Probabilistic Data Structures in the Wild: A Security Analysis of Redis

Mia Filić
ETH Zürich
mfilic@ethz.ch

Jonas Hofmann
ETH Zürich
hofmannj@student.ethz.ch

Sam A. Markelon
University of Florida
smarkelon@ufl.edu

Kenneth G. Paterson
ETH Zürich
kenny.paterson@inf.ethz.ch

Anupama Unnikrishnan
ETH Zürich
aunnikris@phys.ethz.ch

ABSTRACT

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

1 INTRODUCTION

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

A prominent example of such a system is Redis, a general purpose, in-memory database. Redis is integrated into general data analytics and computing platforms offered by AWS, Google Cloud, IBM Cloud, and Microsoft Azure, amongst others. Redis supports a variety of PDS: HyperLogLog (HLL), Bloom filter, Cuckoo filter, t-digest, Top-K, and Count-Min sketch [3]. While Redis was mostly used as a cache in the past, it is now a fully general system, used by a companies like Adobe [31], Microsoft [32], Facebook [29] and Verizon [30] for a variety of purposes. These include security-related applications, such as traffic analysis and intrusion detection systems [36].

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"* [35]. In reality, users failed to comply with this [18]. 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 blocklisting amongst other features [5]. Redis now also recognise 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 behaviour triggered by crafted inputs [5]. The first issue is prevented in Redis by using hash function seeding; the second issue is not currently addressed. However, Redis' consideration of malicious inputs does not seem to extend to their PDS implementations.

1.1 Our contributions

Given its prominence in the marketplace and the many other systems that rely on it, we contend that the PDS used in Redis are deserving of detailed analysis. Moreover, in view of the broad set of use cases for these PDS, including those where adversarial interference is anticipated and would be damaging if successful, this analysis should be done in an adversarial setting. This approach follows a line of recent work on PDS analysis [11, 14, 20, 24, 28, 33]. In this paper, we make a comprehensive security analysis of the suite of PDS provided by Redis, with a view to understanding how its constituent PDS perform in adversarial settings. As argued in [6], we regard the observation, documentation, and analysis of such security phenomena "in the wild" as constituting scientific contributions in their own right.

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

Since HLL in Redis was already comprehensively studied in [28], we do not consider it further here. We note only that [28] 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 [16]; it uses a k-means clustering technique [21] to estimate percentiles over a collection of measurements. The structure is an outlier in the Redis PDS suite as it does not work in the streaming setting, but necessitates the batching of data in memory, and it is not really probabilistic in the same sense as the other PDS in Redis. For these reasons, we omit a security evaluation of t-digest (both in general and in the case of the Redis implementation).

This leads us to focus on the remaining four PDS in Redis: Bloom filter, Cuckoo filter, Top-K, and Count-Min sketch. For each PDS, we discuss how the PDS was originally described in the literature and lay out how the Redis implementation differs from this "theoretical" description. We then develop attacks for each of these four PDS, with the attacks in most cases exploiting specific features of the Redis implementations and being more efficient for this reason (simultaneously, we have to deal with the many oddities of the Redis codebase in our attacks). In total, we present 10 different attacks across the four PDS. We compare our attacks with known attacks for these PDS from the literature. We also look at how the PDS in Redis can be protected against attacks, drawing on existing literature that considers this question for PDS more generally [11, 19, 24, 28].

We give a brief flavour of our attacks on the Redis PDS suite. For the Bloom filter implementation, we show how to make any target data item a false positive with few insertions. For the Cuckoo filter, we show how to launch an attack that disables insertions after only a few insertions have been made, far fewer than the filter's expected capacity. For Count-Min sketch, we can inflate the frequency estimate of any target data item to any target level. For Top-K, we can block the PDS from reporting the true K most frequent data items in a stream.

1.2 Responsible disclosure

We notified Redis of our findings on 29.04.2024. This version of our paper 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 analysing Redis' PDS in adversarial settings; naturally we disagree with their viewpoint. However, they also committed to consider changes to their implementation in future versions, including using random seeds instead of fixed seeds, considering alternative hash functions, and adding disclaimers to their documentation. They did not commit to a timeline for this consideration. They decided not to handle our disclosure as a "Redis vulnerability".

1.3 Related work

We are not the first to ask how PDS perform under adversarial inputs. Early work in this direction includes [13, 22]. In particular, [13] addresses hash-table exhaustion, a threat that is actually recognised and addressed by Redis. There is now a rich and growing literature studying the performance of PDS in adversarial settings. Prominent works include [11, 20], which considered the case of Bloom filters, [14, 28, 33] which examined the privacy and security of the HyperLogLog cardinality estimator in adversarial settings (including in [28] an analysis of Redis), and [24] which focussed on attacks on the Count-Min sketch frequency estimator and the HeavyKeeper PDS (for solving the top-k problem). At the same time, researchers have begun to examine the question of how to provably protect PDS against attacks [9, 11, 19, 26].

In comparison to the previous work, our attacks exploit implementation choices of Redis, especially their choice of weak hash functions, to strongly improve the known attacks on PDS. Further, we present new attacks on PDS, such as the insertion failure attack on Cuckoo Filters, using novel techniques to attack these structures. To the best of our knowledge, we are the first to present any attacks on scalable Bloom and Cuckoo Filters.

1.4 Paper organisation

In Section 2 we discuss each of the four PDS from Redis that we study here, both as presented in the literature, and as implemented in Redis. In Section 3 we present our attacks on the Redis implementations of these PDS. Section 4 covers countermeasures, and we conclude in Section 5 with a discussion of our findings and open problems.

2 PDS IN REDIS

We start by introducing the PDS that we consider in this work. For each PDS, we will describe their original specification, the probabilistic guarantees they provide, and give a detailed description of their Redis implementation.

2.1 Bloom filters

A Bloom filter is a commonly-used PDS for answering membership queries in an approximate manner [10], allowing insertions (but not deletions) of elements. It is represented by an m-bit vector σ, along with a collection of k independent hash functions $h_1, ..., h_k$. An element x is inserted into the Bloom filter by setting bits at positions $h_1(x), ..., h_k(x)$ to 1. A membership query on an element x checks if all bits at positions $h_1(x), ..., h_k(x)$ are set to 1.

A *scalable* Bloom filter, proposed in [7], is composed of an assembly of one or more Bloom filters. The Redis PDS module implements such a scalable variant. So we will focus our discussion there. A standard Bloom filter is then a special case of a scalable Bloom filter.

In Redis, a scalable Bloom filter is initialised by the user specifying a desired false positive probability (ε), capacity (c, the number of elements they expect to store), and an expansion factor (ef) by calling BF.setup(ε, c, ef). (The analogous function in Redis is called BF.RESERVE.) We refer to the resulting filter as BF[ε, c, ef]. First, the number of bits per element is derived from this as $bpe \leftarrow -\frac{\log(\varepsilon/2)}{\ln 2}$. The size of the Bloom filter is then calculated as $m \leftarrow \lceil c \cdot bpe/64 \rceil \cdot 64$, and the number of hash functions as $k \leftarrow \lceil \ln 2 \cdot bpe \rceil$. This follows the standard settings in the literature, except that the use of $\varepsilon/2$ in defining bpe implicitly sets the target false positive probability to $\varepsilon/2$ instead of ε. To implement a standard (non-scaling) Bloom filter, the user can set a *no-scaling* flag.

In a Redis Bloom filter, the bit positions to be set for an input x are calculated as follows. First, two values a, b are derived using a hash function h as $a \leftarrow h(x, seed), b \leftarrow h(x, a)$, for a fixed $seed =$ `0xc6a4a7935bd1e995`. The k bit positions corresponding to x are then derived as $p_i \leftarrow (a + (i-1) \cdot b) \bmod 2^{64} \bmod m$ for $i \in [k]$. The first modular reduction here is implicit, and comes from storing the summation result in an integer of type `unsigned long long`. To insert an element, its bit positions are first checked if they are already all set to 1, returning 0 if so; if not, all are set to 1. To query an element, its bit positions are simply checked if all are set to 1, returning 1 if so and 0 if not. Redis sets h to *MurmurHash64A*, a fast (but weak) hash function.

When the Redis Bloom filter reaches its capacity (i.e. when c elements have been inserted), a new subfilter is added. The parameters of the new subfilter are calculated as follows. The false positive probability of the i^{th} subfilter is derived as $\varepsilon_i \leftarrow \varepsilon \cdot 2^{-i}$, and its capacity as $c_i \leftarrow c \cdot (ef)^{i-1}$. Then, m_i, k_i are computed from these in the same manner as before, and the internal representation σ is extended with a new bit vector σ_i of size m_i. Further elements will only be inserted into the new (or some subsequent) subfilter. Redis has no limit on the number of subfilters that can be created.

When multiple subfilters are in use, a membership query on x is made by testing for the presence of x in each subfilter until it is found. Each new subfilter is created with a target false positive probability that half of the previous one and the first has this value set to $\varepsilon/2$, so the false positive probability over the whole assembly is well-approximated by a geometric series converging to the desired value of ε. Of course, this only holds in the honest setting where inserted values are assumed to be independent of any randomness used to construct the filter (i.e. the hash function used). In Redis, there is no randomness (the hash function has a fixed seed), the hash function is weak, and its outputs have additional structure.

In the following, we will refer to the Redis implementations of the insertion and query algorithms on element x as BF.ins(x, σ) and BF.qry(x, σ) respectively, where σ is the assembly of currently instantiated bit vectors. (In Redis, the analogous functions are called BF.ADD and BF.EXISTS, respectively.) We note that Redis does not allow users to view σ (the so-called internal state of the PDS); this observation holds for all Redis PDS. For full details of the Redis implementation of Bloom filters, see Fig. 3 in the Appendix.

2.2 Cuckoo filters

A Cuckoo filter, proposed in [17], supports approximate membership queries. In contrast to a Bloom filter, it allows both insertions and deletions of elements, and offers improved performance. It is represented by a collection σ of n_B buckets, each with s slots, containing compact fingerprints of elements.

In the original specification of the Cuckoo filter [17], an element x is inserted by computing its fingerprint $fp = h_{fp}(x)$, finding its two bucket indices $i = h_1(x) \bmod n_B$, $j = (i \oplus h_2(fp)) \bmod n_B$, and inserting the fingerprint into either bucket. Here, h_{fp}, h_1, h_2 are hash functions. If there is no space in either bucket for x, an eviction process begins: one of x's buckets is chosen at random, along with a random slot. The fingerprint in this slot is moved to its alternative bucket if there is space, and the fingerprint of x is inserted in its place. This eviction process can happen a maximum of *num* times before the insertion is said to fail. Under an "honest" assumption that insertions are independent of the hash functions, [17] showed that Cuckoo Filters can be filled to a very high proportion (eg. load factor of 95% for $s = 4$) before insertions are expected to fail. An element is deleted by removing its fingerprint from one of its buckets, and a membership query simply checks for the presence of x's fingerprint in either of its buckets.

A loose upper-bound on the false positive probability of the Cuckoo filter was derived in [17] as $1-(1-2^{-\lambda_{fp}})^{2s} \leq 2s \cdot 2^{-\lambda_{fp}}$, where λ_{fp} is the size of the fingerprint in bits. This implicitly assumes a worst-case scenario of every bucket being filled with distinct fingerprints. Then, a membership query on x compares the fingerprint of x with $2s$ fingerprints in total (i.e. those stored in the two buckets of x).

The Redis implementation of Cuckoo filters differs from the original description in multiple ways. We will highlight some of the major differences here, but for the full specification of the Redis Cuckoo filter, see Fig. 4 in the Appendix.

First, Redis Cuckoo filters are scalable, similar to the case of Bloom filters. That is, the Cuckoo filter is composed of a sequence of *subfilters*, where the number of buckets in each subfilter is larger than (or equal to) that of the previous subfilter, depending on some specified expansion factor ef.

To initialise a Cuckoo filter in Redis, the user needs to specify a desired capacity, number of slots per bucket (s), maximum number of evictions (num), and the expansion factor (ef) by calling CF.setup(c, s, num, ef). (The analogous function in Redis is called CF.RESERVE.) We will refer to the resulting filter as CF[c, s, num, ef]. To calculate the capacity, the user takes into account the number of elements they wish to insert, the fingerprint length (fixed to $\lambda_{fp} = 8$ bits in Redis) and their desired load factor. The number of buckets in the first subfilter, n_{B_1}, is calculated by dividing the capacity by the number of slots, s. These parameter choices are determined following [17]. Both n_{B_1}, ef are rounded up to the nearest power of two.

An element x is inserted into the Redis Cuckoo filter by calculating its corresponding buckets in the last subfilter using h_1, h_2, and inserting its fingerprint calculated using h_{fp}. However, if there is no space in either bucket, Redis attempts to insert the element into its corresponding buckets in the previous subfilter, and so on, until it reaches the first subfilter. If there is no space in x's buckets in previous subfilters, the eviction process is carried out in the last subfilter. We point out that, in contrast to the original Cuckoo filter, evictions are deterministic (and not randomised) in the Redis implementation. That is, the element in the first slot of the first bucket of x is evicted, followed by the second, and so on (rather than choosing a random slot each time). If, even after evictions, the element cannot be inserted, Redis reverts all evictions and creates a new subfilter to insert x. The number of buckets in the ℓ^{th} subfilter is computed as $n_{B_\ell} \leftarrow n_{B_1} \cdot (ef)^{\ell-1}$. Up to 32 subfilters can be created but only a single element can be inserted into the last one; crucially, after this, all future insertions are disabled.

We will refer to the Redis Cuckoo filter insertion algorithm as $\mathsf{CF.ins}(x, \sigma, \mathit{unique})$, with the parameter $\mathit{unique} \in \{\top, \bot\}$ allowing to switch between Redis' two modes of insertion CF.ADD, CF.ADDNX. In the first mode ($\mathit{unique} = \bot$), x is inserted as described above, returning 1 if successfully inserted and $\bot$ if insertions are disabled. In the second mode ($\mathit{unique} = \top$), x is queried before it is inserted. If x's fingerprint is already present in one of its buckets, 0 is returned; otherwise, insertion proceeds as in the first mode.

Deletion of an element x removes its fingerprint from one of its buckets. The $\mathsf{CF.del}(x, \sigma)$ algorithm searches for x's fingerprint starting in the last subfilter all the way down to the first, returning 1 if successfully deleted and 0 if not found. A membership query on x, $\mathsf{CF.qry}(x, \sigma)$, simply checks for its fingerprint in either of its buckets, but starts from the first subfilter and runs to the last, returning 1 if found and 0 if not. The Redis API further gives users access to the parameters of the Cuckoo filter through $\mathsf{CF.info}(\sigma)$, in particular the current number of subfilters, but in addition n_{B_1}, s, ef, num, and the number of insertions and deletions made so far. (The analogous functions in Redis are called CF.DEL, CF.EXISTS and CF.INFO, respectively.)

The original paper on Cuckoo filters [17] does not specify choices of hash functions. Redis sets $h_{fp}(x) \leftarrow (h_1(x) \bmod 255)+1, h_1(x) \leftarrow$ *MurmurHash64A*(x), and $h_2(x) \leftarrow const \cdot x$, where $const$=`0x5bd1e995` is a constant taken from the 32-bit variant of *MurmurHash*. As we will show, the invertibility of *MurmurHash64A* has have significant consequences for the security of Redis Cuckoo filters.

The false positive probability of the Redis Cuckoo filter is higher than that of the original version, due to the presence of subfilters. With ℓ subfilters, we now compare against $\ell \cdot 2s$ fingerprints instead of $2s$ in the worst case, leading to an upper-bound of $\ell \cdot 2s \cdot 2^{-\lambda_{fp}}$. Note that this bound requires all fingerprints to be equally likely in every bucket. In Redis, fingerprints and buckets are computed from the same hash value but the moduli involved, n_{B_i} and 255 (number of possible fingerprints), are always co-prime because n_{B_i} must be a power of two. This can be used to show that the probability for a fingerprint collision is actually the required value $2^{-\lambda_{fp}}$.

2.3 Count-Min sketches

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 [12], a Count-Min sketch consists of a $k \times m$ array σ of (initially zero) counters, and k pairwise independent hash functions $h_1, ..., h_k$ that map between the universe $\mathcal{U}$ of data items and $[m]$.

An element x is added to a Count-Min sketch by computing $(p_1,..., p_k) \leftarrow (h_1(x),..., h_k(x))$, 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 [12] 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 initialised by the user calling $\mathsf{CMS.setup}(\varepsilon, \delta)$. We will refer to the resulting sketch as $\mathsf{CMS}[\varepsilon, \delta]$. The dimensions m, k of the Count-Min sketch are then calculated as above, and a $k \times m$ array of zeros is initialised. We note that it is also possible to initialise 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. $h_1(x) \leftarrow h(x, 1), ..., h_k(x) \leftarrow h(x, k)$, 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 Fig. 5 in the Appendix.

We point out that using fixed hash functions violates the honest setting assumptions that are required for the guarantees on frequency estimation errors in [12]. We will leverage this and the properties of *MurmurHash2* in our attacks to cause large frequency overestimates.

2.4 Top-K

A Top-K structure, originally introduced as the HeavyKeeper in [37], solves the *approximate* top-K problem.

The exact version of the problem is defined as follows: given elements of a data collection $C \subseteq \{e_1, e_2, ..., e_m\}$ with associated frequencies $(n_{e_1}, n_{e_2}, ..., n_{e_m})$, we can order the elements $\{e_1^*, e_2^*, ..., e_M^*\}$ such that $(n_{e_1}^* \geq n_{e_2}^* \geq ... \geq n_{e_M}^*)$. Then, for some $K \in \mathbb{Z}^+$, we output the set of elements $\{e_1^*, e_2^*, ..., e_K^*\}$ with the K largest frequencies $(n_{e_1}^* \geq n_{e_2}^* \geq ... \geq n_{e_K}^*)$. Given space linear in the stream this is trivial to solve exactly. However, by the pigeonhole principle, it is not possible to find an exact solution with space less than linear (see [34] for a formal impossibility argument). A common technique is to place a small data structure of size $O(K)$, like a heap or list, on top of a compact frequency estimator. By updating this small structure at most once upon an insertion of each element, we can approximate this top-K set [23, 25]. Using this technique we will obtain the K elements with the largest *estimated* frequencies.

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

(1) **if** $fp^* = \star$, 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 $cnt_x < 1 : cnt_x \leftarrow 1$.
(2) **else if** $fp_x = fp^*$, we add 1 to the counter value, and if $cnt_x < c : cnt_x \leftarrow c$, where c is the current counter value at matrix position (i, p_i).
(3) **else** we select a random value $r \leftarrow\!\!\$ \; [0,1)$. If $r < \text{decay}^c$, where c is the current counter value at matrix position (i, p_i), we decrement the counter value stored at this position. If, after decrementing, this value is 0, we then set the counter value to 1, the fingerprint to fp_x, and if $cnt_x < 1 : cnt_x \leftarrow 1$. 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 $cnt_x > H.\mathsf{min}$, 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 $fp_x \leftarrow h_{fp}(x)$, and returning $\hat{n}_x = \max_{i \in [k]}\{\sigma[i][p_i]\}$ where $\sigma[i][p_i].fp = fp_x$. 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 [37], 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 [24] for a meaningful result). Moreover, the results in [37] 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 [24], characterising 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 initialise 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 $\mathsf{TK}[k, m, \text{decay}, K]$. The hash functions for each row are again computed as $h_1(x) \leftarrow h(x, 1), ..., h_k(x) \leftarrow h(x, k)$, with h set to *MurmurHash2* mod m. The fingerprint hash function is computed as $h_{fp} \leftarrow h(x, seed)$, with h_{fp} set to *MurmurHash2* ($\lambda_{fp} = 32$) with a fixed $seed = 1919$. The decay value is by default set to 0.9.

Insertions and frequency queries on an element x then proceed as described above, through the $\mathsf{TK.ins}(x, \sigma)$ and $\mathsf{TK.qry}(x, \sigma)$ functionalities. Similar to the Count-Min sketch, multiple instances of an element can be added to the Top-K, however this is implemented through repeated invocations of the insert algorithm described above. To return the top-K elements, one invokes $\mathsf{TK.list}(\sigma)$. (The analogous functions in Redis are called TOPK.ADD, TOPK.COUNT and TOPK.LIST, respectively.) For full details of the Redis Top-K structure, see Fig. 6 in the Appendix.

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.

3 ATTACKS ON THE REDIS PDS SUITE

In this section, we construct attacks against the Redis implementations of Bloom filters, Cuckoo filters, Count-Min sketches and Top-K structures. 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 [2].

3.1 MurmurHash

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 [1]. However, we adapt this method to make our algorithms randomised 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 [8]. 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.

3.2 Bloom filters

We focus on two attacks: a targeted false positive attack and a pollution attack.

3.2.1 Targeted false positive attack. Let ε, c, ef be Bloom filter parameters. After initialising $\mathrm{BF}[\varepsilon, c, ef]\ \sigma$, the adversary $\mathcal{A}$ has access to insertion and query oracles: $\mathbf{Qry}(\cdot) := \mathsf{BF.qry}(\cdot, \sigma)$ and $\mathbf{Ins}(\cdot) := \mathsf{BF.ins}(\cdot, \sigma)$. In a targeted false positive attack, the adversary receives a target element x as input and wins if x ever becomes a false positive, i.e. $[\mathsf{BF.qry}(\cdot, \sigma) = 1]$ without $\mathbf{Ins}(x)$ being executed.

Let h denote *MurmurHash64A*, and *seed*$\leftarrow$`0xc6a4a7935bd1e995` (as in Redis). For simplicity, we assume that only a single subfilter has been instantiated, but the attack easily extends to the situation where there are multiple subfilters. Let k be the number of bit positions set during insertion and m the size of the Bloom filter bit vector. Set $a \leftarrow h(x, seed)$ and $b \leftarrow h(x, a)$. The bit positions p_i that are set when x is inserted are calculated as $q_i \leftarrow a + (i-1) \cdot b \bmod 2^{64}$ and $p_i \leftarrow q_i \bmod m$ for $i \in [k]$.

Let x_j be ASCII formatted pre-image of q_j under h and the above *seed*. So, $h(x_j, seed) = q_i$. Hence if x_j is inserted, then the value of a that is used equals q_j and so the bit positions set in the Bloom filter are equal to $q_j + (i-1) \cdot b_j \bmod 2^{64} \bmod m$ for $i \in [k]$ and some b_j (which is not material to our attack). Taking $i = 1$, we see that the bit position that is set is equal to $q_j \bmod 2^{64} \bmod m = p_i$.

Therefore, by using k insertion queries $\mathbf{Ins}(x_1), \ldots, \mathbf{Ins}(x_k)$, the adversary can set all the bits p_i for $i \in [k]$, forcing x to be a false positive.

The pre-images x_j are obtained using our *MurmurHash64A* inversion algorithm introduced above. This costs on average 258 hash inversions per target. So, in addition to making k insertion queries, our attack requires about $258k$ hash inversions. We implemented this attack and it worked with the expected complexity.

The cost of our attack can be contrasted with that of the standard targeted false positive attack, cf. [20], in which the attacker chooses random inputs, computes the hash forwards on them, and tries to "cover" all k positions in the Bloom filter for target input x. Modelling this attack as a coupon collector problem, with k coupons being tested for each hash computation, it is easy to show that its expected cost is $\approx 2m(H_k)/k$ forward hash computations. Here the factor of 2 comes from the need to compute *MurmurHash64A* twice to calculate a set of bit positions, while H_k denotes the k-th harmonic number, i.e. $H_k = \sum_{j=1}^{k} 1/j \approx \gamma + \ln(k)$ where $\gamma \approx 0.577$ is the Euler-Mascheroni constant. Comparing the costs, we see that our attack has lower computational cost when $129k^2/H_k < m$. While k is small in practice, the bit-size of the filter m is potentially large, in which case the condition for our attack to have lower cost will be satisfied.

We can further reduce the number of insertions required in our attack above, namely k, by choosing pre-images x_i more carefully. In particular, we can try to select x_i such that inserting it covers not only position p_i but also at least one other bit position for target x. Doing so reduces the number of insertion queries to at most $\lceil k/2 \rceil$ at the cost of increasing the number of hash computations required.

Alternatively, the number of insertion queries needed could be reduced to just 1 (the minimum possible) if we could construct universal (seed-independent) second pre-images for *MurmurHash64A*. Such pre-images for target x would result in identical a and b values, and hence an identical set of positions in the Bloom filter, as for x. As we remarked above, finding such pre-images that are also ASCII encoded seems difficult.

3.2.2 Pollution attack. Let ε, c, ef be Bloom filter parameters. After initialising $\mathrm{BF}[\varepsilon, c, ef]\ \sigma$, an adversary $\mathcal{A}$ is given access to insertion and query oracles: $\mathbf{Qry}(\cdot) := \mathsf{BF.qry}(\cdot, \sigma)$ and $\mathbf{Ins}(\cdot) := \mathsf{BF.ins}(\cdot, \sigma)$. In a pollution attack, the adversary does not receive input and is tasked with setting the false positive probability significantly above the target value of ε. This can be achieved by setting as many bits as possible in the Bloom filter, via careful insertions [20].

The maximal number of bits one can set when the capacity is c insertions is $c \cdot k$. A naïve approach to obtain this optimal result is as follows, cf. [20]. Start with an empty insertion set S. Keep testing distinct (random) inputs and adding to S only the ones mapping to k new positions in the filter, i.e. k positions not hit by elements already in S, until S reaches capacity c. Insert all of S into the Bloom filter. The approach requires c $\mathbf{Ins}$ oracle calls and $2(\sum_{i=0}^{c-1} m^k/[(m-ik)(m-ik-1)\ldots(m-(i+1)k+1)])$ forward hash computations on average for Redis.

Each step in this naïve attack becomes more expensive as S increases in size, as sets of k new bits become harder and harder to collect. However, we can reduce the total number of hash computations by exchanging some number of forward computations for a significantly smaller number of hash inversions, as we explain next.

Let j be any bit position in a Bloom filter of size m. Similarly as for the targeted false positive attack above, we can obtain inputs whose corresponding first bit position in the filter is equal to j, by inverting the hash function at j. Consider the following attack.

Initialise S to be the empty set. Using hash inversion, find t ASCII-encoded inputs having as their first bit position a value that none of the elements already in S covers. Add the pre-image that maps to the maximal number of positions not already covered by elements from S. Repeat until $|S| = c$. With this approach, adding each new element to S costs $258t$ inversions, and t forward hash computations. Each step adds at least one and at most k new covered bits.

We now combine this approach with the naïve one. We start with the naïve approach and switch to the inversion-based one when $|S| = z$, with z minimising the hash computation cost function $g(z) := 2(\sum_{i=0}^{z-1} m^k/[(m-ik)(m-ik-1)...(m-(i+1)k+1)]) + 259t(c-z)$. Our combined approach caps the growing cost of the naïve approach while still achieving a false positive probability significantly higher than ε.

For example, take $\varepsilon \leftarrow 0.02, c \leftarrow 2^{10}, ef \leftarrow 1, t \leftarrow 10$. Then, the first Redis Bloom filter is of size $m = 9856$, each element is associated with $k = 7$ bit positions, and we find $z = 902$. Running the combined approach attack 100 times and averaging, we get that the resulting set S sets 6899 out of 9856 bit positions. This implies an expected false positive probability of $(6899/9856)^7 \approx 0.08$. Our combined approach reduces the number of offline hash computations by 53.2% compared to the naïve approach. On the other hand, the naïve approach sets 7168 bits, yielding a higher expected false positive probability of $(7168/9856)^7 \approx 0.11$.

The presented attack can be optimised further and can be extended to the case of multiple subfilters as in Redis. It is an interesting open problem to find an attack that makes every possible input a false positive in the case of multiple subfilters (this is not possible with a single filter because $ck < m$ always in Redis). This is non-trivial because of how the parameters change across subfilters.

3.3 Cuckoo filters

We give four attacks against Cuckoo filters in Redis. The first attack creates targeted false positives, while the second creates targeted false negatives. The third and fourth attacks focus on causing insertion failures.

3.3.1 Targeted false positive attack. For a Cuckoo filter with parameters c, s, num, ef, we define our attack model as follows. After initialising $\mathrm{CF}[c, s, num, ef]$ σ, an adversary $\mathcal{A}$ is given access to insertion, deletion, query and info oracles: $\mathbf{Ins}(\cdot, unique \in \{\top, \bot\}) := \mathsf{CF.ins}(\cdot, \sigma, unique), \mathbf{Del}(\cdot) := \mathsf{CF.del}(\cdot, \sigma), \mathbf{Qry}(\cdot) := \mathsf{CF.qry}(\cdot, \sigma)$, and $\mathbf{Info} := \mathsf{CF.info}(\sigma)$. In a targeted false positive attack, the adversary receives a target element x as input and wins if x ever becomes a false positive, i.e. $[\mathsf{CF.qry}(x, \sigma) = 1]$ without $\mathbf{Ins}(x, unique)$ being executed for any $unique \in \{\top, \bot\}$.

We use the invertibility of *MurmurHash64A* (h_1 in Redis) to conduct this attack. First, $\mathcal{A}$ calls **Info** to obtain n_B. Then, it computes the first bucket and fingerprint of the target element x by $i \leftarrow h_1(x) \bmod n_B$ and $fp \leftarrow (h_1(x) \bmod 255) + 1$. Its goal is to then find an element $y \neq x$ corresponding to the same bucket and fingerprint. For this, it can use the Chinese Remainder Theorem on the following system of equations: $h' \bmod n_B = i$ and $h' \bmod 255 = fp-1$, to get a set of values for h'. It picks a value that is different from $h_1(x)$, and then inverts this value under h_1. This yields a $y \neq x$ satisfying $h_1(y) \bmod n_B = i$ and $h_1(y) \bmod 255 = fp-1$. $\mathcal{A}$ then calls $\mathbf{Ins}(y, unique)$ for any $unique \in \{\top, \bot\}$, which makes x a false positive. The cost of this attack is dominated by that of inverting *MurmurHash64A*, so on average 258 hash function evaluations.

Note that polluting the filter does not constitute an attack for Cuckoo filters in the same way as it does for Bloom filters. The derivation of the honest setting bound on the false positive probability implicitly assumes the worst possible Cuckoo filter with distinct fingerprints in each slot (see Section 2.2). Thus, one cannot violate this bound when querying a randomly sampled element: even with knowledge of the hash function and unbounded precomputation we cannot create a worse filter.

3.3.2 Targeted false negative attack. We again consider a Cuckoo filter with parameters c, s, num, ef, and define our attack model as follows. After initialising $\mathrm{CF}[c, s, num, ef]$ σ, an honest user makes some number of insertions, deletions and membership queries. Let $\mathcal{X}$ be the set of elements that the honest user inserted, but did not delete. Then, an adversary $\mathcal{A}$ is given access to insertion, deletion, query and info oracles: $\mathbf{Ins}(\cdot, unique \in \{\top, \bot\}) := \mathsf{CF.ins}(\cdot, \sigma, unique), \mathbf{Del}(\cdot) := \mathsf{CF.del}(\cdot, \sigma), \mathbf{Qry}(\cdot) := \mathsf{CF.qry}(\cdot, \sigma)$, and $\mathbf{Info} := \mathsf{CF.info}(\sigma)$. In a targeted false negative attack, the adversary receives a target element $x \in \mathcal{X}$ as input, and wins if x ever becomes a false negative, i.e. $[\mathsf{CF.qry}(x, \sigma) = 0]$ without $\mathbf{Del}(x)$ being executed.

Since an honest user has already made some number of insertions, σ might be composed of multiple subfilters. First, $\mathcal{A}$ calls **Info** to find ef, n_{B_1} (the number of buckets in the first subfilter), and the number of subfilters $subf$. It then computes the number of buckets in the last ($subf^{th}$) subfilter as $n_{B_{subf}} \leftarrow n_{B_1} \cdot (ef)^{subf-1}$. Then, $\mathcal{A}$ proceeds in the same manner as the targeted false positive attack with $n_B \leftarrow n_{B_{subf}}$. This results in an element $y \neq x$ corresponding to the same bucket and fingerprint as x in every subfilter. By calling $\mathbf{Del}(y)$, $\mathcal{A}$ can make x a false negative. The attack cost is the same as the one above.

3.3.3 Insertion failure attack with chosen unique. Consider a Cuckoo filter with parameters c, s, num, ef. After initialising $\mathrm{CF}[c, s, num, ef]$ σ, an honest user makes some number of insertions, deletions and membership queries. Then, an adversary $\mathcal{A}$ is given access to insertion, deletion, query and info oracles: $\mathbf{Ins}(\cdot, unique \in \{\top, \bot\}) := \mathsf{CF.ins}(\cdot, \sigma, unique), \mathbf{Del}(\cdot) := \mathsf{CF.del}(\cdot, \sigma), \mathbf{Qry}(\cdot) := \mathsf{CF.qry}(\cdot, \sigma)$, and $\mathbf{Info} := \mathsf{CF.info}(\sigma)$. In an insertion failure attack, the adversary does not receive input and is challenged with modifying the filter such that all future insertions will fail. The adversary wins if $[\mathsf{CF.ins}(x, \sigma, unique) = \bot]$ for all $x \in \mathcal{U}$ and $unique \in \{\top, \bot\}$.

Since the Redis API allows users to freely choose the parameter *unique*, $\mathcal{A}$ can launch a very simple attack to cause insertion failures with probability 1. We start by assuming the filter is empty. $\mathcal{A}$ first calls **Info** to obtain s. Then, it picks any element x and inserts it $2s + 1$ times with $\mathbf{Ins}(x, \bot)$. The first s insertions fill x's first bucket, and the second s insertions fill its second bucket. The $(2s + 1)^{th}$ insertion leads to an eviction process in this subfilter. However, since evictions will only move elements between these two (already filled) buckets, it will not succeed. We call this an *overflow* of the subfilter; it triggers the creation of a second subfilter and x is inserted into its first bucket in this second subfilter. Then, $\mathcal{A}$ calls $\mathbf{Ins}(x, \bot)$ another $2s + 1s$ times. The first $2s$ insertions fill x's buckets in the second subfilter. The $(2s + 1)^{th}$ insertion leads to the CF.ins algorithm attempting to insert x into its buckets in the previous subfilter, but this will not succeed as they are full. It then attempts evictions in the second subfilter, which will fail as before. Thus, a third subfilter is created to insert x.

$\mathcal{A}$ repeats this process to overflow all 31 subfilters, with a final $\mathbf{Ins}(x, \perp)$ call to create the 32^{nd} subfilter, therefore requiring a total of $31 \cdot 2s+1$ calls to $\mathbf{Ins}(x, \perp)$. After this, all future insertions will fail. This is because once we reach the maximum number of subfilters, CF.ins returns $\perp$ no matter what we insert. Note that, instead of inserting the same element repeatedly, $\mathcal{A}$ can also insert different elements with colliding hash values to achieve the same effect.

We can apply the same strategy to cause an insertion failure after some number of honest insertions. However, in this case, there may already exist multiple prefilled subfilters, and possibly elements in x's buckets. Then, we may require fewer than $2s$ $\mathbf{Ins}(x, \perp)$ calls to fill x's buckets in each subfilter.

3.3.4 Insertion failure attack with unique=$\top$. In light of the simple attack above, a simple countermeasure would be for the Redis API to enforce *unique*=$\top$, rather than allowing the user to choose it. To model this setting, we will consider a similar attack model to Section 3.3.3 but with $\mathcal{A}$'s insertion oracle replaced with $\mathbf{Ins}(\cdot)$:= CF.ins$(\cdot, \sigma, \top)$. We show that, even with this fix, $\mathcal{A}$ can cause insertion failures, though at a higher cost.

With this change, when an element is inserted, it is first queried to check if its fingerprint is already in either of its buckets. The insertion proceeds only if the query returns 0. Consequently, $\mathcal{A}$ can no longer repeatedly insert the same element, or elements with colliding hash values, as their fingerprints will not be added to the filter. Further, since there can only exist one fingerprint connecting any two buckets, we cannot apply the same strategy as in Section 3.3.3; we will need many more elements and many more buckets. So constructing an insertion failure attack in this setting is much more involved. However, we will show that it is still possible to do so efficiently, by finding the smallest set of buckets that we need to fill in order to cause an overflow of a subfilter.

We start by formulating this problem more concretely, assuming an empty subfilter for now. The problem can be stated as follows: find a subset of buckets V' and at least $|V'|s + 1$ elements both of whose buckets are all members of V'. Since each bucket has s slots, at most $|V'|s$ elements can fit into V'. Then any further insertions that require buckets in V' will not fit into this subfilter. This problem can be reformulated in terms of a *Cuckoo Graph.* Various definitions of Cuckoo Graphs exist in the literature [15, 38] for the analysis of Cuckoo Hashing schemes [27]. To analyse Cuckoo filters, we will use the following definition for a Cuckoo Graph: each vertex in the graph corresponds to a bucket, and each edge corresponds to a fingerprint connecting two buckets. We formally define this grpah below.

Definition 1 (Cuckoo Graph). *Consider a Cuckoo filter with n_B buckets, mapping each element $x \in \mathcal{U}$ to fingerprint $fp = h_{fp}(x)$, and buckets $i = h_1(x) \bmod n_B$, $j = (i \oplus h_2(fp)) \bmod n_B$. We define a Cuckoo Graph $G = (V, E)$ associated with the Cuckoo filter with $V := \{1, \dots, n_B\}$ and $E := \{(i, j) : \exists x \in \mathcal{U}, fp = h_{fp}(x), i = h_1(x) \bmod n_B, j = (i \oplus h_2(fp)) \mod n_B\}$.*

To obtain the Cuckoo Graph for the Redis Cuckoo filter, we set $h_{fp}(x) \leftarrow (h_1(x) \bmod 255) + 1$, $h_1(x) \leftarrow MurmurHash64A(x)$, $h_2(x) \leftarrow const \cdot x$ in Def. 1. Note that if the number of buckets is less than or equal to the number of possible fingerprints (i.e. if $n_B \leq 255$), there exists a fingerprint connecting every pair of buckets. In this case, the Cuckoo Graph is a complete graph. However, in practice, $n_B > 255$, and so there may not exist elements mapping to every combination of bucket pairs (or equivalently, edges between every pair of vertices).

Armed with Def. 1, we rewrite the problem of overflowing a subfilter as follows: *given a Cuckoo Graph $G = (V, E)$, find a subgraph $G' = (V' \subseteq V, E' \subseteq E)$ such that $|E'| \geq |V'|s + 1$.* To do this efficiently, V' should be as small as possible, therefore we are interested in finding the smallest subgraph with an edge density strictly greater than s.

Once we have found G', we can convert each bucket pair (i, j) in E' to an element with first bucket i and second bucket j. Redis' choices of h_{fp}, h_1, h_2 allow us to easily find an element x that maps to a particular bucket pair (i, j) in a subfilter with n_B buckets: we first find a fingerprint $fp \leftarrow (j \oplus i) \cdot const^{-1}$, then solve the pair of equations i=$h' \bmod n_B$ and fp−1=$h' \bmod 255$ using the Chinese Remainder Theorem, and finally invert *MurmurHash64A* at h' to recover x. We will call the set of elements corresponding to G' the *overflow set* of the subfilter.

With the above concrete formulation of our problem, we are ready to describe our attack in detail. Recall that $\mathcal{A}$ gets access to a Redis Cuckoo filter after some number of honest insertions, thus there may exist multiple prefilled subfilters, of which the last subfilter is partially prefilled. At the start, $\mathcal{A}$ calls **Info** to get values of n_{B_1}, s, num, ef, as well as the index of the last subfilter. Our attack is then comprised of three main stages for each subfilter: (1) finding an overflow set, (2) sorting the overflow set, and (3) finding a blocking set. We explain each stage in detail below, and give a full pseudocode description in the full version.

Finding an overflow set. To find the subgraph G' (and therefore the overflow set) for a subfilter, we will construct a greedy algorithm. Our algorithm first chooses a random vertex $r \in V$ and adds it to V'. It then goes through all the neighbouring vertices of V' (i.e. all vertices connected by a fingerprint to V') to find the vertex v_{best} with the most connections to V'. It adds v_{best} to V'. Further, all new edges from v_{best} to V' are added to E'. We then repeat the above steps for our updated V', thus growing our subgraph G' until we reach sufficient density (i.e. $|E'| \geq |V'|s + 1$).

We then make a number of pruning steps, to compensate for the random choice of the initial vertex r and to remove any sparsely connected vertices. In each step, we shrink G' by removing the vertex with the least connections to V'. If the density of G' is now lower than required, we add a new vertex by finding the most well-connected neighbour to V', as above. The end result is a subgraph G' with sufficient density, which can be converted to elements that, when inserted, will not fit into the last subfilter.

We mention a few considerations that must be taken into account. Firstly, recall that when elements are inserted with *unique* = $\top$, any element whose fingerprint is already present in any subfilter will not be inserted, since the CF.ins algorithm first queries the element in all subfilters. Therefore, in order for the elements in the overflow set to be successfully inserted and cause an overflow, we must ensure that they do not overlap with the honestly inserted elements, or with elements that we inserted as part of our attack. For this, when choosing the best neighbouring vertex to add to V', we eliminate those corresponding to elements that return 1 when queried, and those that we have already inserted.

Secondly, we will remove the assumption that the subfilter is empty, and consider the case where there exist prefilled elements (eg. if we are overflowing the last, partially prefilled filter). We would like to avoid these elements interfering with our attack, but in reality they might reside in buckets that we wish to overflow. Since we cannot determine exactly which buckets these elements are in, we will simply delete all possible fingerprints in any bucket of V' by calling $\mathbf{Del}(\cdot)$ on appropriate inputs.

At this stage, we have found a set of $(|V'|s{+}1)$ elements such that there exists no allocation of the elements to their buckets in the subfilter. Let x be an element that does not fit into either of its buckets in the subfilter. The CF.ins algorithm tries two more techniques to insert x: first, it attempts to insert x into either of its buckets in previous subfilters, followed by up to *num* evictions in the last subfilter. If either of these operations succeed, x may not go into our desired bucket or cause an overflow. In the following, we demonstrate how to find a subset of n insertions, where $n \leq |V'|s{+}1$, along with an order of inserting them, such that the first $(n{-}1)$ elements will be successfully inserted, and the n^{th} element will fail to be inserted after executing CF.ins. In other words, we will sort the elements of our overflow set such that they will actually cause an overflow.

Sorting the overflow set. For this, we will simulate the insertion of the $(|V'|s{+}1)$ elements, by inserting each element one-by-one into an empty test subfilter, containing the same number of buckets as the actual subfilter. If there is no room in an element's buckets, we carry out maximally *num* evictions. Recall that, unlike in the original Cuckoo filter, evictions in Redis are deterministic. Therefore, we are able to perfectly simulate the eviction steps that would happen upon insertion of an element. We stop inserting elements in our simulation as soon as we find an element that cannot be inserted even after *num* evictions. In the worst case, this will happen at the last, or $(|V'|s{+}1)^{th}$, insertion. Due to our perfect simulation of the eviction process, this gives us the following guarantee: given the allocation of the $(n{-}1)$ elements to buckets in our test subfilter, insertion of the n^{th} element will not succeed after the eviction steps in CF.ins, where $n \leq |V'|s{+}1$. We call the n^{th} element the *overflow element* of this subfilter.

In the third stage, we block the buckets corresponding to the overflow element of a particular subfilter in all previous subfilters, by filling them with dummy fingerprints. We refer to these elements as the *blocking set.* Then, if we insert the n elements in the same order as in our simulation, there will be no space for the overflow element in either of its buckets in the last or previous subfilters, and from the previous stage the eviction process is guaranteed to not succeed. This creates a new subfilter to insert the overflow element.

Finding a blocking set. Let us outline in more detail how to find a set of blocking elements. Consider blocking a bucket i in a subfilter w, where i is not in V' (i.e. i will not be filled due to the overflow set, in which case no blocking would be necessary). We generate a random fingerprint *fp*, calculate the second bucket $j = (i \oplus h_2(fp)) \bmod n_{B_w}$ (where n_{B_w} is the number of buckets in subfilter w), and find the element corresponding to (i, j). Note that a user can only directly insert elements into the last subfilter. Therefore, we must insert this element into subfilter w at the point where it is the last subfilter, i.e. before we insert its overflow element. Then, the blocking element will be inserted into its first bucket i, since it will be empty. We repeat this process to generate and insert s blocking elements to fill bucket i.

However, this procedure only works if subfilter w is empty. Blocking buckets in prefilled subfilters is more challenging, since we cannot directly insert elements into subfilters smaller than the last (partially prefilled) subfilter. Consider blocking a bucket i in a prefilled subfilter w (again where $i \notin V'$). To ensure that our blocking element is actually inserted into subfilter w, both of its buckets must be full in all larger prefilled subfilters, so that the CF.ins algorithm even reaches subfilter w. Now, bucket i must indeed be full, since we must have already blocked it in all later subfilters. However, this may not be true for the second bucket j of our blocking element. In this case, the blocking element might be inserted into bucket j in a later prefilled subfilter, rather than into bucket i in subfilter w. To overcome this, we require additional blocking elements to further block j in all later prefilled subfilters. Nevertheless, this can be optimised by various methods (eg. by choosing a j that will be filled in the process of overflowing the next subfilter, or a j that is strongly connected to the set of full buckets in the next subfilter). After choosing (and possibly blocking) a suitable j, we find the element corresponding to (i, j), and repeat this process s times to block bucket i.

We now have all the ingredients for our attack. We demonstrated how to construct an overflow set, which is a set of elements that will not fit into a subfilter. We determined an order of inserting the overflow set such that inserting the final overflow element will not succeed, even with evictions. And we showed how to find a blocking set, which is a set of elements that fill the buckets of the overflow element in all previous subfilters. In combination, inserting the overflow and blocking sets will lead to the creation of a new subfilter.

Our final step is to insert all required elements for all 31 subfilters. We begin with the last subfilter that currently exists, and insert both its blocking and overflow sets (we exclude its overflow element for now). Next, we insert the blocking sets of all previous prefilled subfilters. (Note that they do not need to be overflowed, as later subfilters already exist.) We then insert the overflow element of the last subfilter, triggering the creation of a new empty subfilter. Finally, we insert the blocking and overflow sets for all empty subfilters up to the 31st subfilter. Note that the insertion order of overflow and blocking sets within a subfilter does not matter, as we carefully avoid placing blocking elements in buckets belonging to an overflow set.

We emphasise that $\mathcal{A}$ can construct overflow and blocking sets for various parameter choices offline, since Redis uses fixed hash functions. Then, in the online phase of the attack, $\mathcal{A}$ simply inserts each element through $\mathbf{Ins}(\cdot)$ calls, overflowing all subfilters and leading to insertion failures.

We present experimental results for the number of insertions required for an insertion failure attack for different parameters of the Cuckoo filter in Fig. 1. The number of insertions strongly

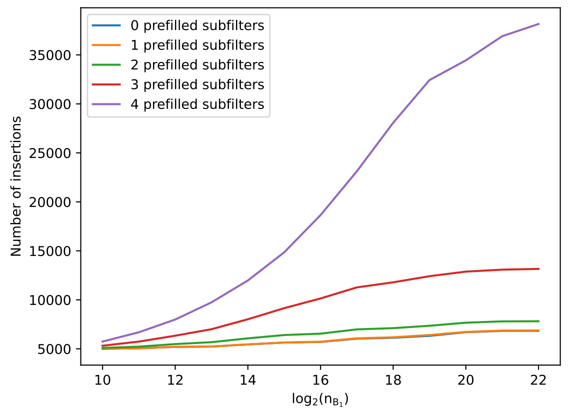

Figure 1: Experimental number of insertions required (averaged over 100 trials) to achieve an insertion failure attack for varying n_{B_1} with $s = 4$, $ef = 1$, $num = 100$.

depends on the number of prefilled filters, due to our expensive blocking strategy, and increases with the number of initial buckets n_{B_1}. We compare our results with the number of insertions required to cause insertion failures in the honest setting, which would require all 31 subfilters to be filled to capacity. From [17], if s=4 then the filter can reach load factors of 95% before insertions are expected to fail. Let n_{B_1}=2^{25}, ef=1 in an initially empty filter. We would then expect to need at least $2^{25}{\cdot}4{\cdot}(0.95){\cdot}31 \approx 2^{31.5}$ insertions to cause an insertion failure in the honest setting, while our attack only requires $8507 \approx 2^{13}$ insertions on average. So, our attack reliably causes insertion failures using a relatively small number of insertions.

3.4 Count-Min sketches

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

3.4.1 Overestimation attack. Consider a Count-Min sketch with parameters ε, δ. After initialising CMS[ε, δ] 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 2.3). 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.

Markelon et al. [24] presented attacks against the general 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' ASCII encoding constraint, we expect this to cost an equivalent of about 16 hash function evaluations for each i (as per Section 3.1). 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 Fig. 7 in the Appendix. We remark that the attack also works against structures that already have elements stored in them.

ϵ, δ (m, k)	Ours	[24]
$2.7 \times 10^{-3}, 1.8 \times 10^{-2}$ $(1024, 4)$	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 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 [24], 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 in [24] with the one we present here. The results are summarised in Table 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 [24]. 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.

3.5 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.

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

To carry out this attack, we adapt the strategy from [24]. 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 $\mathbf{Ins}(\cdot)$ 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 Fig. 8 in the Appendix.

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 initialised a $\mathsf{TK}[4096, 20, 0.9, 20]$, 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 [4]. 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.

3.5.2 Hidden top-K hiding attack. We consider a similar attack model to Section 3.5.1 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 $\mathbf{Ins}(\cdot)$ calls, with t=500 (corresponding to p=2^{-128}, n=10^{11} from the previous method of computing t). In practice, setting t=500 means that with overwhelming probability no true top-K element will ever own its counters for any realistic data collection. Then, for any subsequent items inserted that are not part of the cover, their estimated frequency will be zero. In practice, this blocks any F from any realistic data collection C from being reported in the top-K list. This attack can be seen as a denial-of-service attack, as after the attack phase the structure is prevented from making accurate frequency estimates for any elements that are subsequently inserted into the Top-K. Our full attack is given in Fig. 9 in the Appendix. We verified the correctness of the attack as in Section 3.5.1, except again now setting t=500.

3.5.3 NFC assumption violation attack. Consider a Top-K structure with parameters m, k, decay, K. After initialising $\mathsf{TK}[m, k, \text{decay}, K]$ σ, an adversary $\mathcal{A}$ is given access to insertion and query oracles: $\mathbf{Ins}(\cdot) := \mathsf{TK.ins}(\cdot, \sigma)$ and $\mathbf{Qry}(\cdot) := \mathsf{TK.qry}(\cdot, \sigma)$. The adversary's goal in an NFC assumption violation attack equates to the same goal as of that in Section 3.4.1. 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. To create large error on a given target x, we compute

(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 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.

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 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 3.4.1, obtaining the expected error I on the randomly select target x over all trials. For more details of our attack, see Fig. 10 in the Appendix.

4 COUNTERMEASURES

4.1 Bloom filters

Replacing *MurmurHash64A* with a cryptographic hash function would prevent our attacks based on inversion. However, the generic attacks (based on computing the hash forwards) would remain. The implementation should also be updated to generate the k indices independently of one another, instead of the current "$a+(i-1)\cdot b$" approach. A more comprehensive solution would be to replace the hash function with a keyed PRF. As shown in [19], this leads to Bloom filters that are provably protected against a broad range of attacks, at the cost of roughly doubling m (the filter's size) and introducing the need to manage cryptographic keys. Microbenchmarks in [28] show that a keyed function (PRF) like SipHash is only roughly 2 times slower than MurmurHash3. Therefore, the use of a PRF is feasible, even in a high performance system. Note that the analysis of [19] does not cover the case of an array of subfilters as in Redis; it is an interesting open problem to extend their analysis to this setting.

4.2 Cuckoo filters

Switching from *MurmurHash64A* to a cryptographic hash function for h_1 would make our attacks more difficult, as the pre-image resistance of h_1 prevents us from inverting the function. However, repeated forward computation of h_1 allows us to build a table of bucket pairs and corresponding pre-images, after which our attacks are still possible. We can model the construction of the table as a coupon collector problem for $n_B{\cdot}255$ different combinations of fingerprint and bucket. We expect $n_B{\cdot}255{\cdot}H_{n_B\cdot 255}$ (forward) hash computations in the worst case, where n_B is the number of buckets in a subfilter and H_x refers to the x^{th} harmonic number. This formula corresponds to the computation of a complete table, while for our attack, we only need to find pre-images for specific bucket pairs, reducing the precomputation effort required in practice.

The use of a cryptographic hash function for h_2 complicates our attacks slightly, as we cannot trivially find fp such that $j=(i\oplus h_2(fp)) \bmod n_B$ for given i, j and n_B. However, since $fp\in[1, 255]$ in Redis, finding such fp requires at most 255 trials. Additionally, this countermeasure can greatly complicate the insertion failure attack if the new function distributes its values more uniformly than the h_2 in Redis. A more uniform distribution leads to fewer densely connected subgraphs in the Cuckoo Graph. Therefore, the algorithm for finding an overflow set with the required density will return larger subgraphs. This greatly increases the number of required insertions for the insertion failure attack (Fig. 2).

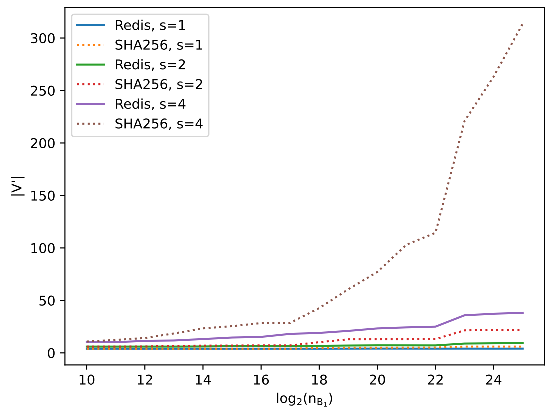

Figure 2: Experimental size of the overflow set $|V'|$ (averaged over 100 trials) required to overflow one subfilter, for $s \in \{1, 2, 4\}$. Solid lines are for Redis' choice of $h_2(\circ) \leftarrow const \cdot \circ$, dotted lines are for $h_2(\circ) \leftarrow \mathsf{SHA256}(\circ)$.

PRF-wrapped Cuckoo filters, introduced in [19], preprocess inputs to all the filter's algorithms with a PRF. This introduces the need to manage keys, but would prevent our attacks: the adversary can no longer compute h_1, and therefore cannot make targeted insertions into the filter. While a proof of correctness under adversarial inputs for PRF-wrapped Cuckoo filters was provided in [19], a security proof for a PRF-wrapped Redis variant is left to future work.

4.3 Count-Min sketches and Top-K

Protecting the Count-Min sketch and Top-K against frequency estimation attacks is more challenging. In [24], some countermeasures are explored, such as switching the hash functions to keyed PRFs, or keeping the structure's internal state private. However, efficient attacks resulting in massive frequency estimation errors were still possible. This implies 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 facilitate these attacks. One could use some public-key infrastructure to only allow insertions from authenticated parties, or explore new ways of constructing frequency estimation PDS, such as the Count-Keeper [24]. While this structure is susceptible to the types of attacks we presented, the attacks become less effective, and it has the ability to flag suspicious frequency estimates.

5 CONCLUSIONS

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

Our work opens up interesting directions for future work. Various other PDS suites exist in the wild, such as in Google BigQuery and Apache Spark, and could also be subjected to detailed security analysis as we have done for Redis here. Methods to provably protect PDS against attacks have been proposed in [11, 19, 26, 28]. However, these analyses tend to focus on textbook versions of the PDS. Adapting these analyses to cater to the specifics of different implementations would help improve confidence in the deployed variants.

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

REFERENCES

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

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

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

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

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

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

[7] Paulo Sergio Almeida, Carlos Baquero, Nuno Preguica, and David Hutchison. Scalable Bloom Filters. *Information Processing Letters*, 101(6):255–261, 2007.

[8] Jean-Philippe Aumasson, Daniel J. Bernstein, and Martin Boßlet. Hash-flooding DoS reloaded: attacks and defenses. https://web.archive.org/web/20130913185247/https://131002.net/siphash/siphashdos_appsec12_slides.pdf.

[9] Omri Ben-Eliezer, Rajesh Jayaram, David P. Woodruff, and Eylon Yogev. A framework for adversarially robust streaming algorithms. In *PODS*, 2020.

[10] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970. https://doi.org/10.1145/362686.362692.

[11] David Clayton, Christopher Patton, and Thomas Shrimpton. Probabilistic data structures in adversarial environments. In *ACM SIGSAC CCS*, 2019.

[12] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[13] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security*, 2003.

[14] Damien Desfontaines, Andreas Lochbihler, and David Basin. Cardinality Estimators do not Preserve Privacy. In *Privacy Enhancing Technologies*, pages 26–46, 2019.

[15] Michael Drmota and Reinhard Kutzelnigg. A precise analysis of cuckoo hashing. *ACM Trans. Algorithms*, 8(2):11:1–11:36, 2012.

[16] Ted Dunning. The t-digest: Efficient estimates of distributions. *Software Impacts*, 7:100049, 2021.

[17] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than Bloom. In *CoNEXT*, 2014.

[18] Tobias Fiebig, Anja Feldmann, and Matthias Petschick. A one-year perspective on exposed in-memory key-value stores. In Nicholas J. Multari, Anoop Singhal, and David O. Manz, editors, *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense, SafeConfig@CCS 2016, Vienna, Austria, October 24, 2016*, pages 17–22. ACM, 2016.

[19] Mia Filić, Kenneth G. Paterson, Anupama Unnikrishnan, and Fernando Virdia. Adversarial correctness and privacy for probabilistic data structures. In *ACM SIGSAC CCS*, 2022.

[20] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. The power of evil choices in bloom filters. In *DSN*, 2015.

[21] Trupti M Kodinariya, Prashant R Makwana, et al. Review on determining number of cluster in k-means clustering. *International Journal*, 1(6):90–95, 2013.

[22] Richard J. Lipton and Jeffrey F. Naughton. Clocked adversaries for hashing. *Algorithmica*, 9(3):239–252, 1993.

[23] Ankush Mandal, He Jiang, Anshumali Shrivastava, and Vivek Sarkar. Topkapi: parallel and fast sketches for finding top-k frequent elements. *NeurIPS*, 2018.

[24] Sam A. Markelon, Mia Filić, and Thomas Shrimpton. Compact frequency estimators in adversarial environments. In *ACM SIGSAC CCS*, 2023.

[25] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems*, 31(3):1095–1133, sep 2006.

[26] Moni Naor and Eylon Yogev. Bloom filters in adversarial environments. In *CRYPTO*, Lecture Notes in Computer Science, 2015.

[27] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[28] Kenneth G. Paterson and Mathilde Raynal. Hyperloglog: Exponentially bad in adversarial settings. In *EuroS&P*, 2022.

[29] Guy Yonish Redis Day TLV 2016. Redis @ Facebook, 2024. https://www.youtube.com/watch?v=XGxntWcjI24.

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

[31] Robert Taylor 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.

[32] Charles Morris RedisDays New York 2022. Using AI to Reveal Trading Signals Buried in Corporate Filings, 2024. https://www.youtube.com/watch?v=

_Lrbesg4DhY.
[33] Pedro Reviriego and Daniel Ting. Security of hyperloglog (HLL) cardinality estimation: Vulnerabilities and protection. *IEEE Commun. Lett.*, 24(5):976–980, 2020.
[34] Tim Roughgarden and Gregory Valiant. Cs168: The modern algorithmic toolbox lecture #2: Approximate heavy hitters and the count-min sketch. page 15.
[35] Salvatore Sanfilippo. A few things about redis security. http://antirez.com/news/96.
[36] Kurt John The Data Economy Podcast. Using Real-Time Data and Digital Twins to Improve Cyber Security, 2024. hhttps://www.youtube.com/watch?v=TycylT0J6cc.
[37] Tong Yang, Haowei Zhang, Jinyang Li, Junzhi Gong, Steve Uhlig, Shigang Chen, and Xiaoming Li. Heavykeeper: an accurate algorithm for finding top-k elephant flows. *IEEE/ACM Transactions on Networking*, 27(5):1845–1858, 2019.
[38] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In *CRYPTO*, 2023.

A PDS IN REDIS

We give more details on the specifications of PDS in Redis. The algorithms of Redis Bloom filters are given in Fig. 3, Count-Min sketches in Fig. 5, and Top-K structures in Fig. 6.

B ATTACKS ON THE REDIS PDS SUITE

We present the pseudocode descriptions of some of our attacks. The overestimation attack against the Redis Count-Min sketch is given in Fig. 7. For attacks against the Redis Top-K structure, the known top-K hiding attack is given in Fig. 8, the hidden top-K hiding attack in Fig. 9, and the NFC assumption violation attack in Fig. 10.

```
BF.setup($pp$)
$\varepsilon, c_1, ef \leftarrow pp$
$seed \leftarrow$ 0xc6a4a7935bd1e995
$subf \leftarrow 1$ // no. of subfilters
$n_1 \leftarrow 0$ // no. of elements in subfilter
$\varepsilon_1 \leftarrow \varepsilon \cdot 0.5$
$bpe_1 \leftarrow -\frac{\log_2 \varepsilon_1}{\ln 2}$ // bits per element
$m_1 \leftarrow \lceil c_1 \cdot bpe_1/64 \rceil \cdot 64$ // size in bits
$k_1 \leftarrow \lceil \ln 2 \cdot bpe_1 \rceil$ // number of hashes
$h(\circ) \leftarrow MurmurHash64A(\circ)$
$\sigma_1 \leftarrow 0^{m_1}$
return $\top$
```

```
BF.qry($x, \sigma$)
$a \leftarrow h(x, seed)$
$b \leftarrow h(x, a)$
for $\ell \in [subf, \ldots, 1]$
    $unset \leftarrow 0$
    for $i \in [k_\ell]$
        $p_i \leftarrow (a + (i-1) \cdot b) \bmod 2^{64} \bmod m_\ell$
        if $\sigma_\ell[p_i] = 0$
            $unset \leftarrow 1$
    if $unset = 0$
        return 1
return 0
```

```
BF.ins($x, \sigma$)
if $1 \leftarrow$ BF.qry($x, \sigma$)
    return 0
$a \leftarrow h(x, seed)$
$b \leftarrow h(x, a)$
if $n_{subf} \geq c_{subf}$
    $subf \leftarrow subf + 1$
    $\varepsilon_{subf} \leftarrow \varepsilon_{subf-1} \cdot 0.5$
    $bpe_{subf} \leftarrow -\frac{\log_2 \varepsilon_{subf}}{\ln 2}$
    $c_{subf} \leftarrow c_{subf-1} \cdot ef$
    $m_{subf} \leftarrow \lceil c_{subf} \cdot bpe_{subf}/64 \rceil \cdot 64$
    $k_{subf} \leftarrow \lceil \ln 2 \cdot bpe_{subf} \rceil$
    $\sigma_{subf} \leftarrow 0^{m_{subf}}$
for $i \in [k_{subf}]$
    $p_i \leftarrow (a + (i-1) \cdot b) \bmod 2^{64} \bmod m_{subf}$
    $\sigma_{subf}[p_i] \leftarrow 1$
$n_{subf} \leftarrow n_{subf} + 1$
return 1
```

Figure 3: Redis Bloom filter algorithms. The analogous functions in the Redis API are: BF.setup is BF.RESERVE, BF.qry is BF.EXISTS, and BF.ins is BF.ADD. We refer to a Redis Bloom filter initialised with *pp*=ε_1, c_1, ef as BF[ε_1, c_1, ef].

```
CF.setup(pp)
$c, s, num, ef \leftarrow pp$
$f \leftarrow 256$ // fingerprint length 8 bits
$const \leftarrow$ 0x5bd1e995 // MurmurHash constant
$fmax \leftarrow 32$ // max. no. of subfilters
$subf \leftarrow 1$ // no. of subfilters
$n_{B_1} \leftarrow \text{next\_power\_of\_two}(c/s)$ // no. of buckets
$ef \leftarrow \text{next\_power\_of\_two}(ef)$ // expansion, rounded
$h_1(\circ) \leftarrow MurmurHash64A(\circ)$
$h_2(\circ) \leftarrow const \cdot \circ$
$h_{fp}(\circ) \leftarrow (h_1(\circ) \bmod f-1) + 1$
for $i \in [0, n_{B_1}-1]$
    $\sigma_{(1,i)} \leftarrow \perp^s$
return $\top$
```

```
CF.del(x, σ)
$fp \leftarrow h_{fp}(x)$
for $\ell \in [subf, ..., 1]$
    $i \leftarrow h_1(x) \bmod n_{B_\ell}$
    $j \leftarrow (i \oplus h_2(fp)) \bmod n_{B_\ell}$
    for $slot \in [0, s-1]$
        if $\sigma_{(\ell,i)}[slot] = fp$
            $\sigma_{(\ell,i)}[slot] \leftarrow \perp$
            return 1
    for $slot \in [0, s-1]$
        if $\sigma_{(\ell,j)}[slot] = fp$
            $\sigma_{(\ell,j)}[slot] \leftarrow \perp$
            return 1
return 0
```

```
CF.qry(x, σ)
$fp \leftarrow h_{fp}(x)$
for $\ell \in [1, ..., subf]$
    $i \leftarrow h_1(x) \bmod n_{B_\ell}$
    $j \leftarrow (i \oplus h_2(fp)) \bmod n_{B_\ell}$
    for $slot \in [0, s-1]$
        if $\sigma_{(\ell,i)}[slot] = fp$
            return 1
    for $slot \in [0, s-1]$
        if $\sigma_{(\ell,j)}[slot] = fp$
            return 1
return 0
```

```
CF.info(σ)
return $n_{B_1}, subf, s, ef, num$
```

```
CF.ins(x, σ, unique)
if $subf = fmax$ :
    return $\perp$
if $unique$
    if $1 \leftarrow$ CF.qry$(x, \sigma)$
        return 0
$fp \leftarrow h_{fp}(x)$
for $\ell \in [subf, ..., 1]$
    $i \leftarrow h_1(x) \bmod n_{B_\ell}$
    $j \leftarrow (i \oplus h_2(fp)) \bmod n_{B_\ell}$
    for $slot \in [0, s-1]$
        if $\sigma_{(\ell,i)}[slot] = \perp$
            $\sigma_{(\ell,i)}[slot] \leftarrow fp$
            return 1
    for $slot \in [0, s-1]$
        if $\sigma_{(\ell,j)}[slot] = \perp$
            $\sigma_{(\ell,j)}[slot] \leftarrow fp$
            return 1
// swap in last subfilter
$z \leftarrow h_1(x) \bmod n_{B_{subf}}$
$slot \leftarrow 0$
for $g \in [1, num]$
    $fp' \leftarrow \sigma_{(subf,z)}[slot]$
    $\sigma_{(subf,z)}[slot] \leftarrow fp$
    $fp \leftarrow fp'$
    $z \leftarrow (z \oplus h_2(fp)) \bmod n_{B_{subf}}$
    for $slot' \in [0, s-1]$
        if $\sigma_{(subf,z)}[slot'] = \perp$
            $\sigma_{(subf,z)}[slot'] \leftarrow fp$
            return 1
    $slot \leftarrow (slot + 1) \bmod s$
for $g \in [1, num]$ // revert evictions
    $slot \leftarrow (slot + s - 1) \bmod s$
    $z \leftarrow (z \oplus h_2(fp)) \bmod n_{B_{subf}}$
    $fp' \leftarrow \sigma_{(subf,z)}[slot]$
    $\sigma_{(subf,z)}[slot] \leftarrow fp$
    $fp \leftarrow fp'$
$subf \leftarrow subf + 1$
$n_{B_{subf}} \leftarrow n_{B_1} \cdot ef^{subf-1}$
for $i \in [0, n_{B_{subf}}-1]$
    $\sigma_{(subf,i)} \leftarrow \perp^s$ // new empty subfilter
$i \leftarrow h_1(x) \bmod n_{B_{subf}}$
$\sigma_{(subf,i)}[0] \leftarrow fp$
return 1
```

Figure 4: Redis Cuckoo filter algorithms. The analogous functions in the Redis API are: CF.setup is CF.RESERVE, CF.del is CF.DEL, CF.qry is CF.EXISTS, CF.info is CF.INFO, and CF.ins combines both CF.ADD and CF.ADDNX. We refer to a Redis Cuckoo filter initialised with pp=c, s, num, ef as CF$[c, s, num, ef]$. For $a \in \mathbb{R}$, we use next_power_of_two(a) to denote a function that rounds up a to the next power of two.

```
CMS.setup($pp$)
    $\varepsilon, \delta \leftarrow pp$
    $m \leftarrow \left\lceil \frac{e}{\varepsilon} \right\rceil$
    $k \leftarrow \left\lceil \ln(\frac{1}{\delta}) \right\rceil$
    $h(\circ) \leftarrow \mathit{MurmurHash2}(\circ) \bmod m$
    $\sigma \leftarrow \text{zeros}(k, m)$
    return $\top$

CMS.ins($x, \sigma, v$)
    $(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
    for $i \in [k]$
        $\sigma[i][p_i] += v$
    return $\min_{i \in [k]} \{\sigma[i][p_i]\}$

CMS.qry($x, \sigma$)
    $(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
    return $\min_{i \in [k]} \{\sigma[i][p_i]\}$
```

Figure 5: 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 initialised with $pp = \varepsilon, \delta$ as CMS$[\varepsilon, \delta]$.

```
TK.setup($pp$)
    $m, k, \text{decay}, K \leftarrow pp$
    $\mathit{seed} \leftarrow 1919$
    $h(\circ) \leftarrow \mathit{MurmurHash2}(\circ) \bmod m$
    $h_{fp} \leftarrow \mathit{MurmurHash2}(\circ)$
    for $i \in [k]$
        $\sigma[i] \leftarrow [(\star, 0)] \times m$
    $H \leftarrow \text{initminheap}(K)$
    return $\top$

TK.qry($x, \sigma$)
    $(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
    $fp_x \leftarrow h_{fp}(x, \mathit{seed})$
    $cnt_x \leftarrow 0$
    for $i \in [k]$
        if $\sigma[i][p_i].fp = fp_x$
            $cnt \leftarrow \sigma[i][p_i].cnt$
            $cnt_x \leftarrow \max\{cnt_x, cnt\}$
    return $cnt_x$

TK.list($\sigma$)
    $T \leftarrow H.\text{list}()$
    return $T$

TK.ins($x, \sigma$)
    $r \leftarrow \text{nil}$
    $(p_1, \ldots, p_k) \leftarrow h(x, 1), \ldots, h(x, k)$
    $fp_x \leftarrow h_{fp}(x, \mathit{seed})$
    $cnt_x \leftarrow 0$
    for $i \in [k]$
        if $\sigma[i][p_i].fp \notin \{fp_x, \star\}$
            $r \leftarrow\!\$\ [0, 1)$
            if $r \leq \text{decay}^{\sigma[i][p_i].cnt}$
                $\sigma[i][p_i].cnt -= 1$
        if $\sigma[i][p_i].cnt = 0$
            $\sigma[i][p_i].fp \leftarrow fp_x$
        if $\sigma[i][p_i].fp = fp_x$
            $\sigma[i][p_i].cnt += 1$
            if $\sigma[i][p_i].cnt > cnt_x$
                $cnt_x \leftarrow \sigma[i][p_i].cnt$
    if $cnt_x \in H$
        $H.\text{update}(x, cnt_x)$
    elseif $cnt_x > H.\text{getmin}()$
        $r \leftarrow H.\text{getmin}()$
        $H.\text{poppush}(x, cnt_x)$
    return $r$
```

Figure 6: Redis Top-K structure algorithms. The analogous functions in the Redis API are: TK.setup is TOPK.RESERVE, TK.ins is TOPK.ADD, TK.qry is TOPK.COUNT, and TK.list is TOPK.LIST. We refer to a Redis Top-K structure initialised with $pp = m, k, \text{decay}, K$ as TK$[m, k, \text{decay}, K]$.

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

Figure 7: 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.

```
known_F_attack^Ins(F, n, p, pp)
t ← get_t(n, p, pp)
F_cover ← find_F_cover(F, pp)
for e ∈ F_cover
    for $i \in [t]$
        Ins(e)
return done

get_t(n, p, pp)
m, k, decay, K ← pp
$g(t) \leftarrow \log_2(k \cdot n^t \cdot \text{decay}^{t(t+1)/2}) - \log_2(p)$
$t_1, t_2 \leftarrow$ 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

find_F_cover(F, pp)
m, k, decay, K ← pp
F_cover ← ∅
for $f \in F$
    $(p_1, \ldots, p_k) \leftarrow h(f, 1), \ldots, h(f, k)$
    for $i \in [k]$
        $y \leftarrow$ MurmurHash2Inverse$(p_i, i)$
        F_cover ← F_cover ∪ {y}
return F_cover
```

Figure 8: The Top-K known top-*K* hiding attack.

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

find_S_cover(pp)
m, k, decay, K ← pp
$\eta \leftarrow$ zeros(k, m)
S_cover ← ∅
for $i \in [k]$
    for $j \in [m]$
        if $\eta[i][j] = 0$
            $y \leftarrow$ MurmurHash2Inverse$(j, i)$
            S_cover ← S_cover ∪ {y}
            $(p_1, \ldots, p_k) \leftarrow h(y, 1), \ldots, h(y, k)$
            for $r \in [k]$
                $\eta[r][p_r] \leftarrow 1$
return S_cover
```

Figure 9: The Top-K hidden top-*K* attack.

```
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 ← ⊥
$P \leftarrow (h(x, 1), \ldots, h(x, k))$
$fp_x \leftarrow h_{fp}(x)$
while done = ⊥
    $y \leftarrow$ MurmurHash2Inverse$(fp_x, seed)$
    $C \leftarrow (h(y, 1), \ldots, h(y, k))$
    for $i \in [k]$
        if $P[i] = C[i]$
            done ← ⊤
return y
```

Figure 10: 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.