HyperLogLog: Exponentially Bad in Adversarial Settings

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

Mathilde Raynal
EPFL
mathilde.raynal@epfl.ch

September 7, 2021

Abstract

Computing the count of distinct elements in large data sets is a common task but naive approaches are memory-expensive. The HyperLogLog (HLL) algorithm (Flajolet *et al.*, 2007) estimates a data set's cardinality while using significantly less memory than a naive approach, at the cost of some accuracy. This trade-off makes the HLL algorithm very attractive for a wide range of applications such as database management and network monitoring, where an exact count may not be needed. The HLL algorithm and variants of it are implemented in systems such as Redis and Google Big Query. Recently, the HLL algorithm has started to be proposed for use in scenarios where the inputs may be adversarially generated, for example counting social network users or detection of network scanning attacks. This prompts an examination of the performance of the HLL algorithm in the face of adversarial inputs. We show that in such a setting, the HLL algorithm's estimate of cardinality can be exponentially bad: when an adversary has access to the internals of the HLL algorithm and has some flexibility in choosing what inputs will be recorded, it can manipulate the cardinality estimate to be exponentially smaller than the true cardinality. We study both the original HLL algorithm and a more modern version of it (Ertl, 2017) that is used in Redis. We present experimental results confirming our theoretical analysis. Finally, we consider attack prevention: we show how to modify HLL in a simple way that provably prevents cardinality estimate manipulation attacks.

1 Introduction

Data scientists today make use of systems like Google Big Query, Apache Spark, Presto and Redis to handle large data sets. Such systems include analytics engines capable of performing powerful operations on the data. Particularly useful to data scientists are summary statistics, with a key statistic being the number of distinct data items in a data set, i.e. the data set's cardinality. In many use cases, obtaining an approximate cardinality is sufficient. Over the last decade, the HyperLogLog (HLL) cardinality estimator [1] has emerged as the algorithm of choice for this task, due to its excellent speed, low memory requirements and high proven accuracy.[1] It is also suitable for use in applications where it is not possible (or desirable) to store an entire data set, but instead where the data set is presented in a streaming fashion. HLL is used by all of the above-mentioned systems.

However, HLL is increasingly used in settings where adversaries may have incentives to manipulate the cardinality estimates made by the algorithm, i.e. HLL is being used in *adversarial settings*. For example:

- Facebook uses HLL to find out how many distinct people visited their website in the past week [2].
- Google uses a variant of HLL described in [3] in its PowerDrill system (first described in [4]) and possibly in a variety of other internal systems reported in [3]. For example, cardinality

[1]Section 3 contains a complete specification of HLL.

estimation is used to to determine the number of distinct search queries on `google.com` over a time period [3].

- HLL has been implemented in network soft-switches to approximate the number of distinct packets in traffic flows and overcome the switch memory limits [5]. The number of unique flows traversing a switch is then used to inform a congestion control mechanism.
- HLL has been proposed as a solution to detecting Denial-of-Service attacks [6] (this paper actually uses a variation on HLL called *sliding HLL* [7] which estimates the cardinality over a moving time window). Specifically, HLL is used to count how many different destination ports are seen over all packets received, in an attempt to identify when a port scan attack is underway.

None of the above-cited examples discuss any kind of threat model where an adversary may try to manipulate the HLL cardinality estimate to its advantage. This is despite them all operating in potentially adversarial settings. We infer that the developers of these systems may not be fully aware of the risks they are taking in relying on HLL.

Recently, Reviriego and Ting [8] initiated work on the performance of HLL in adversarial settings. They showed that in very specific attack settings, the cardinality estimate made by HLL can be modified through input selection. In this paper, we present a complete analysis of HLL in adversarial settings, considering more realistic attack scenarios and giving more powerful attacks than [8]. In particular, we show that with only modest knowledge of the HLL internals and a moderate amount of computation, an adversary with sufficient flexibility in the choice of inputs can make the HLL cardinality estimate exponentially smaller than the true cardinality (e.g. constant instead of $\mathcal{O}(2^t)$ where t is the number of bits of flexibility in the adversary's input). We demonstrate this theoretically and experimentally for the "classic" version of HLL from [1]. We also study a specific "modern" version of HLL which adopts recommendations from [9] and show how variants of our attacks apply there too. This modern version of HLL is interesting because it is the one used in Redis.

We stress that none of our attacks are particularly deep in cryptanalytic terms. Yet HLL is being increasingly widely deployed, and never with protections against our attacks (insofar as we are aware). So despite their technical simplicity, we argue that our attacks are important in establishing the limitations of HLL in the broadest set of use cases. We present our attacks in an abstract way rather than tying them to any specific setting in order to illustrate their generality. We do give a more detailed exploration of their impact on the data-centre network monitoring system proposed in [5], see Section 7.

To complement our attack results, we provide a formal security analysis of HLL. We show that in the "shadow device" attack scenario of [8], no security is possible even if HLL is secretly keyed. On the other hand, we also show that in the most powerful attack setting introduced here, where the attacker has full access to the internals of the HLL algorithm (except for a secret key!), it is possible to provably secure HLL against cardinality manipulation by the simple expedient of replacing its internal hash function h by a keyed, variable-input-length pseudo-random function (VIL-PRF) F. We discuss and benchmark a suitable choice for the required VIL-PRF. To support our formal analysis, we introduce a novel simulation-based security model for HLL. We believe our simulation-based approach will be of independent interest and will be useful beyond the analysis of cardinality manipulation attacks, for example when studying privacy properties of HLL and other probabilistic data structures.

1.1 Paper Organisation

The rest of the paper is organised as follows. Section 2 immediately below discusses related work. Section 3 gives an overview of HLL. Section 4 presents our different adversarial models. In Section 5, we describe attacks on HLL and evaluate their impacts under the different adversarial models. Section 6 considers the security of the more modern version of HLL due to Ertl [9] that is used in Redis. Our experimental results are discussed and related back to a specific application setting in Section 7. Section 8 provides our formal security analysis of HLL. Section 9 provides our conclusions and ideas for future work.

2 Related Work

Desfontaines *et al.* [10] studied the privacy properties of HLL in two different attack scenarios, referred to as *insider* and *external*. The attack target of [10] is different from ours, being focused on privacy breaches. We discuss the two attack scenarios from [10] in more detail in Section 4.

In [8], Reviriego and Ting exploit a vulnerability of HLL to manipulate the cardinality estimate, leading to a five-fold reduction in the estimate compared to the true cardinality. We consider the attack model used in [8] to be quite artificial, since it gives the adversary the ability to test whether inserting an item *would* increase the cardinality estimate, without actually having to insert the item. Such a setup might be possible under certain circumstances, for example where the adversary has access to an identical "shadow" device; see further discussion in Section 4. For completeness, we include analyses of this attack scenario, showing that we can drastically improve on the attacks in [8] (e.g. we can reduce the HLL cardinality estimate by a factor of almost 1,000 instead of the factor of 5 achieved in [8]). However, we stress that we consider this scenario to be the least realistic (and therefore the least interesting).

Both [10, 8] propose mitigations using a salted HLL sketch, either as a replacement for (but at the cost of losing mergeability) or in addition to (inducing a memory overhead) an unsalted HLL. We show in Section 8 that when a salt is used appropriately (i.e. as the secret key for a pseudo-random function that is used in place of hash function h), security against adversarial inputs can be achieved even in our strongest adversarial setting.

Our work relates to the broader study of the performance of probabilistic data structures in adversarial environments. Clayton *et al.* [11] recently provided a provable-security treatment of this topic. They used a game-based formalism to analyze Bloom filters, counting (Bloom) filters and count-min sketch data structures. They did not study HLL. Our use of a simulation-based security definition in place of the game-based ones used in [11] is another point of distinction; see further discussion in Section 8.

Prior work also focused on Bloom filters [12, 13] or flooding of hash tables [14, 15]. References [13, 11] provide comprehensive summaries of prior work in this area.

3 Overview of HLL

HLL [1] is based on the key observation that, for a stream of items represented as bit-strings of some fixed length, if all items have at most k leading zero-bits (and one or more items actually have this many leading zero-bits), then the cardinality of the stream (i.e. the number of distinct values it contains) is likely to be on the order of 2^{k+1}. To ensure a uniform distribution on the items' representations as bit-strings, the items are processed by a hash function h before the leading bits are inspected.

So to roughly estimate the cardinality of a stream, we need only to store a representation of k, the length of the largest observed string of leading zero-bits for hashes of items in the stream. This provides a very compact mechanism for estimating a stream's cardinality – if the true cardinality is N, then just $\mathcal{O}(\log\log(N))$ bits of storage are needed to compute the cardinality estimate.

This simple approach suffers from large variance. In order to reduce the variance, HLL uses many estimators in parallel instead of one and averages the results. Each estimate is stored in its own register that we call a bucket. We refer to the collection of buckets and their contents as the HLL *sketch*. To map a value x to one of $m = 2^n$ buckets, [16] suggests using the first n bits of $h(x)$ as a bucket index, and then computing the longest sequence of leading zero bits on the remaining ℓ bits of $h(x)$. The output length of $h(\cdot)$ is thus $n + \ell$. This sum is fixed to 32 in the original HLL paper [1]. Estimates from all buckets are averaged using the harmonic mean and scaled by

```
Let $h : \hat{\mathcal{D}} \to \{0,1\}^{32}$ hash data from some domain $\hat{\mathcal{D}}$ to binary 32-bit strings.
For $s$ of bit-length $\ell$, let $\rho(s)$ denote the position of the leftmost 1-bit of $s$: e.g.,
$\rho(1...) = 1, \rho(0001...) = 4, \rho(0^\ell) = \ell + 1$.
define $m = 2^n$ with $n \in [4..16]$.
define $\alpha_{16} = 0.673$; $\alpha_{32} = 0.697$; $\alpha_{64} = 0.709$; $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m \geq 128$;

Algorithm HyperLogLog (input $\mathcal{X}$ : multiset of items from domain $\hat{\mathcal{D}}$).
initialize a collection of $m$ registers, $M[1], ..., M[m]$, to 0;

for $x \in \mathcal{X}$ do
    set $y := h(x)$;
    set $j = 1 + \langle y_1 y_2 ... y_n \rangle_2$;          {bucket index determined by the first $n$ bits of $y$}
    set $w := y_{n+1} y_{n+2} \cdots y_{n+\ell}$;          {bit string determined by the last $\ell$ bits of $y$}
    set $M[j] := \max(M[j], \rho(w))$;
compute $E := \alpha_m m^2 \cdot (\sum_{j=1}^{m} 2^{-M[j]})^{-1}$;          {the raw HyperLogLog estimate}
if $E \leq \frac{5}{2} m$ then
    let $V$ be the number of registers equal to 0;
    if $V \neq 0$ then set $E^* := m \cdot \ln(m/V)$ else set $E^* := E$;          {small range correction}
if $E \leq \frac{1}{30} 2^{32}$ then
    set $E^* := E$;          {intermediate range — no correction}
if $E > \frac{1}{30} 2^{32}$ then
    set $E^* := -2^{32} \ln(1 - E/2^{32})$;          {large range correction}
return cardinality estimate $E^*$ with typical relative error $\pm 1.04/\sqrt{m}$.
```

Figure 1: The HyperLogLog Algorithm from [1]

a constant α_m, where α_m is empirically computed to correct a systematic multiplicative bias. See Figure 1.

In the original HLL proposal [1], relatively small and large cardinalities are handled separately to correct for systematic errors. When the raw cardinality estimate is below $\frac{5}{2}m$, the Hit Counting algorithm of Whang *et al.* [17] is used to produce the final cardinality estimate. This estimator uses the number of empty buckets (whose register value is 0) instead of the number of leading zeros in the stream. A similar correction is applied for large cardinalities in order to take into account the likelihood of hash collisions. In addition, the cardinality estimate is usually rounded to a whole number before being output. The original HLL algorithm from [1] is shown in Figure 1. This algorithm is able to estimate cardinalities greater than 10^9 with a typical standard error of 4%, using only 1.5 kB of memory. Crucially, the proof of this [1, Theorem 1] makes the assumption that the inputs are drawn independently and uniformly at random from the input domain – loosely put, it is an average case analysis.[2]

Some works building on [1] replace the Hit Counting algorithm by other schemes and use a hash function with a larger output domain to reduce the chances of collision and thus get rid of the need for large cardinality range correction. For example, Google engineers proposed using a 64-bit hash function and Linear Counting along with bias correction in the small cardinality range [3],[3] while Redis has adopted a new estimator based on a more precise statistical analysis of the counting problem provided by [9].

In this paper we mostly focus on the HLL formulation in the original paper [1] as represented in Fig. 1 and as widely used in practice. We also study [9] because of its recent deployment in Redis. We leave for future work the analysis of further HLL variants.

An interesting property of HLL is that it supports merging in a lossless way. To combine two

[2]Technically, the results of [1] are proven for a distribution obtained by making arbitrary replications and permutations after sampling independently and uniformly, to simulate input taken from an *ideal multiset*. But they also hold without replication and permutation because HLL's internal state is invariant under repeated inputs.

[3]See also the Google blogpost describing the implementation in Google Big Query at `https://cloud.google.com/blog/products/gcp/counting-uniques-faster-in-bigquery-with-hyperloglog`.

buckets at index i in two different HLL instances, one can take the maximum of the two bucket entries and assign that to the matching bucket i in the merged HLL sketch. Such a simple set union operation allows easy parallelisation of operations among multiple machines independently, provided they use the same hash function and the same number of buckets. Clearly this procedure cannot be carried out if the different HLL sketches are differently seeded or keyed.

4 HLL in Adversarial Settings

This section presents the different adversarial settings that we consider. In all settings, our adversary's aim is to manipulate the HLL cardinality estimate. We focus on reducing the estimate as much as possible, since this would cause more damage when considering the previously mentioned applications of HLL. It is easy to modify our attack strategies for an adversary who instead wishes to artificially inflate the HLL cardinality estimate given a limited input capability.

4.1 Attacker's Objective, Abstractly

Throughout, we assume the adversary has the ability to vary the contents of free fields in the strings which will be inserted into the stream of items. For example it might manipulate IPID and other header fields in the context of IP packets. We model this in a general way by assuming the adversary has access to a large set $\mathcal{X}$ of potential inputs from which it can select items to insert into the HLL sketch. Our adversary's objective, then, is to insert as many items from $\mathcal{X}$ as possible whilst keeping the HLL cardinality estimate as low as possible.

4.2 Adversarial Scenarios

We model four distinct adversarial scenarios based on the adversary's knowledge and capabilities. To simulate real-life settings, we consider both cases when the sketch is shared and receives inputs from other honest users but also when inputs are under the sole control of the adversary. It is reasonable to assume that the adversary knows which case it is in. These two settings differ mainly in the fact that the adversary is attacking a sketch that may already contain other users' data in the former option, but is empty in the latter.

Among the scenarios, the *insider* setup of [10] that we present as S4 makes the strongest assumptions about the adversarial capabilities since it assumes that the adversary has a perfect view of the sketch (at some point in time).

S1: The adversary does not know the details of the target HLL sketch but can access a shadow copy of the HLL sketch via its API. It can use the API to insert elements into this shadow HLL sketch, get its cardinality estimate, and reset it to its native empty state. The adversary does not know the state of the targeted HLL sketch, meaning in particular that it does not have the list of elements previously inserted into it by other users. It can insert items into the target HLL sketch, but is otherwise "blind".

S2: The adversary has access to the details of the HLL implementation, i.e. it knows the number of buckets $m = 2^n$ and the hash function h in use. It can insert items into the HLL sketch, but is otherwise blind. In particular, an S2 adversary does not have access to the values of the HLL cardinality estimate at any point in its attack. This kind of attack scenario is appropriate in the context of the port scanning attack of [6].

S3: The adversary has access to the details of the HLL implementation, i.e. it knows the number of buckets $m = 2^n$ and the hash function h in use. Additionally, it can access and interact with the sketch via an API provided by the sketch owner, allowing it to insert items into the HLL sketch and ask for the HLL's cardinality estimate at any point in time. However, it does not know the individual bucket contents.

Table 1: Adversarial capabilities in scenarios S1, S2, S3, and S4.

Scenario	S1	S2	S3	S4
API for shadow device	✓	✗	✗	✗
Insertion of items	✓	✓	✓	✓
h and m known	✗	✓	✓	✓
Get cardinality estimate	✗	✗	✓	✓
Get snapshot	✗	✗	✗	✓

S4: The adversary has direct access to the HLL sketch and all its internals, i.e., the number of buckets, the hash function h, and also the contents of each bucket in the HLL sketch *at one specific point in time*, i.e. it is given a *snapshot* of the sketch. It can access and interact with the sketch via an API provided by the sketch owner, as per scenario S3.

Scenarios S1 and S2 are presented as M2 and M1, respectively, in [8]. S3 is the *external* attack scenario in [10]. S4 is a slightly weaker version of the *insider* attack scenario in [10], the difference being that an insider adversary in [10] has continuous access to the HLL sketch internals including all bucket values, while our adversary in S4 only has access to a snapshot of the HLL sketch internals at the start of its attack. We consider a weaker version of S4 than in [10] because it is sufficient for our attacks; when considering formal security analysis in Section 8, we will switch back to the stronger version of S4.

The adversarial capabilities in each of scenarios S1 to S4 are summarised in Table 1.

4.3 Discussion of Adversarial Scenarios

Scenario S1 is distinctive in that it is the only one in which the adversary has access via an API to a shadow device (and no other capabilities). We do not consider scenario S1 typical of HLL deployments. In a networking application, the authors of [8] argue that the assumptions of S1 could be fulfilled if the adversary learns which machine is used to do the monitoring, buys the same, and uses it as the shadow device. It is possible that the provider uses a different salt or key in the hash per device. This would make any pre-computation on the items using the shadow device irrelevant, and considerably weaken the attack presented in Section 3.2 of [8]. Hence we assume for the rest of this work that, in scenario S1, shadow and targeted devices have the same internals and security parameters (if there are any).[4]

Scenarios S2, S3 and S4 gradually increase the power of the adversary.

The S2 adversary can insert items and knows the hash function h and number of buckets m, but is otherwise blind. Such an adversary is quite realistic. For example, both the Redis and Presto implementations of HLL use a fixed hash function and parameters that can be read off from source code.[5]

The S3 and S4 adversaries have stronger capabilities. The access to cardinality estimates in S3 would typically be provided via an API, while the additional access to the bucket contents in S4 may require insider capabilities. At first sight it might appear that an S3 adversary can easily infer the knowledge given to our S4 adversary in its snapshot. Indeed, an S3 adversary could try to recover the contents of each bucket in the sketch simply by analysing which inputs lead to increases in the cardinality estimate: to find the value held in a targeted bucket, the adversary would insert a sequence of items x with an incrementing number of leading zeros in the ℓ rightmost bits of $h(x)$ whilst holding the n bits determining the bucket index constant, until the cardinality

[4]This scenario is roughly analogous to the common situation where a fixed private key is stored in multiple devices acting as TLS termination points for a single web site.

[5]See `https://github.com/redis/redis/blob/unstable/src/hyperloglog.c#L396`, `https://github.com/redis/redis/blob/unstable/src/hyperloglog.c#L466` for Redis and `https://github.com/prestodb/presto/blob/2ad67dcf000be86ebc5ff7732bbb9994c8e324a8/presto-main/src/main/java/com/facebook/presto/type/khyperloglog/KHyperLogLog.java#L20` for Presto.

estimate of the sketch increases. This assumes the adversary has sufficient flexibility in its input that the conditions on $h(x)$ can be forced.

However, this procedure does not work as simply as just described because of numerical issues that arise in the computation of the cardinality estimate. Specifically, increasing the value held in some bucket from $M[j]$ to $M[j]+1$ *decreases* the denominator in the raw cardinality estimate $\sum_{j=1}^{m} 2^{-M[j]}$ by an amount $2^{-M[j]-1}$. This does lead to a change in the raw cardinality estimate $E = \alpha_m m^2 \cdot (\sum_{j=1}^{m} 2^{-M[j]})^{-1}$. However, in practice, the raw cardinality estimate is rounded before being presented to the adversary, so this change may not actually be detectable, especially when $M[j]$ becomes large. We have done experiments which shows this to be the case for parameters arising in practice.

We do not know of attacks which are able to use the extra information available to an S3 adversary over an S2 adversary (namely, the ability to obtain cardinality estimates). Moreover, as we will see below, our attacks in the S2 setting are already quite powerful. For this reason, we will focus in the remainder of the paper on scenarios S1, S2 and S4.

5 Manipulating HLL Cardinality Estimates

We divide our analysis into two cases: when the HLL sketch is initially empty (and the adversary knows this), and when it is not. For each case, we consider adversaries operating in scenarios S1, S2 and S4. We operate in the situation where the adversary has a set $\mathcal{X}$ of potential inputs which it can choose to insert into the HLL sketch, and where it wishes to insert as many of them as it can whilst keeping the HLL cardinality estimate as low as possible.

5.1 A Simple Attack

In scenarios S2 and S4 with an initially empty HLL sketch, there is a very simple attack based on simply evaluating $h(x)$ for each $x \in \mathcal{X}$, looking at the bits of $h(x)$ to see which bucket j would be affected if x was inserted, and then deciding whether to insert or not based on some rule.

For example, the rule could be to insert x if and only if doing so does not increase the entry $M[j]$ stored in the relevant bucket beyond some pre-defined threshold value v. A simple computation based on the HLL raw cardinality estimator function in Fig. 1 shows that this rule would allow the insertion of a fraction of $1 - 2^{-v}$ of the set $\mathcal{X}$ whilst keeping the raw cardinality estimate below $\alpha_m m 2^v$. This attack assumes h behaves like a random function and exploits the geometric distribution of the position of the leading "1" bit in w, where $w = y_{n+1}y_{n+2}\ldots y_{n+1}$ and $y = h(x)$ (as in the HLL algorithm).

This simple attack is already very effective. For example, setting $v = 1$ in the above rule, the attacker can insert half of all elements of $\mathcal{X}$ whilst keeping the cardinality estimate below $2\alpha_m m$ (where, recall, m is the fixed number of buckets in the HLL sketch).

In the remainder of this section, we target attacks that are more effective than this simple attack.

5.2 Attacking an Initially Empty HLL Sketch

We begin with a preliminary lemma, applying to an adversary operating against an initially empty sketch (in any of S1, S2, S4). As usual, we have $m = 2^n$ buckets, while $n + \ell$ is the output length of the employed hash function h.

Lemma 1. *Suppose an adversary inserts elements x from some set $\mathcal{X}$ into an initially empty HLL sketch. Suppose the adversary can guarantee that in its attack, only a subset of size B out of m buckets are hit, where $B \leq B_{\max}$ with*

$$B_{\max} := \left\lfloor m \cdot \left(1 - \frac{2\alpha_m}{5}\right) \cdot \left(1 - 2^{-(\ell+1)}\right)^{-1} \right\rfloor .$$

Then, at the end of the attack, the HLL cardinality estimate is at most $m \cdot \ln(m/V)$ where $V = m - B$.

Proof. Recall that the Hit Counting algorithm is used in the HLL algorithm as described in Figure 1 when the raw estimate $E = \alpha_m m^2 \cdot (\sum_{j=1}^{m} 2^{-M[j]})^{-1}$ does not exceed $\frac{5}{2}m$. Here $M[j]$ is the counter stored in bucket j. Note that E increases with increasing $M[j]$ and we have B buckets each contributing, in the worst case, $2^{-(\ell+1)}$ to the sum (and hence $m - B$ buckets which contribute $2^0 = 1$ to the sum). Hence, we see that for the Hit Counting algorithm to be used, we require:

$$\alpha_m m^2 \cdot \Big((m-B) + B \cdot 2^{-(\ell+1)}\Big)^{-1} \leq \frac{5}{2}m.$$

Solving for B, we obtain:

$$B \leq B_{\max} := \left\lfloor m \cdot \left(1 - \frac{2\alpha_m}{5}\right) \cdot \left(1 - 2^{-(\ell+1)}\right)^{-1} \right\rfloor.$$

When the Hit Counting algorithm is used, the HLL cardinality estimate is set as $m \cdot \ln(m/V)$ where $V = m - B$ is the number of non-empty buckets. This completes the proof. $\square$

For example with $n = 8$ (so $m = 256$) and $\ell = 24$, we have $\alpha_m = 0.72$ and we find that $B_{\max} = 184$. More generally, since for large m we have $\alpha_m \approx 0.72$, and the term $(1 - 2^{-(\ell+1)})^{-1}$ is always close to but greater than 1, we see that $B_{\max} \approx 0.71 \cdot m$. Note also that $m \cdot \ln(m/V)$ can be rewritten as $m \cdot \ln(1 + \frac{B}{m-B})$.

A surprising implication of the above analysis is that, for the HLL algorithm as described in Figure 1, provided $B \leq B_{\max} \approx 0.71 \cdot m$, no matter which fixed set of B buckets are hit in an attack, and no matter how many leading zero bits the items' hashes have, the final cardinality estimate will not exceed $m \cdot \ln(1 + \frac{B}{m-B})$. Setting, say, $B = m/2$, so half of the buckets can be hit in an attack, we would obtain a bound of $m \cdot \ln 2$ on the cardinality estimate.

5.2.1 Initially Empty HLL Sketch, Scenarios S2 and S4:

Armed with the above lemma, we now consider scenarios S2 and S4. We begin by noting that, when dealing with an initially empty sketch, S2 and S4 are actually equivalent, since the only additional information that an S4 adversary has is a snapshot of the HLL sketch, and this information is effectively given to the S2 adversary by virtue of the attack setting. So we consider S2 and S4 together.

Now our proposed attack is very simple. Suppose the adversary has some set $\mathcal{X}$ of items from which to pick its inputs. The adversary selects a value of $B \leq B_{\max}$, and defines its target buckets to be those numbered $1, \ldots, B$. Since the adversary knows h, it can tell whether, for each $x \in \mathcal{X}$, insertion of x into the HLL sketch would hit one of the target buckets or not; it inserts those that do. In expectation, assuming h behaves like a random function, the adversary will successfully insert a fraction B/m of the items, totalling $(B/m) \cdot |\mathcal{X}|$ items. Meanwhile the final cardinality estimate will remain below $m \cdot \ln(1 + \frac{B}{m-B})$.

For example, setting $B = m/2$ (which, for practical parameters, satisfies $B \leq B_{\max}$), we can expect to insert half of all the items in $\mathcal{X}$ while keeping the cardinality estimate below $(\ln 2)m \approx 0.693m$. This is superior to the simple attack in Section 5.1 (which, recall, could insert half of $\mathcal{X}$ while keeping the cardinality estimate below $2\alpha_m m \approx 1.44m$).

As a second example, setting $B = 1$, we can expect to insert $|\mathcal{X}|/2^m$ items with a final cardinality estimate equal to $m \cdot \ln(1 + \frac{1}{m-1})$, which is rounded to 1 by the HLL algorithm (for m of practical interest, e.g. $m = 256$). Thus we keep the HLL cardinality estimate to a constant (1!) whilst being able to insert a constant fraction of the items from the input set $\mathcal{X}$.

5.2.2 Initially Empty HLL Sketch, Scenario S1:

Here the attack is somewhat more complex. We begin by explaining the core idea, then a simple version of our attack, then a more complex but performant version, and finally, we give a heuristic performance analysis of the more complex attack.

Recall that in scenario S1, the adversary has access to a shadow HLL sketch which behaves identically to the target HLL sketch, and which can be reset at will. We will seek to minimise the number of resets performed in our attack, since we assume these are costly. The adversary can also read out the HLL estimate of the shadow device at will.

Core Idea: We assume the adversary again has (or is given) some set $\mathcal{X}$ of items from which to pick its inputs. The core idea of our attack is to use the first $B \leq B_{\max}$ items $x_1, \ldots, x_B$ from $\mathcal{X}$ to define a set of target buckets, and then to insert into the shadow HLL sketch further items $x \in \mathcal{X}$, checking for a cardinality estimate increase on each insertion. We initialise a list $\mathcal{L}$ containing $x_1, \ldots, x_B$. We add those items that do not increase the cardinality estimate on insertion into the shadow HLL sketch to $\mathcal{L}$ and reject the rest. We reset the shadow device when the cardinality estimate has increased too much, reinserting the first B items from $\mathcal{X}$ again to re-establish a "baseline" of target buckets for the shadow HLL sketch. A key observation is that if in the attack we keep the number of "active" buckets to values B' that are always below $B_{\max}$, then the Hit Counting algorithm is used to compute the cardinality estimate; given its output $m \cdot \ln(m/V)$ (or, more exactly, a rounded version of this value) we can infer the exact value of V, and hence B'. This enables precise detection of cardinality increases, provided we perform a reset before the Hit Counting algorithm stops being used (this is ensured by resetting when the estimate indicates that $B_{\max}$ buckets have been hit). After all of $\mathcal{X}$ has been processed as described, we take the constructed list $\mathcal{L}$ and reprocess it in the same way. We must ensure that the same set of initial items $x_1, \ldots, x_B$ is inserted into the shadow HLL sketch before any other items, so as to fix the same set of target buckets. The order of insertion can be otherwise randomised. By repeated reprocessing, we will gradually remove items that do not hit the B target buckets (defined by $x_1, \ldots, x_B$). Eventually, we obtain a final list $\mathcal{L}^*$ that we insert into the target HLL sketch.

By construction and from Lemma 1, we are guaranteed that the cardinality estimate for the target HLL sketch will be at most $m \cdot \ln(m/V)$ where $V = m - B$. Assuming h behaves like a random function, it is clear that the expected size of the final list $\mathcal{L}^*$ is $\frac{B}{m} \cdot |\mathcal{X}|$. Thus the attack's final performance is the same as that in scenarios S2 and S4 above. However, more processing and resets are required (and the number of resets should be considered as something to be minimised since resetting the shadow device may take a certain amount of time).

Simple Attack: The simple version of our attack uses the above idea, inserting items x into the shadow HLL sketch but resetting it as soon as the cardinality estimate increases from B to $B + 1$ (indicating that a non-target bucket has been hit by item x). Acting in this way precisely determines whether each inserted item hits the target buckets or not, so has no "false positives" which need to be eliminated through reprocessing of the list $\mathcal{L}$. This means that the attack can be carried out in a streaming fashion in a single pass over $\mathcal{X}$. Assuming h behaves like a random function, we must perform a reset with probability $\frac{m-B}{m} = 1 - \frac{B}{m}$ per item inserted, and hence in total an expected number of $(1 - \frac{B}{m}) \cdot |\mathcal{X}|$ resets.

Complex Attack: The more complex version of the attack only resets the shadow HLL sketch when its cardinality estimate reaches a value indicating that $B_{\max}$ buckets have been hit (recall this number can be determined precisely because of the use of the Hit Counting algorithm for cardinality estimation). Hence, fewer resets should be required. However, when the number of active buckets has reached some value $B' \leq B_{\max}$ this means that an item x may not increase the cardinality estimate, yet still not hit one of the B target buckets (instead it hits one of the other $B' - B$ active buckets). Such an x acts as a "false positive" and needs to be eliminated through reprocessing of the list $\mathcal{L}$ (as previously described).

Analysis of the Complex Attack: On each pass, the probability that an item x that should *not* be inserted (i.e. x is a true negative) does *not* increase the cardinality estimate (i.e. x is a false positive on this pass) is given by $(B' - B)/m$, where B' is the current number of active buckets at the point when we try to insert x. (Here we ignore any inconvenient dependence issues across the passes.) This is bounded by $q := (B_{\max} - B)/m$ in the attack, since B' does not exceed $B_{\max}$. The probability that such an x is correctly rejected is at least $1 - q$. Hence the expected number of trial insertions done on x (across different passes of the attack) before x is correctly rejected is $1/(1-q) = m/(m - B_{\max} + B)$.

Now we expect to reject in total $(1 - \frac{B}{m}) \cdot |\mathcal{X}|$ items, and each one needs an expected number of trial insertions that is bounded by $m/(m - B_{\max} + B)$. Meanwhile, each of the $\frac{B}{m} \cdot |\mathcal{X}|$ items in the final insertion list $\mathcal{L}^*$ for the target HLL sketch is trial inserted into the shadow HLL sketch on every pass.

All the passes in which some item x is a false positive do not lead to cardinality estimate increases; only the final pass in which x is identified as a true negative does so. Hence each rejected item only increases the number of active buckets once during the entire attack. Since we allow the number of active buckets to start at B and reach $B_{\max}$ before resetting, the expected total number of resets (and passes) needed is given by:

$$N_R := \frac{1 - \frac{B}{m}}{B_{\max} - B} \cdot |\mathcal{X}|.$$

Hence the expectation of the total number of trial insertions N_I is bounded as:

$$N_I \leq \frac{m}{m - B_{\max} + B} \cdot \left(1 - \frac{B}{m}\right) \cdot |\mathcal{X}| + N_R \cdot \frac{B}{m} \cdot |\mathcal{X}|.$$

As a concrete illustration, we take $B = m/2$ and $B_{\max} = 0.71 \cdot m$, to obtain $N_R = (2.38/m) \cdot |\mathcal{X}|$ and $N_I \leq 0.63 \cdot |\mathcal{X}| + 1.19 \cdot |\mathcal{X}|^2/m$. Here the dominant term in N_I is $\mathcal{O}(|\mathcal{X}|^2)$, i.e. the total insertion cost of this attack in S1 grows as the square of the size of the starting set $\mathcal{X}$ (instead of linearly as in S2 and S4). The end result of the attack is that, as in S2 and S4, an expected $|\mathcal{X}|/2$ items can be inserted into the target HLL sketch while keeping its cardinality estimate below $m \ln 2$.

The insertion cost of the attack can be reduced by more cleverly ordering the items x to be trial inserted on each pass. Note that, in a given pass, when B' is still close to B, an item x that does not increase the number of active buckets is more likely to have hit one of the target buckets than is the case when B' is large; such an x can be tried earlier in the sequence of trials in the next pass and its exact status determined sooner. Any x that can be determined to be a true positive (because its insertion into the shadow HLL sketch keeps the number of active buckets at exactly B) can be placed immediately in $\mathcal{L}^*$ and not reprocessed any further. We leave the analysis of this optimisation to future work.

5.3 Attacking an Initially Non-Empty HLL Sketch

Now we turn to the case where the HLL sketch under attack is not initially empty, but instead contains some honestly inserted items. As usual, the adversary's goal is to insert as many items as possible whilst increasing the cardinality estimate as little as possible. We focus on scenarios S2 and S4, and make brief remarks on S1.

We assume that the target HLL in S2 and S4 has already had sufficiently many items inserted that every one of the $m = 2^n$ buckets has counter value $M[j]$ with $M[j] \geq 1$. This is highly likely to be the case (with probability at least $1 - 1/m^2$) as long as at least $m \ln m$ items have already been inserted into the target HLL. This follows from the standard analysis of the coupon collector problem [18, Section 3.6] under our usual assumption that h behaves as a random function. In S4, the adversary can use its snapshot to detect if this is the case or not, and behave differently if not (details for this case follow below).

With $m \ln m$ honest insertions, it is also highly likely that the small range correction (Hit Counting algorithm) will not be used, and the target HLL's cardinality estimate will be computed using the raw estimate.

5.3.1 Initially Non-empty HLL Sketch, Scenario S2:

The adversary knows h so it selects items x from $\mathcal{X}$ for which the ℓ rightmost bits of $h(x)$ start with a “1” bit. In this case, since the relevant bucket j already has $M[j] \geq 1$, the bucket's counter is not updated, and the cardinality estimate of the HLL sketch remains unchanged.

The adversary then expects to be able to insert half of the items from $\mathcal{X}$ into the HLL sketch, without increasing the HLL cardinality estimate at all.

If some buckets are actually empty (but the adversary does not know which ones in the blind setting of S2), then we may expect a moderate increase in the HLL cardinality estimate in the above attack. If the adversary suspects this is the case, then it can first insert m carefully selected items into the target HLL sketch to ensure that $M[j] \geq 1$ for every bucket. This should increase its cardinality estimate by about m (since HLL is a good cardinality estimator!). Then it can mount the attack, and be sure that it does not increase the cardinality estimate any further.

5.3.2 Initially Non-empty HLL Sketch, Scenario S4:

The adversary can adopt the same strategy as for S2, but relax the sampling requirements with the additional information available in this scenario (namely, the count values $M[j]$ per bucket in the HLL sketch). Specifically, it can allow some leading zero bits in the ℓ rightmost bits of $h(x)$ so long as this does not increase the current bucket estimate. In other words, if $h(x)$ is mapped to the j-th bucket containing $M[j] \geq 0$, then, instead of the strict requirement of having a “1” bit in the first position in the ℓ rightmost bits of $h(x)$, the adversary is satisfied when there is a “1” bit in any of the first $M[j]$ positions in this substring. (A value $M[j] = 0$ then never permits insertions, meaning an empty bucket stays empty.) Under these conditions, the estimates in the buckets are never updated, and there is no effect on the cardinality estimate made by HLL.

We now compute the expected number of elements x from $\mathcal{X}$ such that $h(x)$ meets the requirements of the attack. Assume $h(x)$ is mapped to bucket j. Inserting x does not increase the cardinality estimate if the rightmost ℓ bits of $h(x)$ have $M[j] - 1$ or fewer leading zeroes; this probability is equal to $1 - \frac{1}{2^{M[j]}}$ assuming h behaves like a random function. Averaging over all m buckets (and using the fact that the bucket choice is uniformly random since the bits of h are independent and uniformly random), the probability that input x meets the requirements of the attack is given by:

$$\begin{aligned}\sum_{j=1}^{m} \frac{1}{m} \cdot \left(1 - \frac{1}{2^{M[j]}}\right) &= 1 - \frac{1}{m} \cdot \sum_{j=1}^{m} \left(2^{M[j]}\right)^{-1} \\ &= 1 - H_M^{-1}\end{aligned}$$

where H_M is the harmonic mean of $2^{M[1]}, 2^{M[2]}, \ldots, 2^{M[m]}$. So the adversary can expect to insert a fraction $(1 - H_M^{-1})$ of the items from $\mathcal{X}$ without increasing the HLL cardinality estimate at all.

In this setting, where the target HLL sketch is initially non-empty, we see that an adversary who has sufficient flexibility in its inputs (as represented by the size of the set $\mathcal{X}$) can expect to insert many items into the HLL sketch without increasing the cardinality estimate (or whilst increasing the cardinality estimate slightly in the case where the HLL sketch has some empty buckets in S2). The result is largely independent of the HLL parameters. The more full is the HLL sketch, the larger is $1 - H_M^{-1}$ and the easier it is to insert new items without increasing the cardinality estimate.

5.3.3 Initially Non-empty HLL Sketch, Scenario S1:

We briefly sketch an attack in the setting of an S1 adversary attacking an initially non-empty HLL sketch. An alternative attack with moderate performance in this setting is presented in [8].

Since the target sketch has an unknown status for the adversary, our previous strategy for S1 of keeping many buckets empty no longer works. One can attempt to use the shadow device to identify items that would result in small entries $M[j]$ (for unknown indices j) since such items are unlikely to increase the cardinality estimate of the target device by much. This can be done by first filling the shadow device with sufficiently many items to escape from the "small range correction" phase. Then one could enter candidate items x into the shadow device and check for cardinality increases. However this runs into the rounding issue described in Section 4.3. Still, one could retain those items which do lead to a detectable change in cardinality and reject the others (and also estimate the value of $M[j]$ for the unknown bucket j from the magnitude of the change, retaining only those for which $M[j]$ is small). But here the lack of a cardinality change could result either from rounding issues or because insertion of candidate item x into the shadow device really did lead to no increase in cardinality. This indicates that "good" items would be rejected unnecessarily. This could be addressed by using multiple resets of the shadow HLL sketch, filling it with random items after each reset, and performing multiple insertions per candidate item x to try to estimate the true value of $M[j]$ for each item x (since after at least some of the resets followed by random insertions, we could expect $M[j]$ to increase when x is inserted, and if $M[j]$ is small enough relative to the cardinality estimate, the change could be detected). The details of such an attack require further analysis which we do not pursue here, since we consider S1 to be the least realistic attack scenario.

6 Analysis of the Redis HLL Implementation

We present an analysis of the HLL implementation used in Redis, since Redis is a popular distributed, in-memory key-value store implementation and the version of HLL used there differs significantly from that described in Figure 1.

HLL was first added to Redis in release 2.8.9 in April 2014. Redis 4.0 released in April 2017 introduced HLL Beta, which improves on the raw HLL cardinality estimator by using polynomial interpolation to reduce systematic errors. Redis 5.0 released in October 2018 replaces this with a new cardinality estimator based closely on the work of Ertl [9]. This method is still used in the current release (Redis 6.2.5) and so is the object of our study here.[6]

Ertl [9] proposed using the following cardinality estimator in place of the raw estimator in Figure 1:

$$\hat{\lambda} = \frac{\alpha_\infty m^2}{m\sigma(C_0/m) + \sum_{k=1}^{\ell} C_k 2^{-k} + m\tau(1 - C_{\ell+1}/m)2^{-\ell}} \,. \tag{1}$$

Here, in the denominator, C_k denotes the number of buckets containing value k in the HLL sketch, so C_0 is in particular the number of empty buckets; $\sigma(\cdot)$ and $\tau(\cdot)$ are specific functions defined as:

$$\sigma(x) := x + \sum_{k=1}^{\infty} x^{2^k} 2^{k-1}$$

and

$$\tau(x) := \frac{1}{2}\left(-x + \sum_{k=1}^{\infty} x^{2^{-k}} 2^{-k}\right).$$

Finally, $\alpha_\infty = 0.72134752\ldots$ is a constant. (Note that we have translated from the original notation in [9] to ours.)

Redis 5.0 onwards uses exactly the formula in equation (1) to form its cardinality estimate. It uses $n = 14$, $m = 16,384$ and $\ell = 50$ (so $n + \ell = 64$ and a 64-bit unkeyed hash function is used).

Note that when evaluating (1), functions σ and τ are only called on inputs in the range $[0, 1]$. Moreover, in this range, we have $\sigma(x) \geq x$. It is also highly likely that $C_{\ell+1} = 0$ (since it is very unlikely that a 50-bit all-0 string will result from the application of the Redis HLL hash function, and in scenarios S2 and S4, we can always arrange for this to be avoided). When $C_{\ell+1} = 0$,

[6] For details of Redis releases, see `https://github.com/redis/redis/blob/6.2/00-RELEASENOTES` and other files obtained by replacing "6.2" in this URL with the desired version number.

τ is evaluated at 1, where its value is zero. Combining this information, we see that, provided $C_{\ell+1} = 0$, and letting D denote the denominator in equation (1), we have:

$$D \geq m\sigma(C_0/m) \geq C_0$$

and so

$$\hat{\lambda} \leq \frac{\alpha_\infty m^2}{C_0}$$

where, recall, C_0 is the number of empty buckets.

Attacks against the Redis implementation are now straightforward. Suppose we can arrange that C_0, the number of empty buckets, stays above ϵm, a linear function of the total number of buckets, and that $C_{\ell+1} = 0$. Then we obtain $\hat{\lambda} \leq \alpha_\infty \epsilon^{-1} m$, a linear function of m. For example, we can set $\epsilon = 1/2$ to obtain $\hat{\lambda} \leq 2\alpha_\infty m$, where recall that α_∞ is about 0.72, hence $\hat{\lambda} \leq 1.44m$.

The requirement on $C_0 \geq \epsilon m$ is easily met in scenarios S2 and S4 with an initially empty HLL sketch, through the trick of sampling inputs $x \in \mathcal{X}$ so that only a fraction $1 - \epsilon$ of the buckets are ever hit. We can expect to insert a fraction $1 - \epsilon - \epsilon 2^{-\ell}$ of the elements of $\mathcal{X}$ whilst keeping the cardinality estimate $\hat{\lambda}$ below $\alpha_\infty \epsilon^{-1} m$. (The term $\epsilon 2^{-\ell}$ arises from the requirement to keep $C_{\ell+1} = 0$ in the attack; recall that $\ell = 50$ in Redis, so the effect of this term is very small.)

In scenarios S2 and S4 with an initially non-empty HLL sketch, we assume as for our earlier analysis of the standard HLL algorithm that all buckets have already been hit (this will be the case with high probability as soon as $m \log m$ items have been honestly inserted). Hence $C_0 = 0$ and σ is evaluated at 0 in (1), where its value is 0. This means that the estimate $\hat{\lambda}$ is computed using a denominator of the form:

$$\sum_{k=1}^{\ell} C_k 2^{-k} + m\tau(1 - C_{\ell+1}/m)2^{-\ell}.$$

Now we can proceed exactly as in Sections 5.3.1 and 5.3.2, inserting items x whose hashes have rightmost ℓ bits beginning with a "1" bit (and with a more relaxed condition in S4). These insertions do not alter the values of the C_k, and hence $\hat{\lambda}$ is unaffected. In S2, we expect to insert half the items in $\mathcal{X}$ without increasing the cardinality estimate $\hat{\lambda}$ at all. In S4, the analysis is the same as in Section 5.3.2.

This completes our analysis of Redis. The key takeaway is that, in scenarios S2 and S4, an adversary can expect to insert a constant fraction of $\mathcal{X}$ into a non-empty Redis HLL sketch whilst not increasing its cardinality estimate at all (and whilst keeping the estimated cardinality linear in m in the case where the sketch is initially empty).

7 Experimental Results

Many different implementations of HLL are available online. Among the repositories collecting more than 350 stars on GitHub are [19, 20]. HLL is also featured in many frameworks, including Redis [21], Google's BigQuery [22], Facebook's Airlift [23] and several products of Apache such as Spark [24] and Druid [25]. All these implementations use a constant number of buckets and a fixed hash function. Thus they are vulnerable to our attacks.

As a proof-of-concept, we implemented our attacks against the HLL implementation from [19], since it is faithful to the original description of HLL given in [1], and against the HLL implementation used in Redis [21].[7] For simplicity, we take $\mathcal{X}$ to be the set of integers ranging from 0 to $2^{20} - 1$, so $|\mathcal{X}| = 2^{20}$. We stress that this choice of inputs is merely illustrative, as the attacks are not sensitive at all to the form of the inputs – all that is needed is that the hash function h, acting on those inputs, should have approximately uniform outputs. So 20-bit integers are just as good as, say, IP packet headers, for experimental evaluation.

[7] Our attack code is available on request.

7.1 Results for a "Classic" HLL Implementation

We first focus on a classic HLL implementation as per the original HLL paper [1], namely the implementation of [19]. We choose h to be the 32-bit MurmurHash3 function from [26]; again the attacks do not depend on the specifics of what fixed hash function is used. We create an HLL sketch using $n = 8$ and hence $m = 256$ buckets and, for attacks in the setting of an initially non-empty target HLL sketch, initialise it with 1,000 random items representing honest users' data.

We then challenge the adversary to pick the largest possible number of items to add to the sketch from our set of 2^{20} possible inputs while trying to keep the cardinality estimate as low as possible, in each of the scenarios S1, S2 and S4.

As a second experiment, and in order to get results that can be compared to the work of [8], we mount our attack in their more restrictive setting: instead of picking from the universe of possible strings, the adversary has to choose what to insert from a set of 250,000 random items that are given to it. Finally, for completeness, we also run the attacks against HLL sketches filled with adversarial inputs only, thus that are initially empty. We vary the value of B in the attack for scenario S1 to demonstrate some of the available trade-offs. For simplicity and to keep the attack as striking as possible, we set B to 1 in the attacks on an initially empty HLL sketch in scenarios S2 and S4. All attacks were run 30 times and we report average outcomes. Our results are reported in Tables 2 and 3, which detail the estimated cardinality at the beginning of the attack, the number of items added by the adversary, and the cardinality estimate after adding the adversary's items.

These empirical results confirm our analysis on several points.

First, when attacking a sketch already containing 1000 random items, we can see that the adversary is on average able to insert about half of the 2^{20} items in our starting set in scenario S2, and about 83% in scenario S4. The cardinality estimate is, on average, increased slightly in S2 (from 1,000 to 1,011) and remains at the same starting value (1,005 on average) in S4. In both cases, we inserted more than half a million items whilst increasing the cardinality estimate by about 0.1%! These results align well with our theoretical analysis. Second, when attacking an empty HLL sketch, we can see that in all scenarios, the final cardinality estimate is always exactly a rounded version of the Hit Counting estimator $m \cdot \ln(m/(m - B))$ after inserting a fraction $\frac{B}{m}$ of the items, as predicted by our analysis. For example, in scenario S1 with $B = 100$, the final cardinality estimate is 126 (in agreement with the formula) while with $B = 1$ in scenarios S2, S4, the final cardinality estimate is just 1 (again, as predicted by the formula).

We can also compare our results from Table 3 with results from the attack presented in [8, Section 5.1]. In that paper, the authors target the Redis implementation of HLL and perform the attack against an empty sketch filled with adversarial inputs only. Given a set of 250,000 items, the adversary could choose "74,390 distinct items and obtain an HLL estimate of only 15,780" [8], achieving a five-fold reduction from the true cardinality after 4 iterations (resets). As explained in Section 6, the Redis implementation analysed in [8] does not conform precisely to the original HLL algorithm as presented in Figure 1. Also as discussed in Section 6, the Redis implementation has changed its HLL algorithm several times in recent releases, but the authors of [8] do not specify which version was used in their experiments. This makes it difficult to make a precise comparison with the results of [8]. Nevertheless, the S1 adversary following our strategy against the classical HLL algorithm is able to add 977 and 97,433 distinct items while holding the cardinality estimate at 1 and 126, respectively. Compared to [8], we can reduce the cardinality estimate by a factor of almost 1,000 instead of 5. This comes at the cost of performing more resets compared to [8]. We recall that in practice each reset may cause a delay (for example, it may involve rebooting a network device); we also recall that we consider S1 to be a somewhat artificial attack scenario.

7.2 Results for Redis

We also implemented our attacks for the HLL implementation used in Redis as described in Section 6. The hash function h is fixed as the 64-bit MurmurHash3 function. The seed used

Table 2: Attack results for a classic HLL implementation, averaged over 30 iterations.

Scenario	S1		S2		S4	
# Targeted buckets B	1	100	n/a	1	n/a	1
HLL initially empty?	yes	yes	no	yes	no	yes
Original Card. Est.	0	0	1,000	0	1,005	0
# Items added	4,096	409,595	524,288	28,566	870,320	409,595
Final Card. Est.	1	126	1,011	1	1,005	1

Table 3: Attack results for a classic HLL implementation, averaged over 30 iterations, in the setting of [8].

Scenario	S1		S2		S4	
# Targeted buckets B	1	100	n/a	1	n/a	1
HLL initially empty?	yes	yes	no	yes	no	yes
Original Card. Est.	0	0	1,004	0	998	0
# Items added	977	97,433	124,940	979	204,669	988
Final Card. Est.	1	126	1,058	1	998	1

Table 4: Attack results for HLL in Redis, averaged over 30 iterations.

Scenario	S2	
ϵ	n/a	0.5
HLL initially empty?	no	yes
Original Card. Est.	70,140	0
# Items added	524,455	524,275
Final Card. Est.	73,249	13,188

internally is hardcoded and can be extracted by inspection of the openly-available Redis codebase. Furthermore, the number of buckets is constant and set to 16,384. Knowledge of these parameters places us in the setting of an S2 adversary. Since a classic Redis application separates the server holding the sketch from the client interacting with the sketch, we consider that attack scenarios S3 or S4, while possible, are not typical for Redis. Thus, we only consider scenario S2 for Redis. For simplicity, we set attack parameter ϵ to $\frac{1}{2}$ in the attack on an initially empty HLL sketch. The results are averaged over 30 iterations and reported in Table 4. Similarly to the case for classic HLL, our empirical results match with our theoretical analysis.

7.3 Relating the Attacks to Applications

So far, we have treated attacks in an abstract way, working with an off-the-shelf implementation of HLL. We now illustrate what effect such attacks might have on applications relying on HLL for cardinality estimation. Specifically, we analyse a recent paper [5] from the networking systems community.

The authors of [5] propose to use HLL for recording traffic flow statistics in datacenter network switches. The idea is to use HLL so as to reduce the amount of memory needed in the switches. The resulting switches are called *flexswitches* in [5]. In the main example considered in [5, Section 3.1], each flow is identified by a 4-tuple consisting of source IP address, destination IP address, source port, and destination port. Various parameters for the HLL sketch to be used are then explored in [5], trading memory consumed for accuracy of estimation of the number of concurrent flows. The paper also proposes to use the low range correction method of Fig. 1. At no point does the discussion in [5] consider adversarial input (it does mention worst-case performance but only in passing, and no indication is given of how to handle this). It is clear that, by modifying the 16-bit source port (for example), an adversary could fool the switch into undercounting the flows the adversary is responsible for producing. Depending on how the flow count is used, this could lead to undetected DoS conditions, mis-provisioning of network capacity, or under-billing of the adversary for its traffic. In Appendix A we provide further analysis showing that each use of HLL proposed in [5] can be subverted by an S2 adversary.

8 Formal Security Analysis

In this section we present a formal security analysis of HLL, focussing on how to augment the original HLL design with cryptographic mechanisms and thereby provably prevent attacks. We focus on the case of an initially empty HLL sketch, giving the adversary full control over which items are inserted into it.

8.1 Scenario S1

We begin by disposing of scenario S1. In that scenario, the attacker has access to insertion and reset capabilities, and to cardinality estimates, for a shadow HLL that is identical in every respect to the target HLL. The attacks we presented previously are oblivious to any internals of the implementation (other than that they follow the HLL description in Figure 1). This means that, say, replacing the hash function h with a keyed function or any other cryptographic primitive would be futile in preventing attacks in scenario S1. Interestingly, our attacks in scenario S1 are more complex and require more effort than in scenarios S2 and S4. The fact that we cannot prove security in S1 with improved cryptography is not in contradiction to this, but rather an illustration of the power conferred by giving the adversary access to an exact and resettable replica of the target device in the keyed setting.

In summary, we cannot hope to gain any security in scenario S1 by adding more cryptography.

8.2 Intuition for Formal Security Analysis

For scenarios S2, S3 and S4, the picture is much more positive. In the remainder of this section, we introduce a formal security model covering all three scenarios. We work with a simulation-based framework, in contrast to the game-based approach introduced for probabilistic data structures in [11].

We first begin by explaining the high-level idea of our approach. We allow an adversary $\mathcal{A}$ to interact with an HLL sketch via various oracles that capture its capabilities in each of the three scenarios S2, S3 and S4: it may have an $\texttt{insert}(\cdot)$ oracle allowing it to add items to the HLL sketch, a $\texttt{read}$ oracle that returns the HLL cardinality estimate, and a $\texttt{corrupt}$ that returns all the HLL sketch's bucket contents (but not any secret key K); in S4 it has all three oracles. In addition, in all 3 scenarios, $\mathcal{A}$ has access to an $\texttt{init}(\cdot)$ oracle by which it sets the HLL parameters n, $m = 2^n$, ℓ, α_m used to initialise the HLL sketch. We assume $\mathcal{A}$ always makes exactly one $\texttt{init}(\cdot)$ query and that it is its first oracle call.

In the security model, the game selects at random one of two worlds by sampling a bit b. In the real world ($b = 0$), $\mathcal{A}$ interacts with the scheme HLL-F. This is the same as HLL operating as in Figure 1 but where the hash function h is replaced by a keyed function $F : \{0,1\}^k \times \hat{\mathcal{D}} \to \{0,1\}^{n+\ell}$ whose key K is sampled uniformly at random at the beginning of the game and not available to the adversary; $\mathcal{A}$ has access to the relevant selection of oracles and completes its execution with some output $\texttt{out}_{\mathcal{A}}$ (a bit-string).

In the ideal world ($b = 1$), $\mathcal{A}$ is replaced by a simulator $\mathcal{S}$ with blackbox oracle access to $\mathcal{A}$ (that is, $\mathcal{S}$ can run $\mathcal{A}$ in a blackbox manner, receiving its oracle queries and providing responses to $\mathcal{A}$ as it sees fit). In particular, $\mathcal{S}$ receives parameters n, ℓ, α_m from $\mathcal{A}$'s $\texttt{init}(\cdot)$ call. Finally, $\mathcal{S}$ receives $\texttt{out}_{\mathcal{A}}$ and completes its execution by producing its own output $\texttt{out}_{\mathcal{S}}$.

Importantly, we will specify an $\mathcal{S}$ that handles all of $\mathcal{A}$'s $\texttt{insert}(x)$ queries using *random sampling*, that is, by setting $y \leftarrow_{\$} \{0,1\}^{n+\ell}$ instead of $y := F_K(x)$ and then executing the rest of the insertion process as in Figure 1. In essence, our $\mathcal{S}$ will build its own "shadow" HLL in which all insertion queries are handled by random sampling. $\mathcal{S}$ then uses the shadow HLL for handling $\mathcal{A}$'s `read` and `corrupt` queries. Finally, we will specify $\mathcal{S}$ so that it outputs whatever $\mathcal{A}$ outputs.

We will prove that $\mathcal{S}$ is such that, given any adversary $\mathcal{A}$, then any distinguisher $\mathcal{D}$ that can distinguish between the $\mathtt{out}_{\mathcal{A}}$ in the real world and $\mathtt{out}_{\mathcal{S}}$ in the ideal world can be used to build an adversary $\mathcal{B}$ which breaks the PRF security of F. Under the assumption that F is a good PRF, no such efficient distinguisher $\mathcal{D}$ can exist, and we may conclude that (up to PRF security) no efficient distinguisher can tell that it is interacting with the ideal world instead of the real world.

Why does this proof imply that $\mathcal{A}$ cannot manipulate the cardinality estimate? To see this, note that (amongst many other possibilities) $\mathcal{A}$'s output could be the cardinality estimate obtained from a `read` query, allowing a specific $\mathcal{D}$ to try to use that to distinguish between the two worlds. Yet, by the preceding argument, the simulator in the ideal world is able to produce an output that $\mathcal{D}$ cannot distinguish from that of $\mathcal{A}$ in the real world. In particular, our simulator will output whatever $\mathcal{A}$ outputs, so if $\mathcal{A}$'s output $\mathtt{out}_{\mathcal{A}}$ is the result of a `read` query in the ideal world, then so will $\mathcal{S}$'s. But in the ideal world, $\mathcal{S}$ uses random sampling for the y values, so in that world $\mathcal{S}$ (and $\mathcal{A}$ that it runs) are operating in the setting where the average case analysis of HLL applies, as per [1]. So in the ideal world this specific $\mathcal{D}$ would receive an accurate average case cardinality estimate. Since $\mathcal{D}$ tries to distinguish the two worlds from the cardinality estimate and we already showed that no efficient distinguisher $\mathcal{D}$ can exist, it must follow that the cardinality estimates produced in the real and ideal worlds are indistinguishable (up to PRF security). It then follows that $\mathcal{A}$ has no advantage when it comes to manipulating the HLL-F cardinality estimate. So HLL, when instantiated with PRF F, is secure against cardinality manipulation attacks.

8.3 Formal Security Model for HLL

We now formalise the above intuition. Our definitions are specific to HLL-F as the object of analysis, but are easily generalised to any cardinality estimator with the same syntax as HLL-F (that can be formally defined by a triple of algorithms (`Init`, `Insert`, `Read`)).

The distinguisher $\mathcal{D}$ and its advantage. We consider distinguisher $\mathcal{D}$ to be an algorithm running in a security game. The game begins with the selection of a bit b uniformly at random and then runs either adversary $\mathcal{A}$ (the real world, $b = 0$) or simulator $\mathcal{S}$ (the ideal world, $b = 1$). Adversary $\mathcal{A}$ interacts with up to four oracles which are handled by the game using the algorithms of HLL-F. These are defined below; different combinations of oracles define scenarios S2, S3, S4. After making some queries, $\mathcal{A}$ returns its output $\mathtt{out}_{\mathcal{A}}$ to $\mathcal{D}$. Simulator $\mathcal{S}$ runs $\mathcal{A}$ in a blackbox manner (as specified below) and receives $\mathcal{A}$'s output. It then provides its own output $\mathtt{out}_{\mathcal{S}}$ to $\mathcal{D}$. Finally, $\mathcal{D}$ outputs a bit b' estimating whether it is interacting with the real world ($b' = 0$) or the ideal world ($b' = 1$). For $i = 2, 3, 4$, we define $\mathcal{D}$'s Si-Real-or-Ideal advantage against HLL-F with adversary $\mathcal{A}$ and simulator $\mathcal{S}$ to be:

$$\mathrm{Adv}^{\mathrm{S}i\text{-RoI}}_{\text{HLL-}F,\mathcal{A},\mathcal{S}}(\mathcal{D}) := |2\Pr[b' = b] - 1|\,.$$

The adversary $\mathcal{A}$ and its oracles. The four oracles that $\mathcal{A}$ can interact with behave as follows in the real world:

- `init(·)`: this oracle receives as input the HLL-F parameters (n, ℓ, α_m) and performs initialisation of the HLL-F sketch, including generating the array of $m = 2^n$ buckets $M[j]$ and initialising all the entries to 0, and selecting a key $K \leftarrow_{\$} \{0,1\}^k$ for keyed function $F : \{0,1\}^k \times \hat{\mathcal{D}} \to \{0,1\}^{n+\ell}$. This oracle has no output. It is available in all of S2, S3, S4. We assume it is only called once as $\mathcal{A}$'s first oracle query.

- `insert(·)`: on input $x \in \hat{\mathcal{D}}$, the oracle implements HLL-F by setting $y = F_K(x)$ and following the procedure in Figure 1 to insert x into the HLL sketch (that is, it uses the first n bits of y to determine a bucket index j and the next ℓ bits of y to determine a bit-string w; then $\rho(w)$ is used to update the bucket at index j via $M[j] := \max(M[j], \rho(w))$). This oracle has no output. It is available in all of S2, S3, S4. Since HLL and variants of it we consider here have the property that reinserting an item x does not change the internal state of the HLL sketch (nor the cardinality estimate computed from it) we assume without loss of generality that all queries x made by $\mathcal{A}$ to `insert(·)` are distinct.

- `read`: this oracle has no input and returns the HLL-F cardinality estimate as per Figure 1, i.e. it computes a raw cardinality estimate via $E := \alpha_m m^2 \cdot (\sum_{j=1}^{m} 2^{-M[j]})^{-1}$ and then performs corrections for small and large values. We allow multiple queries to this oracle and make it available to $\mathcal{A}$ in S3 and S4, but not S2.

- `corrupt`: this oracle has no input and returns the vector $(M[j])_{j=1}^{m}$ of HLL bucket contents (but not any secret key K). This oracle is available only in S4. We allow the adversary to query `corrupt` as often as it wishes during its attack. Our description of S4 in Section 4 limited the adversary to at most one such query; here we allow multiple queries because doing so does not increase the difficulty of achieving security.

We note that in S4 (and only in S4) the adversary has access to all four oracles. This makes it the strongest model, and hereafter we only target security in this model.

A specific simulator $\mathcal{S}$. In order to obtain a meaningful security definition that prevents cardinality manipulation, we need to restrict $\mathcal{S}$ to doing random sampling when handling `insert(·)` queries coming from $\mathcal{A}$. Otherwise, our definition could be vacuously satisfied (for example $\mathcal{S}$ could just sample the PRF key K for itself and provide a perfect real-world view). This is somewhat different to typical simulation-based security definitions, where $\mathcal{S}$'s behaviour is not restricted but its input might be (e.g. in a confidentiality definition, $\mathcal{S}$ might receive only $|m|$ in place of plaintext m). Crucially, in our setting, security arises from inspecting the details of $\mathcal{S}$ and observing that, due to its use of random sampling, it instantiates HLL in the average case.

In full detail, our specific $\mathcal{S}$ operates by running $\mathcal{A}$ in a blackbox manner and handling all of its queries as follows:

- `init(·)`: on input n, ℓ, α_m, $\mathcal{S}$ initialises an HLL sketch as in Figure 1; that is, it creates an array of $m = 2^n$ buckets $M[j]$ and initialises them all to 0.

- `insert(·)`: on input $x \in \hat{\mathcal{D}}$, $\mathcal{S}$ ignores x, sets $y \leftarrow_\$ \{0,1\}^{n+\ell}$, and proceeds to update the buckets using this value of y as in the original HLL algorithm in Figure 1.

- `read`: $\mathcal{S}$ returns the HLL cardinality estimate computed as per Figure 1.

- `corrupt`: $\mathcal{S}$ returns the current vector $(M[j])_{j=1}^{m}$ of HLL bucket contents that it has stored.

At the end of $\mathcal{A}$'s execution, $\mathcal{S}$ takes $\mathcal{A}$'s output $\mathtt{out}_{\mathcal{A}}$ and forwards it to $\mathcal{D}$ as its own output $\mathtt{out}_{\mathcal{S}}$. Notice that the running time $t_{\mathcal{S}}$ of $\mathcal{S}$ is (roughly) equal to that of $\mathcal{A}$.

Formal security definition and main result. Now we are ready to give:

Definition 1. *The scheme HLL-F is Si-simulation-secure with parameters $(q, t_{\mathcal{S}}, t_{\mathcal{A}}, t_{\mathcal{D}}, \epsilon)$ if, for the specific simulator $\mathcal{S}$ given above, for all adversaries $\mathcal{A}$ running in time at most $t_{\mathcal{A}}$ and making at most q `insert(·)` queries, and for all distinguishers $\mathcal{D}$ running in time at most $t_{\mathcal{D}}$, we have $Adv^{Si\text{-}RoI}_{HLL\text{-}F,\mathcal{A},\mathcal{S}}(\mathcal{D}) \leq \epsilon$.*

In the above definition, we quantify over `insert(·)` queries (via q) and not other types of query because only `insert(·)` queries will appear in our concrete security analysis and we do not want to clutter the notation. Note also that our definition is given with respect to a *specific* simulator, but could be generalised to allow any simulator that performs random sampling when handling `insert(·)` queries.

Our simulation-based security definition does not require us to specify a winning condition for adversary $\mathcal{A}$. Rather, we infer from inspecting the simulator what the adversary can (or cannot) achieve in the ideal world, and show that no efficient algorithm $\mathcal{D}$ can distinguish the real and ideal worlds. This is in contrast to the game-based approach of [11] which needs a complex definition to say when the adversary has successfully manipulated a probabilistic data structure. This reveals a strength of the simulation-based approach.

Before stating our main result, we require one further standard definition:

Definition 2. *Let $F : \{0,1\}^k \times \hat{\mathcal{D}} \to \{0,1\}^{n+\ell}$. We define the PRF advantage of an adversary $\mathcal{B}$ against F outputting a bit d' as being its advantage in distinguishing whether it is interacting with F for a uniformly random key K ($d = 0$) or interacting with a truly random function ($d = 1$). We write:*

$$Adv_F^{PRF}(\mathcal{B}) := |2\Pr[d' = d] - 1|\,.$$

Here, we assume without loss of generality that all of $\mathcal{B}$'s queries are distinct. See [27, Section 4.4.1] for details of the PRF security definition.

Now we are ready to formally state our main result. The proof can be found in Appendix B.

Theorem 1. *Let simulator $\mathcal{S}$ be as described above. For any S4 adversary $\mathcal{A}$ against HLL-F and any distinguisher $\mathcal{D}$, we can construct a PRF adversary $\mathcal{B}$ against F such that:*

$$Adv_F^{PRF}(\mathcal{B}) = Adv_{HLL\text{-}F,\mathcal{A},\mathcal{S}}^{S4\text{-}RoI}(\mathcal{D})\,.$$

Moreover, if $\mathcal{A}$ makes q queries to its `insert(·)` *oracle, then $\mathcal{B}$ makes q queries to its real or random oracle. Finally, the running time of $\mathcal{B}$ is (roughly) $t_{\mathcal{A}} + t_{\mathcal{D}}$.*

This result tightly relates the security of HLL-F against cardinality manipulation to that of the PRF security of the function F used in its construction (for the reasons explained in Section 8.2).

8.4 On Confidentiality

An HLL sketch can leak information about items x that were previously inserted. Consider the following attack from [10]: to test whether an item x was previously inserted into the HLL by an honest user (say), the adversary makes a sequence of queries: `read`, `insert(`x`)`, `read`. If the two cardinality estimates are different, the adversary can conclude definitively that x was *not* previously inserted into the HLL; if they are the same, it can estimate (with some error) that x was previously inserted.[8] This attack still works even if HLL is implemented using a PRF and is unavoidable in view of the oracle queries (and in real settings, the API) available to the adversary. So such privacy leaks are an inherent feature of HLL if a rich enough API is provided. Note that this attack does not contradict our main result, which concerns prevention of cardinality manipulation rather than confidentiality.

We leave the exploration of the adaptation of our simulation-based security definitions to capture confidentiality notions for more restricted APIs to future work.

8.5 Is Universal Hashing Sufficient?

It is natural to wonder whether a PRF is really required in place of hash function h in order to make HLL secure, or whether a weaker cryptographic primitive would suffice. For example, since it has (roughly) uniform outputs, perhaps using a universal hash function (UHF) $h_K(\cdot)$ would be sufficient. In Appendix C we show that, in general, UHFs do not suffice by providing a key recovery attack against HLL when h is replaced by a specific UHF based on polynomial evaluation over a finite field.

[8]Errors can arise in two ways when x was not previously inserted: either inserting x changes a bucket value, but the change is masked by rounding when the cardinality estimate is computed, or inserting x does lead to a change in a bucket value because $\rho(w)$ computed from x is too small.

8.6 Instantiating the PRF

Our HLL construction requires a PRF $F : \{0,1\}^k \times \hat{\mathcal{D}} \to \{0,1\}^{n+\ell}$. Here, the set $\hat{\mathcal{D}}$ represents the space of possible inputs to HLL which can be modelled by a set of bit-strings of some maximum length. This means that F needs to be Variable Input Length PRF (VIL-PRF).

We want F to be competitive with non-cryptographic hashing on small input sizes, as typical in current HLL implementations and applications (e.g. MurmurHash3 applied to an IP header). This makes off-the-shelf approaches like HMAC uncompetitive, since even for the shortest inputs, HMAC would require 4 evaluations of the underlying hash compression function.

SipHash [28] is designed specifically for this use case. It is competitive in speed with non-cryptographic hash functions commonly used in HLL implementations such as MurmurHash3. It has also withstood cryptanalysis well [29, 30]. It has 128-bit outputs that can be truncated to the needs of typical HLL implementations (32 and 64 bits). Our microbenchmarks created on an Intel i7 CPU show that SipHash-2-4 on 90-byte inputs requires 102ns per evaluation, while the 128-bit version of MurmurHash3 takes 60ns. Meanwhile, HMAC-SHA256 requires 705ns. We obtained similar performance ratios for other input sizes. These together show that SipHash is quite competitive with MurmurHash3.

We note that using keyed functions sacrifices the mergeability property that HLL enjoys, if different keys K are used in different HLL sketches. The reason is that one can no longer emulate running a single HLL sketch by taking the maximum over all buckets with each index j in the different sketches, since now the bucket contents are created using different keys in a PRF and hence bear no relation to one another.

9 Conclusion

The HyperLogLog algorithm is an elegant and efficient solution to the problem of estimating the cardinality of large sets. Its simple structure makes it easy to code and use, as shown by the growing number of available open-source implementations. Nonetheless, we have shown that malicious users can manipulate the HLL cardinality estimate and thence break the security properties of systems relying on HLL. Our attacks are simple but powerful and should raise awareness of the limitations of HLL. Our analysis may assist software developers in understanding the risks they run when using HLL in adversarial settings. We have also provided a method for securing HLL in adversarial settings: simply replace the internal hash function by a keyed PRF, and one obtains full protection even in scenario S4, our strongest attack scenario.

Left to future work is the task of extending our attacks and defences to HLL variants such as sliding HLL [7] and HLL++ [3]. It would be interesting to find and analyse a strategy for more cleverly ordering the candidates in our attack in scenario S1 to reduce the total insertion cost of the attack, whilst also keeping the number of resets low. Finding an attack in scenario S1 when the target HLL sketch is initially non-empty is also open.

We have given a formal analysis of using a PRF as a countermeasure to our attacks. The use of a keyed function sacrifices HLL mergeability when the keys in the different HLLs are independently generated. It is an open problem to find methods for securing HLL sketches against adversarial inputs whilst retaining some form of mergeability. This may also require the introduction of entirely new data structures that trade increased storage for mergeability and improved security.

Acknowledgements

The work of Paterson is supported in part by a gift from VMware.

The authors thank Dennis Hofheinz and Anupama Unnikrishnan for useful discussions on simulation-based definitions and Tom Shrimpton for suggesting to look at GoogleBigQuery.

References

[1] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm," in *Proceedings of the International Conference on Analysis of Algorithms (AOFA)*, pp. 127–146, 2007.

[2] M. Honarkhah and A. Talebzadeh, "HyperLogLog in Presto: A significantly faster way to handle cardinality estimation." Online: `https://engineering.fb.com/data-infrastructure/hyperloglog/`, 2018. Accessed: 2021-04-30.

[3] S. Heule, M. Nunkesser, and A. Hall, "HyperLogLog in practice: algorithmic engineering of a state of the art cardinality estimation algorithm," in *Proceedings of the 16th International Conference on Extending Database Technology*, pp. 683–692, 2013.

[4] A. Hall, O. Bachmann, R. Büssow, S. Ganceanu, and M. Nunkesser, "Processing a trillion cells per mouse click," *Proc. VLDB Endow.*, vol. 5, no. 11, pp. 1436–1446, 2012.

[5] N. K. Sharma, A. Kaufmann, T. Anderson, C. Kim, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the Power of Flexible Packet Processing for Network Resource Allocation," in *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, pp. 67–82, 2017.

[6] Y. Chabchoub, R. Chiky, and B. Dogan, "How can sliding HyperLogLog and EWMA detect port scan attacks in IP traffic?," *EURASIP Journal on Information Security*, pp. 1–11, 2014.

[7] Y. Chabchoub and G. Hebrail, "Sliding HyperLogLog: estimating cardinality in a data stream over a sliding window.," in *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops*, pp. 1297–1303, 2010.

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

[9] O. Ertl, "New cardinality estimation methods for HyperLogLog sketches," *CoRR*, vol. abs/1706.07290, 2017.

[10] D. Desfontaines, A. Lochbihler, and D. Basin, "Cardinality Estimators do not Preserve Privacy," in *Proceedings on Privacy Enhancing Technologies*, pp. 26–46, 2019.

[11] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic data structures in adversarial environments," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019* (L. Cavallaro, J. Kinder, X. Wang, and J. Katz, eds.), pp. 1317–1334, ACM, 2019.

[12] M. Naor and E. Yogev, "Bloom filters in adversarial environments," in *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part II* (R. Gennaro and M. Robshaw, eds.), vol. 9216 of *Lecture Notes in Computer Science*, pp. 565–584, Springer, 2015.

[13] T. Gerbet, A. Kumar, and C. Lauradoux, "The power of evil choices in Bloom filters," in *45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2015, Rio de Janeiro, Brazil, June 22-25, 2015*, pp. 101–112, IEEE Computer Society, 2015.

[14] R. J. Lipton and J. F. Naughton, "Clocked adversaries for hashing," *Algorithmica*, vol. 9, no. 3, pp. 239–252, 1993.

[15] S. A. Crosby and D. S. Wallach, "Denial of service via algorithmic complexity attacks," in *Proceedings of the 12th USENIX Security Symposium, Washington, D.C., USA, August 4-8, 2003*, USENIX Association, 2003.

[16] M. Durand and P. Flajolet, “LogLog Counting of Large Cardinalities,” *Annual European Symposium on Algorithms (ESA)*, pp. 605–617, 2003.

[17] K.-Y. Whang, B. Vander-Zanden, and H. Taylor, “A linear-time probabilistic counting algorithm for database applications,” *ACM Transactions on Database Systems*, pp. 208–229, 1990.

[18] R. Motwani and P. Raghavan, *Randomized algorithms.* Cambridge University Press, 1995.

[19] “Hyperloglog by clarkduvall.” Online: `https://github.com/clarkduvall/hyperloglog`. Accessed: 2020-04-28.

[20] “Hyperloglog by ekzhu.” Online: `https://github.com/ekzhu/datasketch/blob/master/datasketch/hyperloglog.py`. Accessed: 2020-04-28.

[21] “Hyperloglog by antirez.” Online: `https://github.com/antirez/redis/blob/unstable/src/hyperloglog.c`. Accessed: 2020-04-28.

[22] “Google’s bigquery.” Online: `https://github.com/google/zetasketch/blob/master/java/com/google/zetasketch/HyperLogLogPlusPlus.java`. Accessed: 2020-04-28.

[23] “Facebook airlift.” Online: `https://github.com/airlift/airlift/blob/master/stats/src/main/java/io/airlift/stats/cardinality/HyperLogLog.java`. Accessed: 2020-04-28.

[24] “Apache spark.” Online: `https://github.com/swoop-inc/spark-alchemy/tree/master/alchemy/src/main/scala/com/swoop/alchemy/spark/expressions/hll`. Accessed: 2020-04-28.

[25] “Apache druid.” Online: `https://github.com/apache/druid/blob/master/hll/src/main/java/org/apache/druid/hll`. Accessed: 2020-04-28.

[26] “Murmur3 by spaolacci.” Online: `https://github.com/spaolacci/murmur3`. Accessed: 2020-04-28.

[27] D. Boneh and V. Shoup, *A Graduate Course in Applied Cryptography.*

[28] J. Aumasson and D. J. Bernstein, “SipHash: A fast short-input PRF,” in *Progress in Cryptology - INDOCRYPT 2012, 13th International Conference on Cryptology in India, Kolkata, India, December 9-12, 2012. Proceedings* (S. D. Galbraith and M. Nandi, eds.), vol. 7668 of *Lecture Notes in Computer Science*, pp. 489–508, Springer, 2012.

[29] C. Dobraunig, F. Mendel, and M. Schläffer, “Differential cryptanalysis of SipHash,” in *Selected Areas in Cryptography - SAC 2014 - 21st International Conference, Montreal, QC, Canada, August 14-15, 2014, Revised Selected Papers* (A. Joux and A. M. Youssef, eds.), vol. 8781 of *Lecture Notes in Computer Science*, pp. 165–182, Springer, 2014.

[30] W. Xin, Y. Liu, B. Sun, and C. Li, “Improved cryptanalysis on siphash,” in *Cryptology and Network Security - 18th International Conference, CANS 2019, Fuzhou, China, October 25-27, 2019, Proceedings* (Y. Mu, R. H. Deng, and X. Huang, eds.), vol. 11829 of *Lecture Notes in Computer Science*, pp. 61–79, Springer, 2019.

[31] E. R. Berlekamp, “Factoring polynomials over finite fields,” *Bell System Technical Journal*, vol. 46, pp. 1853–1859, 1967.

A Further analysis of Flexswitches

Extending our analysis of [5] from the main body, [5, Section 4.2] proposes using HLL for detecting TCP port scans by counting observed destination port numbers in TCP packets with set SYN flags and reporting a scan event if the estimate exceeds a set threshold. Using our attack techniques (specifically, in scenario S2), an attacker could complete a partial port scan of roughly half of all 2^{16} ports whilst keeping the HLL estimate below $0.693m$ where m is the number of buckets used in HLL. While [5] is not specific about the value of m actually used, the authors' focus on reducing memory usage indicates that m would not be larger than 1024.[9] It is not stated in [5] what threshold needs to be crossed to trigger an alarm, but we consider it plausible that it would be above $0.693 \cdot 1024 = 710$.

Similarly, [5, Section 4.2] proposes using HLL for detecting NTP amplification attacks by counting distinct source IP addresses arriving on UDP port 123 over a small time window. No further details are given, but it is clear that an S2 attacker with even limited flexibility in spoofing source IP addresses (e.g. within a private range used in the datacenter) could easily evade detection whilst delivering a large number of NTP requests, thus re-enabling the original attack.

In a more advanced application, [5, Section 4.3] proposes to use HLL as a component in a novel, low-memory version of the RCP congestion control algorithm called RCP-Approx. Again HLL is used for estimating numbers of traffic flows. Obviously, our attack techniques could be used (in scenario S2) to cause RCP-Approx to underestimate the number of traffic flows, leading to non-detection of congestion and so enabling DoS attacks to go undetected.

Similar remarks apply to the treatment of CONGA in [5, Section 4.5]. CONGA is a congestion-aware load balancing system that splits TCP flows into flowlets (collections of IP packets) and allocates them to paths based on congestion feedback from remote switches. In the enhancements to CONGA proposed in [5], HLL is used in the "flow statistics" building block for estimating flowlet statistics in switches; these estimates are sent back to a monitoring system and used to determine the load balancing configuration. This overall system can be abused by an S2 adversary, to cause flowlet numbers to be underestimated (or overestimated) and the load balancing to perform sub-optimally. For example, an adversary could cause all the switches to *over*-estimate the number of flowlets they are handling and ask the load balancing system to send less traffic through them. This could result in a degradation in the total amount of traffic the load balancing system can handle (depending on what fail-safe mode it has). Or an attacker could cause all the switches to *under*-estimate the number of flowlets they are handling, causing undetected congestion in the switches.

B Proof of Theorem 8.1

PRF adversary $\mathcal{B}$ against F is built as follows. Let d denote the hidden bit determining whether $\mathcal{B}$ receives real responses of the form $F_K(x)$ where K is a uniformly random key ($d = 0$) or responses from a truly random function ($d = 1$); $\mathcal{B}$ provides a simulation of either the real or ideal world for distinguisher $\mathcal{D}$ as follows. $\mathcal{B}$ runs $\mathcal{A}$ and answers each of $\mathcal{A}$'s queries as follows:

- `init(·)`: $\mathcal{B}$ receives parameters n, ℓ, α_m, sets $m = 2^n$ and initialises an array $M[j]$ of size m in which each of the entries is set to 0.
- `insert(·)`: on input $x \in \hat{\mathcal{D}}$, $\mathcal{B}$ makes a query on x to its real-or-random oracle and receives value y; $\mathcal{B}$ uses y to update the HLL sketch as in the regular HLL algorithm (shown in Figure 1). That is, it uses the first n bits of y to determine a bucket index j and the next ℓ bits of y to determine a bit-string w; then $\rho(w)$ is used to update the bucket at index j via $M[j] := \max(M[j], \rho(w))$.
- `read`: $\mathcal{B}$ uses its knowledge of the buckets' contents to compute and return the HLL cardinality estimate as per Figure 1.

[9]The paper mentions using 1 byte counters in the buckets and states that using more than 1KB of memory "only improves average error rates marginally"; we infer a maximum value of $m = 1024$ from this.

- `corrupt`: $\mathcal{B}$ returns the vector $(M[j])_{j=1}^{m}$.

When $\mathcal{A}$ returns its final output $\mathtt{out}_{\mathcal{A}}$, $\mathcal{B}$ forwards this output to $\mathcal{D}$. Finally, $\mathcal{D}$ outputs a bit b'. Then $\mathcal{B}$ outputs its own bit $d' = b'$.

Note that when $d = 0$, $\mathcal{B}$ plays against the real F with a random key K and provides a simulation for $\mathcal{A}$ that perfectly instantiates the real world. On the other hand, when $d = 1$, $\mathcal{B}$ plays against a truly random function and perfectly instantiates our simulator $\mathcal{S}$ in the ideal world. To see this, note that sampling the outputs y of a truly random function at distinct inputs x is identical to simply sampling y uniformly at random.[10] The equality of advantages in the statement of the theorem now follows immediately. The remainder of the proof follows on counting queries and noting that the HLL sub-algorithms for insertion and computing cardinality are efficient, so $\mathcal{B}$'s running time due to handling $\mathcal{A}$'s queries is not increased significantly compared to that of $\mathcal{A}$. The final running time for $\mathcal{B}$, namely $t_{\mathcal{A}} + t_{\mathcal{D}}$, follows on noting that $\mathcal{B}$ first runs $\mathcal{A}$ and then $\mathcal{D}$.

C Breaking HLL-UHF

In general the key of a UHF may be recovered from a small number of its outputs. Consider, for example, the UHF built from polynomial hashing over a finite field $\mathcal{F}$. Here, inputs x are encoded as polynomials $p_x \in \mathcal{F}(Z)$, the key is a uniformly random field element, and the UHF output $h_K(x)$ is equal to $p_x(K)$, obtained by polynomial evaluation at K. The specific translation from x to p_x involves encoding x as a vector $(x_1, x_1, \ldots, x_v) \in \mathcal{F}^v$ for some v and then setting:

$$p_x(Z) = Z^v + x_1 Z^{v-1} + \cdots + x_{v-1} Z + x_v.$$

An adversary who knows the input x, and therefore the vector $(x_1, x_2, \ldots, x_v)$ can determine the key K by finding roots of polynomials over $\mathcal{F}$, using Berlekamp's algorithm [31]. If the adversary can choose x, then it can make the polynomials p_x have low degree, making the attack easier to perform.

Now consider using such a UHF in place of h in attack scenario S4. For concreteness, we assume $\mathcal{F} = \mathsf{GF}(2^{128})$, so our UHF has 128-bit keys and 128-bit outputs. The output is then truncated to provide the 32 bits needed by HLL (or, say, the 64 bits consumed by the Redis variant of HLL). The encoding step, taking bit-string x to vector $(x_1, x_2, \ldots, x_v)$, simply splits x into 128-bit strings x_i, each x_i representing a field element. The attack we present below is specific to this UHF, but can be extended to other cases.

An S4 adversary $\mathcal{A}$ can perform a series of pairs of `insert`(x^i) and `corrupt` queries on chosen items $x^0, x^1, \ldots, x^{t-1}$. With high probability (assuming t is not too large), each `insert` query hits a fresh bucket in the HLL sketch. Then comparing the results of the `corrupt` queries either side of the `insert`(x^i) query reveals to $\mathcal{A}$ which bucket was hit by that `insert` query. In turn, this leaks partial information about the corresponding UHF output $p_{x^i}(K)$, specifically, its n least significant bits (since those bits are used to generate the bucket index for insertion). We now show how to exploit this partial information on $p_{x^i}(K)$ to recover K.

By choosing x^i to have 256 bits, we have $v = 2$ and $p_{x^i}(K) = K^2 + x_1^i K + x_2^i$; then further selecting x^i such that x_2^i is a constant value (e.g. $0 \in \mathsf{GF}(2^{128})$) and considering consecutive pairs of x^i items, we obtain equations of the form:

$$p_{x^{2j+1}}(K) \oplus p_{x^{2j}}(K) = (x_1^{2j+1} \oplus x_1^{2j})K$$

where $\oplus$ denotes addition over $\mathsf{GF}(2^{128})$. Here, the n least significant bits of the left-hand side leak through the indices of the buckets hit for each index. By selecting the x_1^i uniformly at random and analysing the equations of the above form that result, we obtain a linear system over $\mathsf{GF}(2)$ in the unknown bits of K. The number t of x^i needed to recover K in its entirety is roughly $2 \cdot 128/n$ since we get n independent linear equations per pair (x^{2j}, x^{2j+1}) and there are 128 unknown bits in K. The required value of t is small for typical HLL parameters (e.g. $n = 16$ would yield $t = 32$).

[10]Our proof relies on the x being distinct, but this is to simplify book-keeping and is without loss of generality.

The above attack exploits specific properties of the chosen UHF, but suffices to demonstrate that UHFs are not a suitable tool for protecting HLL in general. It is still possible that UHFs could be safely used in weaker attack scenarios, e.g. S2. We leave a detailed study to future work.