to the ideal representations. Hash tables are analyzed using standard ball-and-bin arguments. Assuming a collision-resistant hash function and a load factor such that $n \approx b$ (i.e., the size n of the data collection stored is about equal to the number b of buckets), it is known [93] that with probability $p = 1 - \frac{1}{b}$ that at any point in time no bucket has more than $3\frac{\log b}{\log \log b}$ entries. This maximum bucket population bounds directly corresponds with a subsequent operation's maximum insertion, deletion, or query time. Likewise, the maximum search cost path of any element queried to a skip list or treap has been shown to not exceed $O(\log n)$ with high probability (where the exact constants are functions of the parameters of the structure).

The above analyses are done under a strictly *non-adaptive* adversarial assumption. That is, these probabilistic bounds on the "distribution" of elements are done under the assumption that the updates and queries made to the structure do not depend on the internal randomness of the structure, the results of past operations, or the state of the representation. In the adaptive adversarial setting, this cannot be assumed. This is seen in both the hash flooding attack and the skip list degeneration attack [13, 39, 14, 18]. Therefore, intuitively, a robust PSDS would conserve the desired element distribution property of the structure with high probability, even in the face of an adaptive adversary. This is what we aim to capture with our formal security model.

Formal Security Model.

Let $\Pi = (\textsc{Rep}, \textsc{Up}, \textsc{Qry})$ be a probabilistic skipping-based data structure. We define a notion of adversarial property conservation involving Π, a property function $\phi : \mathcal{D} \times \{0,1\}^* \longrightarrow \mathbb{R}$, a target bound $\beta : \mathcal{P} \times \mathbb{Z}^+ \longrightarrow \mathbb{R}$, and a threshold $\epsilon \in \mathbb{R}, \epsilon > 0$.

A property function ϕ takes as input the data object $D \subseteq \mathcal{D}$ represented by repr (the representation the adversary produces during its execution) and the representation repr itself and outputs a value that indicates the concrete property for the given adversarially chosen data collection and corresponding representation. This function represents the desired property one

$\mathbf{Exp}^{\mathrm{aapc}}_{\Pi,\phi,\beta,\epsilon}(\mathcal{A})$

$r \leftarrow 0; K \twoheadleftarrow \mathcal{K}$
$\text{done} \twoheadleftarrow \mathcal{A}^{\mathbf{Rep},\mathbf{Up},\mathbf{Qry}}$
$\textbf{return } \left[\dfrac{\phi(D, \mathsf{repr})}{\beta(\mathcal{P}, |D|)} \geq \epsilon\right]$

$\mathbf{Hash}(X)$

$\textbf{if } X \notin \mathcal{X} : \textbf{return } \perp$
$\textbf{if } H[X] = \perp$
$\quad X[X] \twoheadleftarrow \mathcal{Y}$
$\textbf{return } H[X]$

$\mathbf{Rep}(C)$

$\textbf{if } r = 1 : \textbf{return } \perp$
$r \leftarrow 1$
$\mathsf{repr} \twoheadleftarrow \textsc{Rep}_K(C)$
$D \leftarrow C$
$\textbf{return } \mathsf{repr}$

$\mathbf{Up}(\mathsf{up})$

$\mathsf{repr} \twoheadleftarrow \textsc{Up}_K(\mathsf{repr}, \mathsf{up})$
$D \leftarrow \mathsf{up}(D)$
$\textbf{return } \mathsf{repr}$

$\mathbf{Qry}(\mathsf{qry})$

$\textbf{return } \textsc{Qry}_K(\mathsf{repr}, \mathsf{qry})$

Figure 5-4. The Adaptive Adversary Property Conservation (AAPC) security game. The experiment enforces that the adversary is only able to call **Rep** once. The experiment returns the output of a predicate that returns 1 iff the property function $\phi(D, \mathsf{repr})$ computed over the representation the adversary interacts with is greater than ϵ-times (for some $\epsilon > 0$) larger than some target bound β (that only depends on the parameters of the structure $\mathcal{P}$ and the size of the represented data object $|D|$). The **Hash** oracle computes a random mapping $\mathcal{X} \rightarrow \mathcal{Y}$ (i.e., a random oracle), and is implicitly provided to Rep, Up and Qry as needed.

HT Maximum Search Path: $\phi(D, \mathsf{repr})$

$e \leftarrow 0$
$\textbf{for } i \leftarrow 1 \textbf{ to } m$
$\quad \ell \leftarrow \mathsf{length}(T[i])$
$\quad \textbf{if } \ell > e$
$\qquad e \leftarrow \ell$
$\textbf{return } e$

Figure 5-5. The HT Maximum Search Path function $\phi : \mathcal{D} \times \{0,1\}^* \rightarrow \mathbb{R}$. The function iterates through all m buckets, returning the bucket with the greatest population, which is equivalent to the longest search path in the table.

TR Maximum Search Path: $\phi(D, \mathsf{repr})$

```
return φ^rec(T.root, 0)
```

$\phi^{\text{rec}}(n, e)$

```
if n = null then
    return
e_1 ← φ^rec(n[2], e + 1)
e_2 ← φ^rec(n[3], e + 1)
return max(e_1, e_2)
```

Figure 5-6. The TR Maximum Search Path function $\phi : \mathcal{D} \times \{0,1\}^* \rightarrow \mathbb{R}$. The function performs an in-order traversal for all elements $d \in D$, returning the longest search path cost among them.

SL Maximum Search Path: $\phi(D, \mathsf{repr})$

```
m ← 0
for d ∈ D
    ℓ ← 0, c ← L.header
    for i ← L.level downto 1 do
        while c[i] ≠ null and c[i][0].key < d    do
            c ← c[i], ℓ ← ℓ + 1
    c ← c[1], ℓ ← ℓ + 1
    if c ≠ null and c[0].key = d then
        if ℓ > m then
            m ← ℓ
return m
```

Figure 5-7. The SL Maximum Search Path functions $\phi : \mathcal{D} \times \{0,1\}^* \rightarrow \mathbb{R}$. The function iterates through all elements $d \in D$, returning the longest search path cost among them. Our function only computes rightward pointer traversals, as downward movements equate to a simple array lookup.

would like to conserve. For all structures of interest, this is the maximum search path cost over all elements $d \in D$ [1]. The intuition is that a complexity attack is deemed successful precisely when it significantly increases the maximum search path cost; therefore, a robust data structure must maintain nearly equivalent worst-case performance (with high probability) regardless of adversarial manipulation. We give the exact property function for a hash table in Figure 5-5, for a skip list in Figure 5-7 , and for a treap in Figure 5-6.

A target bound β takes as input the structure parameters $\mathcal{P}$ (e.g., the number of buckets for a given hash table) and a size of the represented data object $|\mathcal{D}|$ (denoted n below), and outputs the resulting bound value. We choose a target bound such that it corresponds to the known non-adaptive bound for the property we want to conserve. For the hash table maximum search path cost, this is $\beta(\langle b \rangle, n) = 3\frac{\log b}{\log \log b}$. For the skip list and treap maximum search path, we chose $\beta(\langle p, m \rangle, n) = c \log_{1/p}(n)$ (for a small constant c), and $\beta(\langle\rangle, n) = 2\lg(n) + 1$, respectively, as these are the (blunt) non-adversarial expected search path lengths [33, 28].

We give this notion of adversarial property conservation in Figure 5-4. The *Adaptive Adversary Property Conservation* (AAPC) experiment aims to capture an adversary's ability to adaptively craft a representation repr of some dynamic and adversarially decided data object D, such that when the property function ϕ is computed, the ratio of its output to the target bound's output is large (to win the experiment this ratio needs to exceed ϵ). As the properties (and their accompanying target bounds) measure how data elements are distributed in a particular representation (and bound how they are distributed in the non-adaptive setting), this notion directly translates to an adversary's ability to disrupt the expected runtime of a data structure's operations.

The AAPC experiment begins by setting a parameter $r = 0$ and selecting a key K from the key space $\mathcal{K}$. For unkeyed hash tables (insecure) and non-deterministic versions of the ordered PSDS, the key space is the empty set. The adversary is then allowed to instantiate the data structure with any initial data object C (including the empty data object) via the **Rep** oracle and

[1]This property sufficiently captures the search path cost of any d in the universe of all possible elements, as a search for an element not in the representation terminates with at most one more pointer traversal compared to any element in the representation.

receives back the resulting representation. We enforce that the adversary is only allowed to call **Rep** once via the parameter r. This is to disallow the adversary from leveraging past information from a data structure that is keyed with the same key K to trivially win the game. That is, keyed hash tables and deterministic PSDS must sample a fresh random key to guarantee security.

The adversary is then allowed to make any sequence of **Up** and **Qry** calls. Upon each update, we also update the internal data object D kept by the experiment, as this is used for computing ϕ and β. After each update, the updated representation repr is returned to the adversary. Thus, the notion of security we propose is quite strong in that it allows an adversary to have complete access to the structure's internals during its execution (as discussed in Section 5.2.1). The only information kept from the adversary is the secret key (in the case the structure relies on one). This further makes calls to **Qry** unnecessary, as the adversary entirely determines the underlying collection represented by the structure and has access to the internal representation at all times.

The adversary ends its execution by announcing **done** or is implicitly done when it exhausts its **Up** budget (the number of updates they are allowed to make). The experiment concludes by outputting a bit that determines whether the adversary has successfully met the winning condition.

With this intuition built, we give our succinct formal definition of security.

Definition 3 (($\phi, \beta, \epsilon, \delta, t$)-Conserved)**.** *We say a skipping-based probabilistic data structure* Π *is* $(\phi, \beta, \epsilon, \delta, t)$*-conserved if the advantage of an AAPC-adversary* $\mathcal{A}$ *running in time* t *is less-than-or-equal to* δ *for some property function* ϕ*, some target bound* β*, some* $\epsilon \in \mathbb{R}, \epsilon > 0$*, and some* $\delta \in [0, 1)$*. More precisely, we say the structure is* $(\phi, \epsilon, \beta, \delta, t)$*-conserved iff,*

$$\mathbf{Adv}^{\mathrm{aapc}}_{\Pi,\phi,\beta,\epsilon}(\mathcal{A}) = \Pr[\mathbf{Exp}^{\mathrm{aapc}}_{\Pi,\phi,\beta,\epsilon}(\mathcal{A}) = 1] \leq \delta$$

and write $\mathbf{Adv}^{\mathrm{aapc[u,v]}}_{\Pi,\phi,\beta,\epsilon}(t, q_Q, q_U, q_H)$ *as the maximum advantage of any AAPC-adversary running in* t *time steps and making* q_Q *calls to* **Qry***,* q_U *calls to* **Up***, and* q_H *calls to* **Hash** *in the ROM. We are interested in ensuring* $\mathbf{Adv}^{\mathrm{aapc}}_{\Pi,\phi,\beta,\epsilon}(t, q_Q, q_U, q_H) \leq \delta$*.*

5.4 Robust Hash Tables

5.4.1 Insecurity Of Standard Hash Tables

Unkeyed Hash Tables.

Consider a standard hash table instantiated with a fixed and publicly known hash function. A simple pre-computation attack will trivially win our security experiment (with the experiment parameters the same as in Theorem 5-1) with probability one (assuming the ability to make sufficiently many local hash computations). An adversary can sample index keys from the universe and compute the bucket they will map to by using the public hash function (assuming the parameters of the structure are known). The adversary can select a target bucket and insert index keys (with some arbitrary value) iff they map to this target bucket. In this way, an adversary can ensure that all elements go to a single bucket, causing a linear overhead when searching for an element in this bucket.

Keyed Hash Tables with Deletions.

Consider a hash table where we replace a public hash function with a secretly keyed primitive, like a PRF. Our security game also yields a simple strategy for an adversary to win our game with a high probability if the hash tables support deletions in the usual way. The adversary selects a target bucket. Then it samples keys from the universe (along with arbitrary values for these keys) and inserts them into the table. Observing the state of the table after each insertion, the adversary deletes the element unless it has been inserted in the target bucket. At the end of the adversary's execution, the hash table will only have elements that reside in a single bucket. For this reason, we do not allow adversaries to make deletions that actually remove elements from the hash table and compare our adversarial results to a standard ball-in-bins result that assumes no deletions.

While an attack of this nature may seem vacuous and an artifact of our security experiment, it is designed to capture something more complex. Consider if you could guarantee that the state of the hash table could remain hidden during the adversary's execution. Then, it seems intuitive that just keying the structure would result in robust construction per our security definition. However, as evidenced by side-channel attacks against hash tables [39], it is nearly impossible to guarantee that the internal structure of the hash table remains entirely hidden. Therefore, we continually leak the entire state of the structure to the adversary during its execution to emulate the best possible side channel (as detailed in Section 5.2.1).

5.4.2 A Robust Construction

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

$\textbf{for } i \leftarrow 1 \textbf{ to } m \textbf{ do}$
$\quad T[i] \leftarrow \mathsf{new\ L}$
$\textbf{for } (x, v) \in \mathcal{S}$
$\quad T \leftarrow \textsc{Up}_K(T, \mathsf{ins}_{(x,v)})$
$\textbf{return } T$

$\textsc{Up}_K(T, \mathsf{del}_x)$

$v \leftarrow \textsc{Qry}_K(T, \mathsf{qry}_x)$
$\textbf{if } v \neq \star$
$\quad i \leftarrow \textsc{Hash}(K, x)$
$\quad T[i].\mathsf{replace}((x, v), (\diamond, \diamond))$
$\textbf{return } T$

$\textsc{Up}_K(T, \mathsf{up}_{(x,v)})$

$v \leftarrow \textsc{Qry}_K(T, \mathsf{qry}_x)$
$\textbf{if } v \neq \star$
$\quad \textsc{Up}_K(T, \mathsf{del}_x)$
$i \leftarrow \textsc{Hash}(K, x)$
$T[i].\mathsf{ireplace}((x, v), (\diamond, \diamond))$
$\textbf{return } T$

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

$v \leftarrow \star$
$i \leftarrow \textsc{Hash}(K, x)$
$v' \leftarrow T[i].\mathsf{find}(x)$
$\textbf{if } v' \neq \mathsf{null}$
$\quad v \leftarrow v'$
$\textbf{return } v$

Figure 5-8. A robust hash table in the AAPC security model. It is an explicitly keyed hash-table structure RHT[$\textsc{Hash}, b$] admitting insertions, modified deletions, and queries for any $k \in \mathcal{U}_\kappa$ and its associated value v. The parameters are an integer $b \geq 1$, and a keyed function $\textsc{Hash} : \mathcal{K} \times \mathcal{U}_\kappa \rightarrow [b]$ that maps the key part of key-value pair data-object elements (encoded as strings) to a position in the one of the table buckets $v.T$. A particular choice of parameters gives a concrete scheme. Each bucket contains a simple linked list L equipped with its usual operations. We define the $\mathsf{replace}$ operation of L, such that if it finds an item with $(x, v) = (\diamond, \diamond)$ during its internal search, the item to be inserted is written in this location; otherwise a regular insertion occurs. If an item is not contained in the map, the distinguished symbol $\star$ is returned.

We give a robust hash table construction in Figure 5-8. The robust hash table requires that a keyed mapping function R is used. Concretely, this can be instantiated as PRF that is then mapped to b (by, say, taking the output of the PRF modulo b). In particular, SipHash [15] provides performance that is comparable to traditionally used non-cryptographic hash functions [11]. We also use our modified deletion scheme. The deletion functionality simply relabels the key-value pair to be deleted as $(\diamond, \diamond)$, where $\diamond$ is a distinguished symbol. The insertion functionality changes such that if an element to be inserted can overwrite a linked list node containing $(\diamond, \diamond)$, it does; otherwise, a normal insertion occurs. The query functionality remains unchanged.

We will now state and prove a formal security theorem and prove the robust hash table construction secure in the AAPC model.

Theorem 5-1 (Robust Hash Table AAPC Security Result)**.** *Let* Π *be our robust hash table from Figure 5-8, using PRF F to map elements to buckets. For integers* $q_U, q_Q, q_H, t \geq 0$ *such that* $q_U = b$ *(i.e., b is the number of buckets in the hash table Π), it holds that* Π *is* $(\phi, \beta, \epsilon, \delta, t)$*-conserved with ϕ being the HT Maximum Bucket Population function (Figure 5-5),* $\beta = 3\frac{\log b}{\log \log b}$, $\epsilon = 1$, *and* $\delta = (\frac{1}{n} + \mathbf{Adv}_F^{\mathrm{prf}}(O(t), b + q_Q))$.

Proof. Observe that the modified insertion and deletion procedures ensure that once an element is inserted into a bucket, it cannot actually be removed but rather only relabeled (either to $(\diamond, \diamond)$ or a newly inserted key-value pair). Observe that an optimal adversary never makes deletions for our construction, as our modified deletion procedure ensures this cannot possibly add to the maximum search path cost. Thus, we start with a game that assumes the adversary never makes deletions, and the proof follows from a simple hybrid argument.

We start with a game $\mathbf{G}_0$ that is that the AAPC security game instantiated with our robust hash table Π using PRF F, property function ϕ as the HT Maximum Bucket Population function (Figure 5-5), and target bound $\beta = 3\frac{\log b}{\log \log b}$. As indicated by the theorem statement, we assume that the adversary cannot insert more than $q_U = b$ distinct elements into the table and, from above, never makes a deletion. In this game, the number of times F is evaluated on distinct inputs bounded by the adversary's resource budget. Calls to **Up** (also implicitly used by **Rep**) call F

once. Calls to **Qry** also call F once. Thus, when executed with $\mathcal{A}$, game $\mathbf{G}_0$ makes at most $Q = b + q_Q$ queries to F.

Let $\mathbf{G}_1$ be identical to $\mathbf{G}_0$ except we use truly random sampling (modeled in the ROM) in place of the PRF. If $\mathcal{A}$ cannot distinguish F from a random function. Then, these games are indistinguishable from the adversary's perspective. We build a $O(t)$-time PRF distinguishing adversary $\mathcal{B}$ making at most Q queries to its oracle such that

$$\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{B}) = \Pr[\mathbf{G}_0(\mathcal{A}) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}) = 1]. \tag{5-1}$$

Adversary $\mathcal{B}^F$ works by executing $\mathcal{A}$ in $\mathbf{G}_1$. Whenever $\mathbf{G}_1$ calls F, adversary $\mathcal{B}$ computes the response using its own oracle. When $\mathcal{A}$ halts, if the winning condition of $\mathbf{G}_1$ is satisfied, then $\mathcal{B}$ outputs 1; otherwise it outputs 0. Conditioning on the outcome of the coin flip z in $\mathcal{B}$'s game, we have the following:

$$\begin{aligned}
\mathbf{Adv}_F^{\mathrm{prf}}(\mathcal{B}) &= 2\Pr[\mathbf{Exp}_F^{\mathrm{prf}}(\mathcal{B} = 1)] - 1 \\
&= 2(\frac{1}{2}\Pr[\mathbf{Exp}_F^{\mathrm{prf}}(\mathcal{B} = 1)|z = 1] \\
&+ \frac{1}{2}\Pr[\mathbf{Exp}_F^{\mathrm{prf}}(\mathcal{B} = 1)|z = 0]) - 1 \\
&= \Pr[\mathbf{Exp}_F^{\mathrm{prf}}(\mathcal{B} = 1)|z = 1] + \Pr[\mathbf{Exp}_F^{\mathrm{prf}}(\mathcal{B} = 1)|z = 0] - 1 \\
&= \Pr[\mathbf{G}_0(\mathcal{A}) = 1] - \Pr[\mathbf{G}_1(\mathcal{A}) = 1].
\end{aligned}$$

Now, with $\mathbf{G}_1$, we immediately have a standard insertion-only truly random balls-and-bins problem with $\le q_U = b$ balls being randomly thrown into $q_U = b$ bins. We can apply the standard bound and conclude $\phi(\cdot) \le \beta(\cdot)$ (that is $\frac{\phi(\cdot)}{\beta(\cdot)} \le \epsilon = 1$) with probability $1 - \delta$ where $\delta = (\frac{1}{b} + \mathbf{Adv}_F^{\mathrm{prf}}(O(t), Q))$. The first term comes from the standard bound and the second results from the hybrid we showed above. $\square$

To give a concrete illustration of this bound, suppose we had $n = b = 2^{32}$ and $\epsilon = 1$. Leveraging our results from Theorem 5-1, the probability our maximum search cost path is greater than $M = 3\frac{\log 2^{32}}{\log\log 2^{32}} \approx 21.47$ is less than or equal to $\delta = \frac{1}{2^{32}} + \mathbf{Adv}_F^{\mathrm{prf}}(O(t), b + q_Q) \approx 2.33 \cdot 10^{-10} + \mathbf{Adv}_F^{\mathrm{prf}}(O(t), b + q_Q)$.

5.4.3 Robust Hash Tables in Real World Deployments

When initializing a hash table, there's an implicit promise to allocate enough memory for a pre-defined number of elements. If the collection grows too large and exceeds this capacity, the structure must be resized, typically by doubling the number of buckets. For a key-value pair where keys are x bits and values are v bits, we expect to allocate up to $\alpha \cdot b \cdot x \cdot v$ bits of memory, where α is the load factor defined as $\alpha = \frac{n}{b}$, with n being the number of elements and b the number of buckets [1]. If the load factor exceeds a set limit, resizing is required. In our security experiment, we implicitly specify a load factor of $\alpha \leq 1$ by setting $q_U = b$, and in turn, never allow this load factor to be exceeded. That is, we do not consider attacks that would trigger resizing. Hence, we discuss the consequences of our robust construction in real-world deployments below by analyzing how our modifications change the frequency of required resizing.

Consider a standard hash table with I successful insertions and D successful deletions. For resizing to be necessary, it must be that $\frac{I-D}{b}$ has exceeded α. At some point, before resizing is triggered, if the rate of insertions and deletions are roughly equal, a structure could persist indefinitely without resizing.

Now consider a modification where deletions merely mark elements as deleted without allowing for the possibility of being replaced by fresh insertions. That is, we do not modify the insertion procedure to replace previously deleted elements. In this scenario, resizing occurs when $\frac{I}{b}$ has surpassed α, even if only a few elements are actually represented in the structure. This could occur when an adversary inserts $\approx \alpha \cdot b$ elements, then deletes nearly all of them[2], and finally triggers a resizing with a few subsequent fresh insertions. Although this seems wasteful, it aligns with the resizing logic since the total insertions exceed the threshold. That is, a resizing is

[2]Of course, if an adversary deleted all elements, it would be trivial to flush the table and reinitialize the structure.

triggered only after the total number of insertions exceeds the threshold set by α (regardless if a deletion has subsequently nullified them).

We would like our robust hash table to conserve the property where deletions free space, such that $\frac{I}{b} > \alpha$ does not necessarily trigger a resizing. Thus, in addition to marking deleted elements, we also prefer replacing said deleted elements with new insertions. This is desirable in the non-adversarial case (where insertions, deletions, and queries do not depend on the internal randomness of the structure, the internal state of the structure, or past operations), as one expects freshly inserted elements will eventually replace deleted elements.

Adversarial strategies can still trigger resizing with few non-deleted elements. For example, an adversary could insert $I = \alpha \cdot b - 1$ elements, delete all but those in the least populated bucket, and with a $1/b$ probability, trigger resizing with only those elements in that bucket remaining. While this requires the adversary to exceed the threshold number of insertions, making it marginally problematic in practice, the collection size at the time of resizing may be smaller than the non-adversarial threshold. In sum, while adversaries can still trigger small collection resizing under certain conditions, our approach ensures the hash table is provably robust and allows it to persist for extended periods without resizing if insertions and deletions are balanced in the non-adversarial setting.

5.5 Robust Skip Lists

5.5.1 Insecurity of Standard Skip Lists

As noted in [33], the heights of the elements in the skip list must be kept secret, or otherwise, the skip list can be degenerated by simply deleting all elements in the list that are not at height zero. In our security model, this attack is trivial. However, even when disallowing deletions (or using our modified deletion) functionality, an adaptive adversary can still degenerate the skip list using a powerful but subtle strategy. We call this the *gap attack* and detail it next, but use an intuitive rather than a formal description.

We assume the skip list takes values from $\{0, 1\}^n$, which we interpret as integers between 0 and $2^n - 1$. The gap attacker proceeds as in Figure 5-9. It starts by inserting an element in the middle $M = 2^{n-1}$ of the interval $[L, R] = [0, 2^n]$. If this element gets assigned a height of 0 in the skip list, i.e., is only inserted in the bottom list, then the attacker secures it by shifting the left bound L of the interval to M, moving to that gap. In the other case, if the height is large, it "gives up" this part of the skip list and moves the right bound R of the interval to M. Continue with the new interval $[L', R']$ of half the size until n elements have been inserted.

Gap attack on skip list

$L \leftarrow 0, R \leftarrow 2^n$
for $i \leftarrow 1$ **to** n
  insert element $M \leftarrow (R + L)/2$
  if height$(M) = 0$ **then** $L \leftarrow M$ **else** $R \leftarrow M$

Figure 5-9. The gap attack on skip lists, inserting n elements from $\{0, 1\}^n$.

By construction, the value M in each iteration is always an integer between L and R. Moreover, at the end of each iteration, there are only elements of height 0 in the interval $[0, L]$ (if any), and all elements of larger height in $[R, 2^n]$ (if any), since we set the left resp. right bound accordingly in each iteration. Hence, after n iterations and $R - L = 1$ we have all elements of height 0 in $[0, L]$, and elements of larger height in $[L + 1, 2^n]$. In each iteration, we insert an element of height 0 with constant probability $1 - p$, which will eventually lie in the interval $[0, L]$. Therefore, the expected number of elements in $[0, L]$ is $(1 - p)n$. The resulting skip list is now highly degenerated in the interval $[0, L]$. Specifically, it corresponds to a simple linked list of average length $(1 - p)n$ in this part. Hence, the search for the element L takes linear time on average, whereas a regular skip list would yield a logarithmic average search time. This is an exponential blow-up in running time, which the gap attacker enforces.

5.5.2 A Robust Construction

In Figure 5-10, we give a pseudocode description of the robust skip list structure. Notably, elements can be marked are deleted by marking a "deleted" bit d, that is stored in the node as $\top$. It is important to point out that deleting and reinserting an element does not change the associated height, as only the d bit is flipped to $\bot$. Importantly, we do not allow for deleted elements to be

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

```
h ← NewNode(m, ⋆)
L.header ← h, L.level ← 1
for (x, v) ∈ S
    L ← Up_K(L, ins_(x,v))
return L
```

$\textsc{NewNode}(\ell, (x, v))$

```
    / array position 0 is reserved for
    / a deleted bit, key, value triple (d, x, v)
    / accessible via n.del, n.key and n.value
    / array positions 1 . . . ℓ are forward pointers
    / level is accessible via n.level
node ← new [0, .., ℓ]
node[0] ← (⊥, x, v)
for i ← ℓ downto 1 do
    node[i] ← null
return node
```

$\textsc{RandomLevel}_K(\boxed{x})$

```
[ ℓ ← R(K, x, m, p) ]
[ return ℓ ]
ℓ ← 1, r ↞ [0, 1)
while r < p and ℓ < m do
    ℓ ← ℓ + 1, r ↞ [0, 1)
return ℓ
```

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

```
c ← L.header
for i ← L.level downto 1 do
    while c[i] ≠ null and c[i][0].key < x do
        c ← c[i]
c ← c[1]
if c ≠ null and c[0].key = x and c[0].del ≠ ⊥ then
    return c[0].value
else
    return ⋆
```

$\textsc{Up}_K(\mathsf{L}, \mathsf{ins}_{(x,v)})$

```
u ← new[1, .., m]   / local array of pointers
c ← L.header
for i ← L.level downto 1 do
    while c[i] ≠ null and c[i][0].key < x do
        c ← c[i]
    u[i] ← c
c ← c[1]
if c ≠ null and c[0].key = x then
    c[0].value ← v, c[0].del = ⊥
else
    ℓ ← RandomLevel_K([x])
    if ℓ > L.level then
        for i ← L.level + 1 upto ℓ do
            u[i] ← L.header
        L.level ← ℓ
    n ← NewNode(ℓ, (x, v))
    for i ← 1 upto ℓ do
        n[i] ← u[i][i], u[i][i] ← n
    / find layer ℓ − 1 middle element using tortoise and hare
    middle ← u[ℓ], fast ← u[ℓ]
    while fast ≠ n[ℓ] and fast[ℓ − 1] ≠ n[ℓ] do
        middle ← middle[ℓ − 1], fast ← fast[ℓ − 1][ℓ − 1]
    / swapping logic
    if ℓ > middle.level then
        middle.append(n[ℓ]), n ← n[0 : ℓ − 1]
        u[ℓ][ℓ] ← middle
return L
```

$\textsc{Up}(\mathsf{L}, \mathsf{del}_x)$

```
c ← L.header
for i ← L.level downto 1 do
    while c[i] ≠ null and c[i][0].key < x do
        c ← c[i]
c ← c[1]
if c ≠ null and c[0].key = x then
    c[0].del = ⊤
return L
```

Figure 5-10. A robust, possibly "deterministic" (and keyed) skip list structure SL[$\boxed{R}$, m, p] admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameters are an integer $m \geq 0$ representing the maximum level of the structure, a fraction $p \in (0, 1)$ used for determining an element's random level, and, if using the deterministic version of the structure, a keyed function $R : \mathcal{K} \times \mathcal{U} \times \mathbb{Z}^{+} \times (0, 1) \to [m]$ that maps an element to a level in accordance with the distribution imposed by m and p. A concrete scheme is given by a particular choice of parameters. Subroutines used by the deterministic version of the structure appear in the boxed environment. We define the append operation, such that it appends an element to the end of the node array.

replaced by subsequent insertions due to the need to preserve order-sensitive operations, and the fact that such a mechanism could be used to accelerate the gap attack we present above.

Swapping mechanism for robust skip lists

```
  / find layer ℓ − 1 middle element using tortoise and hare
middle ← u[ℓ], fast ← u[ℓ]
while fast ≠ x[ℓ] and fast[ℓ − 1] ≠ x[ℓ] do
  middle ← middle[ℓ − 1], fast ← fast[ℓ − 1][ℓ − 1]
  / swapping logic
if ℓ > middle.level then
  middle.append(x[ℓ]), x ← x[0 : ℓ − 1]
  u[ℓ][ℓ] ← middle
```

Figure 5-11. The swapping mechanism for robust skip lists, which is invoked after a node x has been inserted on layer ℓ and update vector u has been constructed during this process.

We use a simple swapping mechanism to make the skip list robust (depicted in Figure 5-11). When the skip list inserts an element (denoted as x), it first performs a standard insertion and then invokes the swapping procedure using the update vector u constructed during insertion. After node x is inserted on layer ℓ, the mechanism counts nodes on layer $\ell - 1$ between $u[\ell]$ and $\mathsf{x}[\ell]$ (x's successor on level ℓ).

The middle element is then identified in a single pass using the tortoise and hare algorithm [94]. If the middle element is not x itself (verified by checking $\ell <$ middle.lvl), a height swap occurs: the middle element's height increases to level ℓ while node x's height decreases to level $\ell - 1$, effectively exchanging their heights.

This mechanism locally balances the skip list, preventing adversaries from creating large sequences of same-height elements that would result in search path blowup. The gap attack specifically becomes highly infeasible, as elements of a fixed height can no longer be shifted toward one side of the data structure. Instead, heights are immediately swapped at the interval's midpoint, halving long sequences of elements on level $\ell - 1$. Note, this approach effectively handles corner cases where $\ell =$ list.header or $n[\ell] =$ null. Moreover, the interval typically contains a constant, denoted a, number of nodes with overwhelming probability, ensuring the mechanism operates in constant time with high probability.

We will formally show that our robust skip list is secure via a number of intermediary lemmas. The first of which proves a necessary condition for degenerating a skip list (including our robust version). We specifically analyze the skip list from the point of view of being able to have infinite height (we rectify this with reality before delivering our final result). One examines the set of elements that appear in the skip list strictly below level $L(n) = \log_{1/p}(n)$, where n is the number of ("deleted" or actual) elements in the skip list (i.e., its implied capacity), and the set of elements that appear at or above level $L(n)$.

We first relate the length of a search path (i.e., the number of nodes to be visited) to the maximal width w on each level below $L(n)$, where the maximal width describes the maximal number of level i elements between level $i+1$ elements in the skip list over all levels $i = 0, 1, \ldots, L(n) - 1$. Here, we call an element a *level i element* if the node's height is at least i. In particular, any level j element is also a level i element for $i \leq j$. We say that the element is a *max-level i element* if it is a level i element but not a level $i+1$ element.

Lemma 5-1 (Necessary Condition for Degenerating a Skip List)**.** *Consider a skip list for parameter $p \in (0, 1)$ holding n elements (possibly inserted by an adaptive adversary). If on all levels $i \in \{0, 1, \ldots, L(n) - 1\}$ the number of level i elements between any pair of level $i+1$ elements is at most w, then any search path is of length at most $2w \log_{1/p} n$, or the total number of elements on or above level $L(n)$ exceeds $w \log_{1/p} n$.*

The lemma states that, for the adversary to create a bad skip-list representation, it may either hope that many elements are assigned a height beyond $L(n)$—which is very unlikely since the heights are determined faithfully by the data structure—or it must ensure that there is a "degenerated" sub-lists exceeding the width w on some level. The latter matches our gap attack in Section 5.5.1, where we followed this strategy, and the lemma states that this is indeed the only valid attack strategy.

Proof. Assume that on all levels i, there exists at most w elements between any two elements on level $i+1$, and that the total size of the skip list on or above level $L(n)$ is at most $w \log_{1/p} n$. Then the search path below level $L(n)$ is at most $w \log_{1/p} n$ because whenever we descend to a level i (and the index to be searched is thus between the indexes of both level $i+1$ elements), we make at most w steps on the level i. This bounds the total number of steps on all levels i below $L(n)$ by $w \cdot L(n) = w \cdot \log_{1/p} n$. In addition, on level $L(n)$ or above, the total number of elements is bounded by $w \log_{1/p} n$, such that even searching all these elements cannot increase the overall number of inspected elements by more than $w \log_{1/p} n$. This yields an overall length of the search part of $2w \log_{1/p} n$. □

We formulate the following game to bound the number of elements on max-level i between two level $i+1$ elements for the robust skip list. We assume that the adversary can insert as many elements as they like (up to its insertion limit q_U) and that the adversary can insert into any gap arbitrarily many times. Further, observe that the adversary cannot influence the height of any particular element or alter the heights that were chosen by deletion due to our special deletion method. Then, the ability of the adversary to accrue elements that exist on level i between two level $i+1$ elements distills down to a coin-flipping game.

Given the number of individual trials n and probability p, the game is as follows. For each individual trial, a coin (that is *heads* with probability p and is *tails* with probability $1-p$) is flipped until a tail appears, at which point the particular trial is concluded. The outcome of a trial is the total number of *heads* that occurred during a particular trial. For instance, the outcome *tails* maps to 0, while the outcome *heads*, *heads*, *tails* maps to 2.

The game keeps a sequence of all the outcomes. Say the sequence at a point in time $t-1$ is $o_1, o_2, \ldots, o_{t-1}$. The adversary is allowed to run the next trial t anywhere within the sequence. That is, they could dictate the outcome of trial t (the result of which they do not control, as coin flips determine this) at the beginning of the sequence (before o_1), at the end of the sequence (after o_{t-1}), or anywhere in between two adjacent $o_{i-1}, o_i, i \leq t-1$. The trial is then run, the outcome recorded in the sequence, and the sequence relabeled (depending on where the adversary decided to place the outcome of the most recently run trial).

For each possible trial outcome, we have the following "halving" behavior concerning runs (consecutive subsequences) of outcome i for each $i \in \{0, 1, \ldots \log_{1/p}(n)\}$. If the adversary is trying to extend a particular run, they always insert it at the beginning or end of the run. By inspection of our robust skip list structure, this strategy is optimal, as it maximizes the probability of extending a particular run (by minimizing the probability of halving). Given this, when an adversary tries to extend a run of outcome i, three possible outcomes can occur:

1. if the outcome of this fresh trial is i, then the run extends by length 1;

2. if the outcome of this trial is $i + 1$; the length of the run is halved (or more precisely, the updated run length is the ceiling of dividing the previous run length by 2);

3. if the outcome of the fresh trial is any other outcome; the run length remains the same.

Observe that this is precisely equivalent to the procedure for an adaptive adversary inserting elements into our robust skip list with probability parameter p and insertion budget (skip list capacity) $n = q_U$. We specifically consider the scenario where the adversary tries to accrue elements that exist on level 0 between two level 1 elements. This is because the probability of the accruing elements on this level is maximized. Looking ahead, we will cast this run width accruing game as a stochasic process that is supermartingale, generalize our result for level 0 to all levels $i \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$, and combine them to get a bound on the maximum search path cost over the entire robust skip list.

Lemma 5-2 (Bounding Layer Sequential Elements for the Robust Skip List)**.** *Denote W_i the random variable describing the maximum sequence of elements that exist on max-layer i between two level $i + 1$ elements for $i \in \{0, 1, \ldots, \log 1/p(n) - 1\}$ for a robust skip list. For $\epsilon > 0$ and $a = \frac{2(1+p)}{p}$ let W be the event that there exists $W_i > a(1 + \epsilon)$ for some $i \in \{0, 1, \ldots, L(n) - 1\}$, then*

$$\Pr[W] \leq e^{(\lambda^* a) - (\epsilon \lambda^* a)},$$

where λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

Proof. Defining the Probabilistic Process. We begin by considering the adversary trying to accrue a run of outcome 0. Given the adversary can play many independent trials, indexed by integer t (and in reality bounded by $n = q_U$), we define a counter tracking the run length X_t, initialized to 0, that is updated as follows:

$$X_{t+1} = \begin{cases} X_t + 1 & \text{with probability } 1-p \\ \left\lceil \frac{X_t}{2} \right\rceil & \text{with probability } p(1-p) \\ X_t & \text{with probability } 1 - ((1-p) + p(1-p) = p^2 \end{cases}$$

For $X_t = x$,

$$\begin{aligned} \mathbb{E}[X_{t+1}|X_t = x] &= (1-p)(x+1) + p(1-p)(\frac{x}{2}+1) + p^2 x \\ &= \left((1-p) + \frac{p(1-p)}{2} + p^2\right)x + (1-p) + p(1-p) \\ &= \left(1 - \frac{p}{2} + \frac{p^2}{2}\right)x + 1 - p^2, \end{aligned}$$

where we approximate $\left\lceil \frac{x}{2} \right\rceil$ as $\frac{x}{2} + 1$.

Setting $A = \left((1-p) + \frac{p(1-p)}{2} + p^2\right)$ and $D = 1 - p^2$, we can solve for a fix point (i.e., the steady-state solution where the drift is zero) by solving $a = Aa + D = a(1-A) = D$:

$$a = \frac{D}{1-A} = \frac{1-p^2}{\frac{p(1-p)}{2}} = \frac{2(1+p)}{p}.$$

Therefore, for any, p the fixed point is $a = \frac{2(1+p)}{p}$.

Bounding the Process for Outcome 0. We next cast this process as martingale to be able to apply a concentration bound. Specifically, for trying to accrue a run of outcome 0, we define the process

$$M_t = e^{(\lambda(X_t - a))},$$

where $\lambda > 0$ is a parameter to be selected. Our goal is to show that when X_t exceeds a certain threshold (say $x \geq a + B$ for some constant $B > 0$), the process M_t is supermartingale. That is for all $x \geq a + B, \mathbb{E}[M_{t+1}|X_t = x] \leq M_t$.

Using the update rule for our process, we have for $X_t = x$:

$$\begin{aligned}\mathbb{E}[M_{t+1}|X_t = x] &= (1-p)e^{\lambda((x+1)-a) + p(1-p)e^{\lambda}((x/2+1)-a) + p^2 e^{\lambda(x-a)}} \\ &= e^{\lambda(x-a)}\left((1-p)e^{\lambda} + p(1-p)e^{\lambda(1-x/2)} + p^2\right).\end{aligned}$$

Observe that since the term $e^{\lambda 1 - x/2}$ decreases in x, the worst case for $x \geq a + B$ is exactly at $x = a + B$. Hence, it suffices to have

$$(1-p)e^{\lambda} + p(1-p)e^{\lambda(1-a+B/2)} + p^2 \leq 1$$

for our stochastic process to satisfy the supermartingale condition.

Further, we have $\frac{a+b}{2} = \frac{1+p}{p} + \frac{B}{2}$ and $1 - \frac{a+B}{2} = -\frac{1}{p} - \frac{B}{2}$, thus, our condition simplifies to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p} - \frac{B}{2}\right)} + p^2 \leq 1.$$

For any fixed $p \in (0, 1)$ and chosen $B > 0$ (in practice we chose B to be a small constant that is $\approx a$), one can solve for that largest λ that satisfies this inequality; denote this $\lambda^* = \lambda(p, B)$.

Now, define a stopping time

$$t_0 = \min\{t \geq 0 : X_t \geq a + B\}.$$

At this stopping time, we have

$$M_{t_0} = e^{\lambda^*(X_{t_0}-a)} \leq e^{\lambda^* B},$$

as $X_{t_0} \geq a + B$. We then work with the stopped process $M_{t \wedge t_0}$ (or more precisely with the process from time t_0 onward) and apply Ville's inequality [95]. This yields for all $k \geq 0$

$$\Pr\left[\max_{0 \leq t_0 \leq n} X_t \geq a + k\right] \leq \frac{\mathbb{E}[M_{t_0}]}{e^{\lambda^* k}} \leq e^{\lambda^* B} e^{-\lambda^* k}.$$

Define $C = e^{\lambda^* B}$, we then obtain the concentration bound for outcome 0 for all $k \geq 0$:

$$\Pr[X_n \geq a + k] \leq C e^{-\lambda^* k}.$$

Recasting in the multiplicative form (as $(1 + \epsilon)a = a + \epsilon a$), for any $\epsilon > 0$ we have

$$\Pr[X_t \geq (1 + \epsilon)a] \leq C e^{-\lambda^* \epsilon a}.$$

Lifting Result to All Outcomes. Next, we use a chaining argument to lift our result to the maximum over all outcomes $j \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$.

Let $X_n^{(j)}$ denote the run length process for outcome

$j \in \{0, 1, \ldots, \log_{1/p}(n) - 1\}$. Observe that the probability of outcome j is $p^j(1 - p)$ and the outcome $j + 1$ is $p^{j+1}(1 - p)$. Thus, the ratio is

$$\frac{p^j(1 - p)}{p^{j+1}(1 - p)} = \frac{1}{p},$$

which is independent of j. In turn, the fixed point $a = \frac{2(1+p)}{p}$ is identical (up to a constant additive error) for every outcome j. Therefore, for each j, we have

$$\Pr\left[\max_{0 \leq t_0 \leq n} X_t^{(j)} \geq a + k\right] \leq C e^{-\lambda^* k}.$$

A naive union bound would suggest

$$\Pr\left[\max_{j=0}^{\log_{1/p}(n)-1} \max_{0\le t_0\le n} X_t^{(j)} \ge a+k\right] \le \log_{1/p}(n) C e^{-\lambda^* k}.$$

However, using a standard chaining and peeling argument [96] we can show that in fact, there exist constants $C', \lambda' > 0$ (depending only on p) such that

$$\Pr\left[\max_{j=0}^{\log_{1/p}(n)-1} \max_{0\le t_0\le n} X_t^{(j)} \ge a+k\right] \le C' e^{-\lambda' k},$$

or equivalently, for any $\epsilon > 0$,

$$\Pr\left[\max_{j=0}^{\log_{1/p}(n)-1} \max_{0\le t_0\le n} X_t^{(j)} \ge (1+\epsilon)a\right] \le C' e^{-\lambda' \epsilon a}.$$

In practice, we simply take $C' \approx C$ and $\lambda' \approx \lambda^* = \lambda(p, B)$ (the same constants as above for outcome 0), as there is at most a negligible difference between the bound for different outcomes in $\{0, 1, \ldots, \log_{1/p}(n) - 1\}$ and the bound is maximized at outcome 0. Then, by choosing $B = a$, we obtain our result in the lemma. □

Lemma 5-3 (Overall Robust Skip List Search Path Cost). *For $\epsilon > 0$ and $a = \frac{2(1+p)}{p}$, let S be the total search path cost of the robust skip list, , then*

$$\Pr[S \ge a(1+\epsilon)\log_{1/p}(n)] \le e^{(\lambda^* a)-(\epsilon\lambda^* a)},$$

where λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \le 1.$$

Proof. The lemma directly follows from Lemma 5-2, and the fact that

$$S = \sum_{j=0}^{\log_{1/p}(n)} X^{(j)}$$
$$\leq \sum_{j=0}^{\log_{1/p}(n)} max_{j=0}^{\log_{1/p}(n)-1} \max_{0\leq t_0\leq n} X_t^{(j)}.$$

□

Next, we bound the number of elements in a skip list above level $L(n) = \log_{1/p}(n)$, addressing the second point in Lemma 5-1.

Lemma 5-4 (Bound on the Size of the list Above Level $L(n)$)**.** *Given a (robust) skip list with probability parameter p, let H be the number of elements that appear at heights* $\geq L(n) = \log_{1/p}(n)$*. That is, H counts the total occurrences of elements at or above height* $L(n)$ *resp. the total size of the skip list at or above height* $L(n)$*. Then,*

$$\mathbb{E}[H] = \frac{1}{1-p},$$

and

$$\Pr\left[H \geq w \cdot \log_{1/p}(n)\right] \leq e^{-\frac{((1-p)w\log_{1/p}(n)-1)^2}{(1-p)(2+(1-p)w\log_{1/p}(n)-1)}}.$$

Proof. Let $H_i \sim \mathrm{Bin}(n, p^i)$ denote the random variable describing the number of elements on level i among the n elements, such that $\mathbb{E}[H_i] = np^i$. Let $H = \sum_{i\geq L(n)} H_i$ be the total number of times an element appears at some level in all levels above level $L(n) = \log_{1/p}(n)$. Then,

$$\mathbb{E}[H] = \sum_{i\geq L(n)} np^i = np^{L(n)} \sum_{j\geq 0} p^j = np^{L(n)} \frac{1}{1-p}.$$

Now, observe that $p^{L(n)} = p^{\log_{1/p}(n)} = n^{\log_{1/p} p} = n^{-1}$, in turn $np^{L(n)} = n \cdot p^{L(n)} = n \cdot \frac{1}{n} = 1$. Therefore, the expected number of elements that appear at any level on or above level $L(n)$ is

$$\mathbb{E}[H] = \frac{1}{1-p}.$$

We then obtain a tail bound via the standard Chernoff bound, solving for a value δ[3] such that $(1+\delta)\left(\frac{1}{1-p}\right) = w\log_{1/p}(n)$. This completes the proof. The value $\delta = (1-p)w\log_{1/p}(n) - 1$ works, and we get the upper bound of $\exp\left(-\frac{\delta^2}{(1-p)(2+\delta)}\right)$. □

In our robust skip list, we define a maximum level m. This, in turn, defines the capacity of the list n by solving $m = \log_{1/p}(n) = L(n)$. So, in reality, this result actually reflects the maximum number of elements on level $L(n)$. To win in our game, the adversary must either craft a structure such that the total search path cost below level $L(n)$ exceeds $w\log_{1/p}(n)$ or the above "bad" event happens where there exists more than $w\log_{1/p}(n)$ elements on level $L(n)$. Combining these results gives us the following theorem.

Theorem 5-2 (Robust Skip List AAPC Security Result)**.** *Let Π be the robust skip list from Figure 5-10 with parameters $p \in [0,1]$ and $m = \log_{1/p}(q_U)$. For integers $q_U, q_Q, t \geq 0$, it holds that Π is $(\phi, \beta, \epsilon, \delta, t)$-conserved with ϕ being the Maximum Search Path Cost function (Figure 5-7), $\beta = c\log_{1/p}(n)$, $\epsilon > 0$, and*

$$\delta = e^{(\lambda^* a)-(\epsilon\lambda^* a)} + e^{-\frac{((1-p)a\log_{1/p}(n)-1)^2}{(1-p)(2+(1-p)a\log_{1/p}(n)-1)}},$$

where $a = \frac{2(1+p)}{p}$, $c = a(\epsilon+1)$, and λ^* is the maximal solution $\lambda > 0$ to

$$(1-p)e^{\lambda} + p(1-p)e^{-\lambda\left(\frac{1}{p}+\frac{a}{2}\right)} + p^2 \leq 1.$$

[3] Here δ refers to the usual difference from the mean in the Chernoff bound, not the parameter of the AAPC security notion.

Proof. The theorem directly follows from the observation deletions do not help the adversary (as in Theorem 5-1) and Lemma 5-1, Lemma 5-3, and Lemma 5-4. □ □

To give a concrete illustration of this bound, suppose we had $n = 2^{32}$, $p = \frac{1}{2}$. Then our fixed point $a = 6$, and solving for λ^* numerically yields $\lambda^* \approx 0.34$. Then, choosing $\epsilon = 8$ (hence, $c = 54$), the probability that the maximum search cost path exceeds $2 * c \log_{1/p}(n) = 108 \log_{1/p}(n)$ is less than or equal to $\delta \approx 6.28 \times 10^{-7}$. While a constant $2c = 108$ may appear large, consider that, λ^* solely depends on p. In turn, for any fixed ϵ this bound is constant as $n \to \infty$, showing that our adaptive search path is indeed $O(\log n)$. We further, remark that this constant is likely "artificially" large, in the sense that the stochastic process we bound is complex, leaving us only to be able to use blunt Markov-like concentration bounds.

5.5.3 Robust Skip Lists in Real World Deployments

Skip lists, like hash tables, have an explicit capacity for a set number of elements and require resizing when exceeded. While skip lists do not require upfront memory allocation, they require setting a maximum node height of $m = \log_{\frac{1}{p}} n$ for expected n insertions. Exceeding n insertions necessitates resizing as the probabilistic guarantees otherwise deteriorate [33]. Our security analysis avoids considering attacks that trigger re-initialization by enforcing $m = \log_{\frac{1}{p}} q_U$/ We now evaluate how our modifications affect resizing frequency in practice.

Standard skip lists with I successful insertions and D successful deletions require resizing when $\log_{\frac{1}{p}}(I - D) > m$. Previously, structures could operate indefinitely without resizing if insertion and deletion rates remained balanced. Our modified structure, which merely marks elements as "deleted" without allowing replacement, requires resizing when $\log_{\frac{1}{p}}(I) > m$ regardless of remaining elements. This allows adversaries to trigger early resizing by inserting approximately $\left(\frac{1}{p}\right)^m$ elements, deleting most, then forcing a resize with few additional insertions.

Unlike hash tables, we cannot replace deleted nodes without creating a security vulnerability where adversaries could manipulate the skip list by strategically deleting elements and inserting new ones, effectively repositioning unfavorable heights in other parts of the skip list – essentially enhancing our gap attack. While our approach requires more frequent resizing, this represents an essential trade-off ensuring provable robustness against adaptive adversarial attacks while preserving expected performance characteristics.

5.6 Robust Treaps

5.6.1 (In)Security of the Standard Treap

Unlike other probabilistic data structures in this study, treaps (without deletions) demonstrate intrinsic security against search path cost blow-up. However, adaptive adversaries can still mount attacks that force certain elements to be near the root with high probability.

Consider a lottery system designed to select a number of winners with uniform probability from a participant pool. Imagine, the implementation uses a treap data structure with an in-order traversal limited to a constant path length, thereby theoretically ensuring equal selection probability for all participants.

However, this implementation contains a critical security vulnerability against adaptive adversaries. While attackers cannot directly manipulate the random priority values assigned to entries, they can execute a more sophisticated attack by strategically inserting elements with carefully chosen keys positioned adjacent to a target element. By continuing this insertion pattern until placing an element with exceptionally low priority, they force the treap to perform rotation operations that elevate their target element toward the root. Since elements closer to the root are more likely to be selected during the limited-depth traversal, this compromises the lottery's fairness.

Concretely, let $x_1, x_2, \ldots, x_{j-1}$ be the keys inserted in sorted order with associated priorities

$$r^{x_1}, r^{x_2}, \ldots, r^{x_{j-1}},$$

drawn independently from the uniform distribution on $[0, 1]$. An adaptive adversary selects an arbitrary target element x_i. The adversary then repeatedly inserts new elements into the gaps between x_i and x_{i+1} and between x_{i-1} and x_i until obtaining an exceptionally low priority value. This is expected to occur after a linear number of insertions.

After inserting, let $S^n_{x_i}$ denote the search path to x_i. Since

$$S^n_{x_i} = |\{\text{records in sequence } r^{x_i}, r^{x_{i-1}}, \ldots, r^{x_1}\}| + |\{\text{records in sequence } r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_n}\}| - 1,$$

and the number of records in these intervals is constant (as the neighboring elements to x_i have exceptionally low priorities for which there exists only a constant number of nodes with lower priorities), x_i now resides near the top of the treap with high probability.

Importantly, for our purposes, treaps maintain their expected $O(\log n)$ operational complexity against adaptive adversaries only when our modified deletion procedure is applied. Without it, an adversary could simply re-insert (and delete) an element until obtaining a favorable priority, making degeneration attacks trivial.

This resistance to performance degradation attacks under lazy deletion represents a significant finding, as all other PSDS examined proved vulnerable. The treap's rebalancing mechanism, based on previously sampled priorities, provides a natural defense against malicious attempts to create operation sequences that would otherwise lead to worst-case runtime scenarios.

5.6.2 A Robust Construction

We give a pseudocode description of the robust treap using our modified deletion procedure in Figure 5-12. We will formally show the security (with regard to the maximal search path) of our modified-deletion treap. However, we first formalize a view of the treap's representation via a stochastic process. We start by analyzing the representation formed by a non-adaptive adversary and the subsequent maximum search path cost.

Consider a treap containing n elements inserted by a non-adaptive adversary, i.e., selected uniformly at random from the universe of all possible elements. Consider all inserted elements in the sorted order of their key value $x_1 \leq x_2 \leq \ldots \leq x_n$. Each key x_i is assigned a random priority r^{x_i} drawn independently from the uniform distribution on $[0, 1]$.

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

$\mathsf{T.root} \leftarrow \mathsf{null}$
for $(x, v) \in \mathcal{S}$ **do**
  $\mathsf{T} \leftarrow \textsc{Up}_K(\mathsf{T}, \mathsf{ins}_{(x,v)})$
return T

$\textsc{RandomPriority}_K(\boxed{x})$

$\boxed{p \leftarrow R(K, x)}$
$\boxed{\textbf{return } p}$
$p \twoheadleftarrow (0, 1)$
return p

$\textsc{NewNode}((x, v), p)$

/ array position 0 is reserved for
/ a deleted bit, key, value triple (d, x, v)
/ accessible via $n.\mathsf{del}$,, $n.\mathsf{key}$ and $n.\mathsf{value}$
/ array positions 2, 3 are child pointers and 1 is priority
$\mathsf{node} \leftarrow [(\bot, x, v), p, \mathsf{null}, \mathsf{null}]$
return node

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

$\mathsf{T.root} \leftarrow \textsc{Qry}^{\mathrm{rec}}(\mathsf{T.root}, \mathsf{qry}_x)$
return T

$\textsc{Qry}^{\mathrm{rec}}(c, \mathsf{qry}_x)$

if $c = \mathsf{null}$ **then**
  return $\star$
if $c[0].\mathsf{key} = x$ **then**
  return $c[0].\mathsf{key}$
$b \leftarrow (x > c[0].\mathsf{key})$
return $\textsc{Qry}^{\mathrm{rec}}(c[2 + b], \mathsf{qry}_x)$

$\textsc{Rotate}(c, b)$

$\mathsf{tmp} \leftarrow c[2 + b][3 - b]$
$c[2 + b][3 - b] \leftarrow c$
$c[2 + b] \leftarrow \mathsf{tmp}$
return tmp

$\textsc{Up}_K(\mathsf{T}, \mathsf{ins}_{(x,v)})$

$\mathsf{T.root} \leftarrow \textsc{Up}_K^{\mathrm{rec}}(\mathsf{T.root}, \mathsf{ins}_{(x,v)})$
return T

$\textsc{Up}_K^{\mathrm{rec}}(c, \mathsf{ins}_{(x,v)})$

if $c = \mathsf{null}$ **then**
  $p \leftarrow \textsc{RandomPriority}_K(\boxed{x})$
  return $\textsc{NewNode}((x, v), p)$
if $c[0].\mathsf{key} = x$ **then**
  $c[0].\mathsf{value} \leftarrow v,\ c[0].\mathsf{key} \leftarrow \bot$
  return c
$b \leftarrow (x > c[0].\mathsf{key})$
$c[2 + b] \leftarrow \textsc{Up}_K^{\mathrm{rec}}(c[2 + b], \mathsf{ins}_{(x,v)})$
  / maintain MIN Heap property
if $c[1] > c[2 + b][1]$ **then**
  $c \leftarrow \textsc{Rotate}(c, b)$
return c

$\textsc{Up}_K(\mathsf{T}, \mathsf{del}_x)$

$\textsc{Up}_K^{\mathrm{rec}}(\mathsf{T.root}, \mathsf{del}_x)$
return T

$\textsc{Up}^{\mathrm{rec}}(c, \mathsf{del}_x)$

if $c = \mathsf{null}$ **then**
  return
if $c[0].\mathsf{key} = x$ **then**
  / Remove node
  $c[0].\mathsf{del} \leftarrow \top$
else
  $b \leftarrow (x > c[0].\mathsf{key})$
  $\textsc{Up}_K^{\mathrm{rec}}(c[2 + b], \mathsf{del}_x)$
return

Figure 5-12. A robust, possibly "deterministic" (and keyed) robust MIN treap structure $\mathrm{TR}[\boxed{R}]$ admitting insertions, deletions, and queries for any $x \in \mathcal{U}$ for some well-ordered universe $\mathcal{U}$. The parameter is a keyed function $R : \mathcal{K} \times \mathcal{U} \to (0, 1))$ that assigns an element a random priority. Subroutines used by the deterministic version of the structure appear in the boxed environment. Let $\textsc{MinPrioChild}(c)$ denote the function that returns the child index (0 or 1) of node c with the minimum priority, or null if c has no children.

Let $S^n_{x_i}$ denote the random variable representing the search path length for a fixed element x_i. The search path to element x_i consists of all ancestors of x_i in the treap structure. From Aragon and Seidel [90], x_j is an ancestor of x_i if and only if x_j has the lowest priority among all elements between x_i and x_j (inclusive). Specifically:

- If $j > i$, then x_j is an ancestor of x_i if and only if $r^{x_j} = \min\{r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_j}\}$
- If $j < i$, then x_j is an ancestor of x_i if and only if $r^{x_j} = \min\{r^{x_j}, r^{x_{j+1}}, \ldots, r^{x_i}\}$

This means that an element is an ancestor of x_i precisely when its priority is a minimum value – what we will refer to as a "record" – in one of two sequences extending from x_i. Hence, we can interpret $S^n_{x_i}$ as:

$$
\begin{aligned}
S^n_{x_i} = &|\{\text{records in sequence } r^{x_i}, r^{x_{i-1}}, \ldots, r^{x_1}\}|+ \\
&|\{\text{records in sequence } r^{x_i}, r^{x_{i+1}}, \ldots, r^{x_n}\}| - 1,
\end{aligned}
$$

where the subtraction of 1 accounts for x_i being counted in both sequences.

A classical fact about random permutations is the behavior of records. For a sequence of k i.i.d. uniformly distributed random variables, the probability that the j-th element is a record (i.e., it is less all $j-1$ preceding values is exactly $\frac{1}{j}$). More precisely, define the following indicator variables for a given sequence:

$$
I_j = \begin{cases} 1, & \text{if the } j\text{-th element is a record,} \\ 0, & \text{otherwise.} \end{cases}
$$

Then we have $\mathbb{E}[I_j] = \frac{1}{j}$. For a sequence of length k, the total number of records is $R_k = \sum_{j=1}^{k} I_j$ and its expectation is $\mathbb{E}[R_k] = \sum_{j=1}^{k} \frac{1}{j} = H_k$, where H_k is the k-th harmonic number. In our context, when considering the "leftward" sequence of priority values L_i (of length i) and the "rightward" sequence if priority R_i (of length $n-i+1$) with respect to key at index i, we have $\mathbb{E}[L_i] = H_i$ and $\mathbb{E}[R_i] = H_{n-i+1}$.

Thus, for a fixed x_i,

$$\mathbb{E}[S^n_{x_i}] = \mathbb{E}[L_i + R_i - 1] = H_i + H_{n-i+1} - 1.$$

Nothing, that for any i it must be that $H_i \leq H_n$ and $H_{n-i+1} \leq H_n$, and the well known fact $H_n \leq \ln(n) + 1$, we have

$$\begin{aligned}\mathbb{E}[S^n_{x_i}] &\leq 2H_n - 1 \\ &\leq 2\ln(n) + 1.\end{aligned}$$

We next argue that even when an adaptive adversary determines the insertions, each inserted element's probability of forming a record remains exactly $\frac{1}{j}$ when it is the j-th element inserted – exactly the same as the non-adaptive case. Even though an adaptive adversary can observe all previous outcomes and choose the next element adaptively (that is, select the key value so it falls into any "gap" of existing key values, like in the case of the skip list in Section 5.5), the new priority is still drawn uniformly and independently from $[0, 1]$. The joint distribution of the prior priorities is unchanged. We formalize this idea in the following lemma.

Lemma 5-5 (Invariant Record Probability under Adaptive Insertion)**.** *Let $x_1, x_2, \dots, x_{j-1}$ be the keys inserted in sorted order with associated priorities*

$$r^{x_1}, r^{x_2}, \dots, r^{x_{j-1}},$$

drawn independently from the uniform distribution on $[0, 1]$. An adaptive adversary (chooses a gap (i.e., a position between any two or before/after these keys) into which to insert a new key x_j. The new key receives an independent priority $r^{x_j} \sim \mathsf{U}[0, 1]$. After relabeling the keys according to *their inherent order, let the sorted sequence of priorities (of all j keys) be*

$$r_{(1)} \le r_{(2)} \le \cdots \le r_{(j)}.$$

Then, even conditioned on the past σ-algebra $\mathcal{F}_{j-1}$ (which contains the ordered priority values and all adversarial decisions regarding the first $j-1$ insertions), we have

$$\Pr\left(r^{x_j} = r_{(1)} \,\middle|\, \mathcal{F}_{j-1}\right) = \frac{1}{j}.$$

Proof. Condition on the σ-algebra $\mathcal{F}_{j-1}$; that is, assume the priorities

$$r^{x_1}, r^{x_2}, \ldots, r^{x_{j-1}}$$

are fixed and rearranged in increasing order:

$$r_{(1)} \le r_{(2)} \le \cdots \le r_{(j-1)}.$$

An adaptive adversary may insert the new key x_j in any gap between any two keys in the current sequence (or before the smallest or after the largest). Still, such a decision affects only the position of the key in the *key order* and does not alter the statistical properties of the newly drawn priority.

The new priority r^{x_j} is drawn independently from $\mathsf{U}[0,1]$. Thus, when the new key is inserted, the complete set of j priorities is

$$\{r^{x_j}, r_{(1)}, r_{(2)}, \ldots, r_{(j-1)}\}.$$

Since the first $j-1$ values are already fixed and r^{x_j} is independent and uniformly distributed over $[0,1]$, the resulting set of j priorities is exactly equivalent to a set of j independent uniform samples upon relabeling.

In any sequence of j i.i.d. $\mathsf{U}[0,1]$ random variables, symmetry implies that the probability that any particular one (here, the newly inserted element) is the minimum is exactly $1/j$. Formally, we have

$$\Pr\left(r^{x_j} = \min\{r^{x_j}, r_{(1)}, r_{(2)}, \ldots, r_{(j-1)}\} \,\middle|\, \mathcal{F}_{j-1}\right) = \frac{1}{j}.$$

Thus, regardless of where the adversary chooses to insert x_j, the probability that x_j is a record (i.e., its priority is the smallest among the first j keys) remains equal to $1/j$, as in the non-adaptive setting. □

Theorem 5-3 (Treap AAPC Result)**.** *Let* Π *be the robust treap from Figure 5-12. For integers* $q_U, q_Q, t \geq 0$*, it holds that* Π *is* $(\phi, \beta, \epsilon, \delta, t)$*-conserved with* ϕ *being the Maximum Search Path Cost function (Figure 5-6),* $\beta = 2\ln n + 1$*, any* $\epsilon > 0$ *and*

$$\delta = ne^{-\frac{\epsilon^2 H_n}{2(1+\epsilon)}},$$

where $n = q_U$ *and* H_n *is the n-th harmonic number.*

Proof. Observe that deletions do not help the adversary, as by construction, they at most relabel an existing entry and cannot possibly extend the longest path. Therefore, we consider a treap of $n = q_U$ keys (i.e., a treap with the maximal number of insertions made) built by an adaptive adversary.

Casting the Insertion Process as a Doob Martingale.

For a fresh key inserted at step j, take the indicator variable I_j as defined above. Then, conditioned on the past σ-algebra $\mathcal{F}_{j-1}$ (which contains the ordered priority values and all adversarial decisions regarding the first $j-1$ insertions), and letting

$$m_{j-1} := \min\{r^{x_1}, r^{x_2}, \ldots, r^{x_{j-1}}\} \quad (\text{with } m_0 = 1),$$

it is easy to see that

$$\Pr(I_j = 1 | \mathcal{F}_{j-1}) = \Pr(r^{x_j} < m_{j-1} | \mathcal{F}_{j-1}) = m_{j-1}.$$

From Lemma 5-5, we have that even under adaptive insertions, the unconditional expectation remains

$$\mathbb{E}[m_{j-1}] = \frac{1}{j}.$$

Then, if letting X_n denote the total number of records over all n insertions, the unconditional expected number of records is

$$\mathbb{E}[X_n] = \sum_{j=1}^{n} \mathbb{E}[I_j] = \sum_{j=1}^{n} \frac{1}{j} = H_n,$$

where H_n is the n-th harmonic number.

From our above analysis, we have that the search path length S_x^n for key x is bounded in terms of the number of records X_n by

$$S_n^x \leq 2X_n - 1.$$

Thus, if we can show that X_n is concentrated around H_n, we also have a bound on the search cost for a particular element. To do this, define the Doob martingale

$$M_j = \sum_{i=1}^{j} (I_i - \mathbb{E}[I_i | \mathcal{F}_{i-1}]), \quad j = 0, 1, 2, \ldots, n,$$

with $M_0 = 0$. By construction, $\{M_j\}$ is a martingale relative to the filtration $\{F_j\}$.

Next, observe that the martingale difference satisfy

$$D_j = M_j - M_{j-1} = I_j - \mathbb{E}[I_i | \mathcal{F}_{i-1}].$$

Since $I_j \in \{0, 1\}$ and $\mathbb{E}[I_i | \mathcal{F}_{i-1} \in [0, 1]$, we have $|D_j| \leq 1$.

Since $I_j \in \{0, 1\}$ is a Bernoulli random variable with parameter m_{j-1}, its conditional expectation is

$$\mathbb{E}[I_j \mid \mathcal{F}_{j-1}] = m_{j-1},$$

and the conditional variance is computed as:

$$\begin{aligned}\operatorname{Var}(I_j \mid \mathcal{F}_{j-1}) &= \mathbb{E}\Big[(I_j - m_{j-1})^2 \,\Big|\, \mathcal{F}_{j-1}\Big] \\ &= m_{j-1}(1 - m_{j-1}).\end{aligned}$$

Now, note that subtracting the constant $\mathbb{E}[I_j \mid \mathcal{F}_{j-1}]$ does not change the variance. That is,

$$\begin{aligned}\operatorname{Var}(D_j \mid \mathcal{F}_{j-1}) &= \operatorname{Var}(I_j - \mathbb{E}[I_j \mid \mathcal{F}_{j-1}] \mid \mathcal{F}_{j-1}) \\ &= \operatorname{Var}(I_j \mid \mathcal{F}_{j-1}) \\ &= m_{j-1}(1 - m_{j-1}).\end{aligned}$$

Thus, computing the predictable quadratic variation, we have

$$V_n = \sum_{j=1}^{n} \operatorname{Var}(D_j \mid \mathcal{F}_{j-1}) = \sum_{j=1}^{n} m_{j-1}(1 - m_{j-1}).$$

Since $m_{j-1}(1 - m_{j-1}) \leq m_{j-1}$ (because $1 - m_{j-1} \leq 1$ for all $m_{j-1} \in [0, 1]$), we obtain

$$V_n \leq \sum_{j=1}^{n} m_{j-1}.$$

Further, as $E[m_{j-1}] = \frac{1}{j}$, we have

$$\mathbb{E}[V_n] \leq \sum_{j=1}^{n} \frac{1}{j} = H_n.$$

Applying A Concentration Bound.

Freedman's inequality [97] states that if $\{M_j\}$ is a martingale with a difference bounded by 1 with predictable quadratic variation V_n, then for any $a, b > 0$,

$$\Pr(M_n \geq a \text{ and } V_n \leq b) \leq e^{-\frac{a^2}{2(a+b)}}.$$

We set $b = H_n$ (as typically V_n will not exceed H_n by much) and chose $a = \epsilon H_n$, where $\epsilon > 0$ is our parameter from our security statement.

Then, Freedman's inequality gives us

$$\Pr(M_n \geq \epsilon H_n) \leq e^{-\frac{\epsilon^2 H_n^2}{2(\epsilon H_n + H_n)}} = e^{-\frac{\epsilon^2 H_n}{2(1+\epsilon)}}.$$

Since

$$X_n = \sum_{j=1}^{n} I_j = M_n + \sum_{j=1}^{n} \mathbb{E}[I_j | \mathcal{F}_{j-1}],$$

and $\sum_{j=1}^{n} \mathbb{E}[I_j|\mathcal{F}_{j-1}]$ has expectation H_n, the above inequality shows that with probability at least $1 - e^{-\frac{\epsilon^2 H_n}{2(1+\epsilon)}}$ we have $X_n \leq (1+\epsilon)H_n$. Further, recalling that $S_x^n \leq 2X_n - 1$, this implies with the same probability, $S_x^n \leq 2(1+\epsilon)H_n - 1$.

Bounding the Search Cost Path Over All Elements.

Let E_x be the event that the search path cost for a fixed element exceeds the threshold $T = 2(1+\epsilon)\ln(n) + 1$.

From the above, we have that

$$\Pr(E_x) \leq e^{-\frac{\epsilon^2 H_n}{2(1+\epsilon)}},$$

as $H_n \leq ln(n) + 1$.

Then, applying a standard union bound over all the n elements in the treap, the event that there exists some element with a search path cost exceeding T is bounded by

$$\Pr\left(\bigcup_{x \in \{x_1, \ldots, x_n\}} E_x\right) \leq n e^{-\frac{\epsilon^2 H_n}{2(1+\epsilon)}}.$$

□

To give a concrete illustration of this bound, suppose we had $n = 2^{32}$ and select $\epsilon = 5$. Our expected search path cost is $2\ln(2^{32}) + 1 \approx 45.36$, and leveraging our results from Theorem 5-3 the probability the maximum search cost path exceeds this by five times is $\leq \delta = 2^{32} \cdot e^{-\frac{25H_{2^{32}}}{12}} \approx 6.65 \times 10^{-12}$.

5.6.3 Robust Treaps in Real World Deployments

Unlike hash tables and skip lists, treaps operate without an implicit maximum capacity and typically don't require resizing operations. However, our modified structure – which only marks elements as "deleted" without allowing replacement – may still necessitate periodic treap re-initialization to reclaim memory occupied by deleted elements. This limitation exists because allowing the replacement of deleted nodes would create a security vulnerability, enabling attackers to strategically shift unfavorable priorities to different parts of the treap. Similar to our skip list approach, we cannot simply reuse deleted nodes without compromising security. While this design choice increases maintenance overhead compared to standard treaps, it represents an essential trade-off that ensures provable robustness against adversarial attacks while maintaining the treap's expected performance characteristics in adversarial environments.

5.7 Experimental Results

We conducted experiments to empirically validate our analytical results. Our first experiment tested whether robust data structures offer benefits in non-adversarial settings. Using a dataset of 10 million usernames [98], we randomly inserted 1,000 usernames into each data structure. We measured performance by counting hops (forward movements between nodes). The hash table's load factor was limited to 0.7 [99], and the skip list's maximum height was set to $log_2 n$. We used Python's built-in hash function, which is vulnerable to multi-collision attacks. Results were averaged over 100 trials.

As shown in Figure 5-13, the robust skip list consistently required fewer mean and maximum hops than its standard counterpart, demonstrating benefits even in non-adversarial settings with only constant overhead. The robust hash table showed comparable performance to the original structure. We benchmarked an unmodified treap implementation given its inherent adversarial robustness.

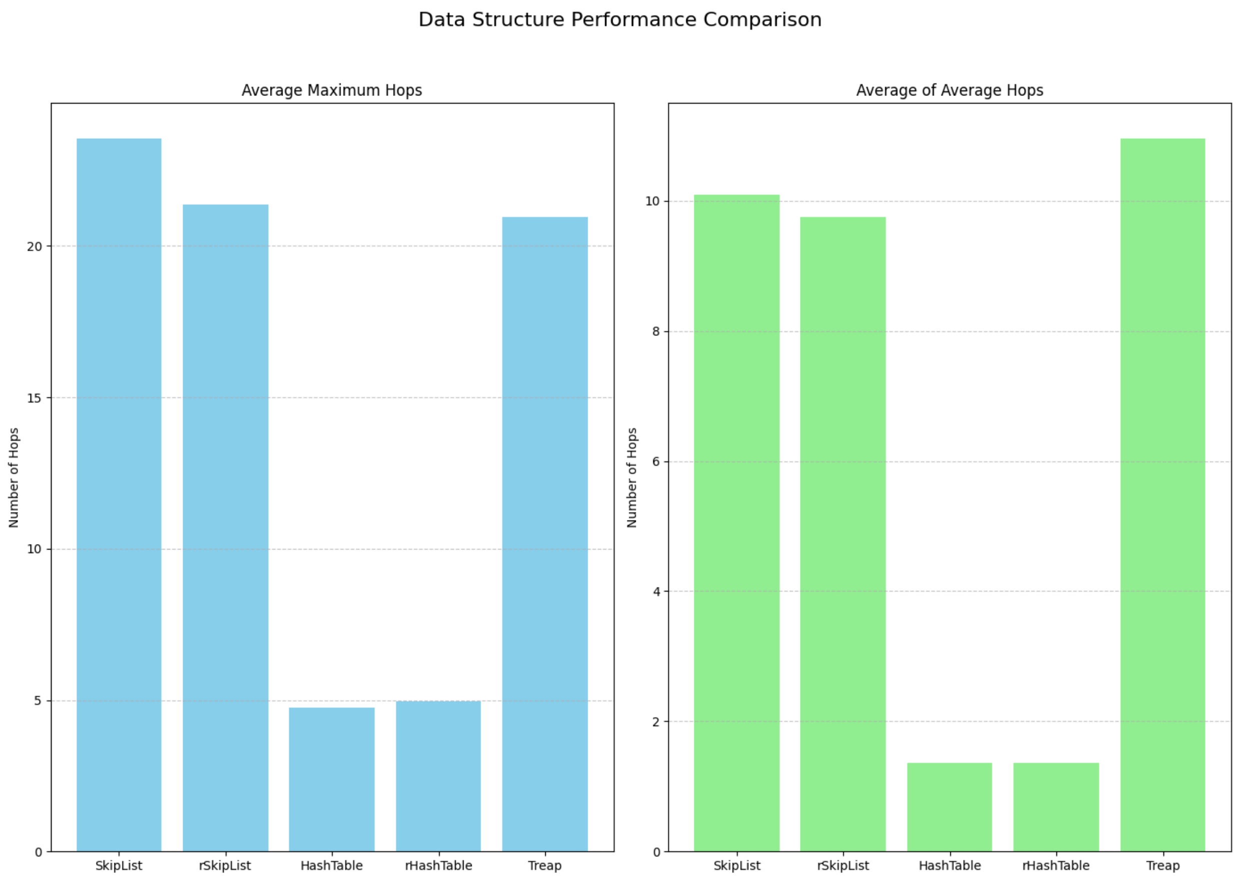


Figure 5-13. Maximum and average hop count in the non-adaptive setting, displayed on a linear scale.

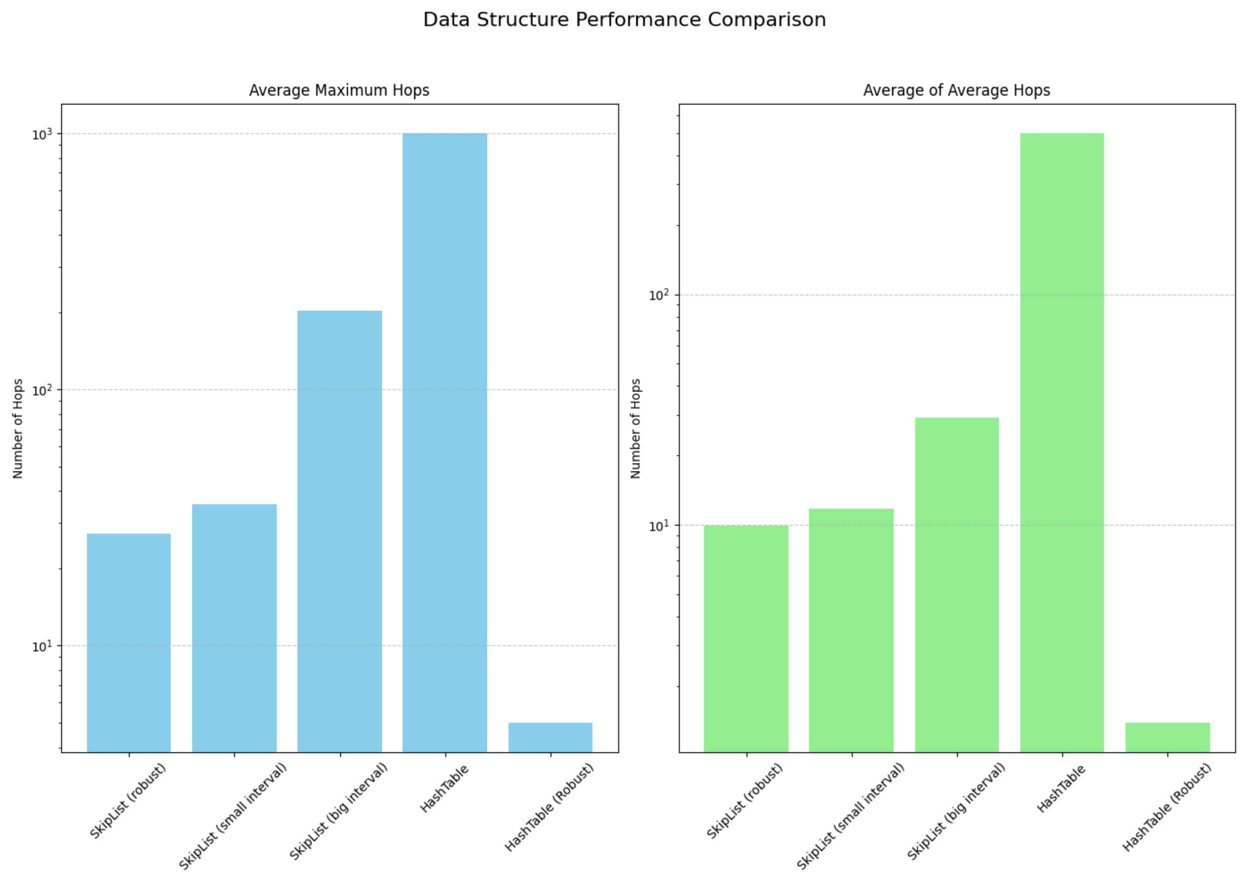


Figure 5-14. Maximum and average hop count in the adaptive setting, displayed on a logarithmic scale.

Our second experiment evaluated performance under adaptive adversarial conditions. We implemented a hash collision attack on hash tables and a gap attack on skip lists, averaging results over 100 trials. Treaps were excluded due to their established inherent robustness.

For the hash collision attack, we pre-calculated bucket values to deliberately insert all elements into a single bucket, creating worst-case conditions for standard hash tables. For the robust implementation, we used a random key for pre-calculation, since the actual secret key would be unknown to an attacker.

For the gap attack against skip lists, we tested two variants: a restricted version using the same username dataset and an unrestricted version using integers within the range $[0, 10^{100}]$[4]. We report the top 1% of outcomes with respect to the maximum hop count.

Results in Figure 5-14 confirm that adversarial attacks significantly degrade standard implementations, while robust counterparts maintain consistent performance. The robust skip list maintained an average maximum hop count of 27.36, compared to 33.17 for the non-robust implementation under non-adaptive conditions. Under adaptive settings, the non-robust implementation degraded to 35.61 maximum average hops, and further to 202.71 hops when using the larger integer range. This validation confirms our theoretical findings on adversarial robustness, and also suggests that our remark regarding the artificial "looseness" of the bound carries weight.

[4]While this serves primarily as a proof of concept, such a vast interval could realistically be achieved using a 20-character limit with Unicode encoding. We emphasize that significant runtime degradation can be observed even with substantially smaller intervals.

CHAPTER 6
CONCLUSION AND FUTURE WORK

This dissertation contributes to a growing body of work that seeks to formalize the security of data structures, bridging the gap between theoretical designs and practical implementations. By rigorously analyzing compact frequency estimators and probabilistic skipping-based data structures under adversarial models, this work demonstrates that the assumptions underlying traditional performance guarantees often collapse in adversarial environments.

The results show that existing data structures, including Count-min sketch, HeavyKeeper, hash tables, skip lists, and treaps, are susceptible to adaptive attacks that can severely degrade correctness and performance. The development of Count-Keeper and the modified PSDS variants provides a concrete pathway toward provably secure, efficient, and practically deployable structures. These constructions are not merely theoretical but are supported by formal proofs and empirical evaluations, highlighting their resilience against adaptive adversaries.

The Redis case study underscores the importance of bridging theory and practice. By exposing vulnerabilities in Redis's CFE implementations, this work demonstrates how small design decisions, such as the use of non-cryptographic hash functions, can open pathways for sophisticated attacks. The proposed countermeasures offer actionable steps toward more secure deployments of widely-used data structures.

In sum, this dissertation not only identifies vulnerabilities in widely adopted data structures but also presents formalized, provably secure alternatives. It offers a rigorous foundation for future work in designing efficient and secure data structures and serves as a call to incorporate adversarial considerations into core data structure design. Below, we highlight several promising directions inspired by this dissertation.

Our work on CFEs suggests that nearly all existing sketch-based frequency estimators may be susceptible to the kinds of attacks we present. While our proposed Count-Keeper structure mitigates the extent of damage an adaptive adversary can inflict and enables the flagging of potentially adversarially influenced estimates, it does not *prevent* adversaries from causing large frequency estimation errors. This aligns with recent theoretical work: Hardt and Woodruff [100], Cohen et al.[101], and Ben-Eliezer et al.[102] have shown that linear sketches (including CMS but not HK) are fundamentally non-robust against well-resourced adaptive attacks, particularly in various L_p-norm estimation tasks such as solving the k-heavy-hitters problem relative to the L_2-norm.

Thus, a significant open problem is the design of a CFE that not only detects adversarial manipulation but also outright prevents it. While Hassidim et al. [103] explore enhancing streaming algorithms with differential privacy to achieve robustness, their approach is impractical for real-world deployments in our setting. Either the query responses would become excessively lossy, or the number of queries allowed would need to be severely restricted to deter attacks. Developing a generically robust CFE structure remains an open and highly relevant challenge. Additionally, it may be fruitful to explore further compositions of CPDS, as we did with Count-Keeper. For instance, could one design more performant and robust approximate set-membership structures by combining (say) Bloom filters and Cuckoo filters?

The Redis case study also opens avenues for further investigation. Many other PDS suites are deployed in real-world systems, such as Google BigQuery and Apache Spark, and could be subjected to the same detailed security analysis applied here. While methods for provably securing PDS against attacks have been proposed [104, 8, 24, 11, 75, 10], these analyses often focus on textbook versions of CPDS. Extending these analyses to the specific implementations used in practice could greatly enhance confidence in their security.

More broadly, there is still a lack of understanding in the developer community about the risks of using CPDS in adversarial environments. Further work is needed to educate practitioners about these risks, and we hope this dissertation can contribute to this effort. Alternatively, to shield developers from these risks, one could develop new CPDS implementations that are secure by default, packaging them into user-friendly libraries with safe APIs. This effort could benefit from the experience of the cryptographic research community, which has developed "safe-by-default" cryptographic libraries.

Regarding PSDS, while our theoretical bounds provide rigorous security guarantees, there is scope to tighten these bounds further. Future work could extend our analysis to related data structures, such as zip trees [105], zip-zip trees [106], skip graphs [107], and randomized meldable heaps [108]. Additionally, exploring more sophisticated deletion strategies may reduce memory overhead while preserving security guarantees. One promising direction involves localized re-initializations, allowing portions of a structure to be safely rebuilt without compromising robustness. Finally, our Adaptive Adversary Property Conservation (AAPC) framework could be applied to broader classes of data structures and properties, given its inherent extensibility.

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

[4] 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.

[5] 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.

[6] “Bip 37.” https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki.

[7] M. Naor and E. Yogev, “Bloom filters in adversarial environments,” in *Annual Cryptology Conference*, 2015.

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

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

[10] M. Filić, K. Kocher, E. Kummer, and A. Unnikrishnan, “Deletions and dishonesty: Probabilistic data structures in adversarial settings,” in *International Conference on the Theory and Application of Cryptology and Information Security*, pp. 137–168, Springer, 2025.

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

[12] 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.

[13] S. A. Crosby and D. S. Wallach, “Denial of service via algorithmic complexity attacks,” in *USENIX Security*, 2003.

[14] A. Klink and J. Walde, “Efficient denial of service attacks on web application platforms,” in *28th Chaos Communication Congress*, 2011.

[15] J.-P. Aumasson, M. Boßlet, and D. J. Bernstein, "Hash-flooding dos reloaded:attacks and defenses." Slides presented at the 2011 Application Security Forum – Western Switzerland, October 2011.

[16] R. Rosen, "Netfilter," *Linux Kernel Networking: Implementation and Theory*, pp. 247–278, 2014.

[17] P. Bottinelli, "Technical advisory – hash denial-of-service attack in multiple quic implementations." https://www.nccgroup.com/us/research-blog/technical-advisory-hash-denial-of-service-attack-in-multiple-quic-implementation April 2025. NCC Group Research Blog.

[18] E. Nussbaum and M. Segal, "Skiplist timing attack vulnerability," in *International Workshop on Data Privacy Management*, pp. 49–58, Springer, 2019.

[19] 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, pp. 1–26, 2008.

[20] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[21] S. Tarkoma, C. E. Rothenberg, and E. Lagerspetz, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys & Tutorials*, vol. 14, no. 1, pp. 131–155, 2011.

[22] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM transactions on networking*, vol. 8, no. 3, pp. 281–293, 2000.

[23] 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.

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

[25] P. Reviriego and D. Ting, "Security of hyperloglog (hll) cardinality estimation: Vulnerabilities and protection," *IEEE Communications Letters*, vol. 24, no. 5, pp. 976–980, 2020.

[26] S. Albers and J. Westbrook, "Self-organizing data structures," *Online Algorithms: The state of the art*, pp. 13–51, 2005.

[27] D. D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[28] N. Reingold, J. Westbrook, and D. D. Sleator, "Randomized competitive algorithms for the list update problem," *Algorithmica*, vol. 11, no. 1, pp. 15–32, 1994.

[29] R. Bayer, “Symmetric binary b-trees: Data structure and maintenance algorithms,” *Acta informatica*, vol. 1, no. 4, pp. 290–306, 1972.

[30] G. M. Adel'son-Vel'skii, “An algorithm for the organization of information,” *Soviet Math.*, vol. 3, pp. 1259–1263, 1962.

[31] J. I. Munro, T. Papadakis, and R. Sedgewick, “Deterministic skip lists,” in *Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pp. 367–375, 1992.

[32] R. Seidel and C. R. Aragon, “Randomized search trees,” *Algorithmica*, vol. 16, no. 4, pp. 464–497, 1996.

[33] W. Pugh, “Skip lists: A probabilistic alternative to balanced trees,” *Commun. ACM*, vol. 33, p. 668–676, jun 1990.

[34] K. Mehlhorn and P. Sanders, “Hash tables and associative arrays,” *Algorithms and Data Structures: The Basic Toolbox*, pp. 81–98, 2008.

[35] J. Blandy, J. Orendorff, and L. F. Tindall, *Programming Rust.* ” O'Reilly Media, Inc.”, 2021.

[36] Z. István, G. Alonso, M. Blott, and K. Vissers, “A hash table for line-rate data processing,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 8, no. 2, pp. 1–15, 2015.

[37] J. Zobel, S. Heinz, and H. E. Williams, “In-memory hash tables for accumulating text vocabularies,” *Information Processing Letters*, vol. 80, no. 6, pp. 271–277, 2001.

[38] V. Paxson, “Bro: a system for detecting network intruders in real-time,” *Computer networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[39] N. Bar-Yosef and A. Wool, “Remote algorithmic complexity attacks against randomized hash tables,” in *International Conference on E-Business and Telecommunications*, pp. 162–174, Springer, 2007.

[40] D. Eckhoff, T. Limmer, and F. Dressler, “Hash tables for efficient flow monitoring: Vulnerabilities and countermeasures,” in *2009 IEEE 34th Conference on Local Computer Networks*, pp. 1087–1094, IEEE, 2009.

[41] J. L. Carter and M. N. Wegman, “Universal classes of hash functions,” in *Proceedings of the ninth annual ACM symposium on Theory of computing*, pp. 106–112, 1977.

[42] A. Appleby, “Smhasher.” https://github.com/aappleby/smhasher, 2016.

[43] 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.

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

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

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

[47] 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.

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

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

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

[51] 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.

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

[53] 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.

[54] 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.

[55] 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.

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

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

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

[59] 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.

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

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

[62] 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.

[63] “Probabilistic: Probabilistic data structures in Redis..” https://redis.io/docs/data-types/probabilistic/.

[64] 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.

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

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

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

[68] 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.

[69] S. Sanfilippo, “A few things about redis security..” http://antirez.com/news/96.

[70] 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.

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

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

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

[74] 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.

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

[76] 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.

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

[78] 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.

[79] 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.

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

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

[82] 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.

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

[84] M. Nowack. https://discord.com/blog/using-rust-to-scale-elixir-for-11-million-concurrent-users, 2019.

[85] Apache. https://vovkos.github.io/doxyrest/samples/apr-sphinxdoc/group_apr_skiplist.html, 2023.

[86] A. Prout. https://www.singlestore.com/blog/what-is-skiplist-why-skiplist-index-for-memsql/, 2019.

[87] A. Ambainis, "Quantum walk algorithm for element distinctness," in *45th Annual IEEE Symposium on Foundations of Computer Science*, pp. 22–31, 2004.

[88] P. Kisters, H. Bornholdt, and J. Edinger, "Skabnet: A data structure for efficient discovery of streaming data for iot," in *2023 32nd International Conference on Computer Communications and Networks (ICCCN)*, pp. 1–10, 2023.

[89] J. Zhao, Y. Pan, H. Zhang, M. Lin, X. Luo, and Z. Xu, "Inplacekv: in-place update scheme for ssd-based kv storage systems under update-intensive worklaods," *Cluster Computing*, 05 2023.

[90] C. R. Aragon and R. Seidel, "Randomized search trees," in *FOCS*, vol. 30, pp. 540–545, 1989.

[91] E. D. Demaine, J. Iacono, and S. Langerman, "Retroactive data structures," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, pp. 13–es, 2007.

[92] Y. Ji, E. Dubrova, and R. Wang, "Is your bluetooth chip leaking secrets via rf signals?," *Cryptology ePrint Archive*, 2025.

[93] S. Chawla, "Cs787: Advanced algorithms scribe notes, lecture 7: Randomized load balancing and hashing," September 2009.

[94] D. E. Knuth, *The art of computer programming. 2. Seminumerical algorithms*. Addison-Wesley, 1971.

[95] J. Ville, *Etude critique de la notion de collectif*, vol. 3. Gauthier-Villars Paris, 1939.

[96] S. Boucheron, G. Lugosi, and P. Massart, *Concentration Inequalities: A Nonasymptotic Theory of Independence*. Oxford University Press, 02 2013.

[97] D. A. Freedman, "On tail probabilities for martingales," *the Annals of Probability*, pp. 100–118, 1975.

[98] M. Burnett. https://medium.com/xato-security/today-i-am-releasing-ten-million-passwords-b6278bbe7495, 2015.

[99] M. T. McClellan and J. Minker, "The art of computer programming, vol. 3: sorting and searching," 1974.

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

[101] 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.

[102] 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.

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

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

[105] R. E. Tarjan, C. Levy, and S. Timmel, "Zip trees," *ACM Transactions on Algorithms (TALG)*, vol. 17, no. 4, pp. 1–12, 2021.

[106] O. Gila, M. T. Goodrich, and R. E. Tarjan, "Zip-zip trees: Making zip trees more balanced, biased, compact, or persistent," in *Algorithms and Data Structures Symposium*, pp. 474–492, Springer, 2023.

[107] J. Aspnes and G. Shah, "Skip graphs," *Acm transactions on algorithms (talg)*, vol. 3, no. 4, pp. 37–es, 2007.

[108] A. Gambin and A. Malinowski, "Randomized meldable priority queues," in *International Conference on Current Trends in Theory and Practice of Computer Science*, pp. 344–349, Springer, 1998.

BIOGRAPHICAL SKETCH

Sam Markelon is a computer science researcher specializing in cryptography and the use of randomization and probabilistic techniques in algorithms and data analysis. His research focuses on applying provable security frameworks to analyze probabilistic data structures under adversarial conditions.

Sam earned his Ph.D. in Computer Science from the University of Florida in 2025, under the guidance of Dr. Vincent Bindschaedler and Dr. Thomas Shrimpton. He completed his Bachelor's degree in Computer Science with a minor in Mathematics from the University of Connecticut in 2020, where he was named a Barry M. Goldwater Scholar. Upon defending his dissertation, Sam joined Proof Trading as a Quantitative Researcher.