Improving Learned Bloom Filters

Gordon Lu, Sanchit Sahay and Subhash Kotaru
The full implementation and experiments for this report are available at:
`Codebase and Results (https://shorturl.at/KfjDK)`

May 10, 2025

Introduction

The Bloom filter is a classical space-efficient data structure for representing a set $S \subseteq U$ while supporting approximate membership queries. It enables testing whether an element $x \in U$ belongs to S with no false negatives and a tunable false positive rate. A standard Bloom filter consists of a bit array of length m and k independent hash functions $h_1, \ldots, h_k : U \to [m]$. To insert an element $x \in S$, the filter sets the bits at positions $h_1(x), \ldots, h_k(x)$ to 1. A query for x checks whether all of these positions are set. If so, the filter returns "yes"; otherwise, it returns "no". The probability of a false positive for an element not in S is approximately $(1 - e^{-kn/m})^k$ under the assumption of fully random hash functions and uniform insertions. Bloom filters are widely used in databases, networking, and security due to their simplicity and memory efficiency.

While Bloom filters are effective in many applications, they are agnostic to the structure of the key set S and the distribution over queries. They do not adapt to patterns or correlations in the data, and their memory consumption scales linearly with the number of stored keys. To address this, the Learned Bloom Filter (LBF) was introduced by Kraska et al. [1], and later formalized by Mitzenmacher [2]. The LBF integrates a learned model into the membership query process in order to reduce the load on the backup filter and thereby improve space efficiency.

The Learned Bloom Filter consists of three components: a classifier $f : U \to [0, 1]$ trained to approximate the indicator function of the key set S, a threshold parameter $\tau \in [0, 1]$, and a conventional Bloom filter B that stores those keys in S for which $f(x) < \tau$. Given a query element $x \in U$, the LBF first evaluates $f(x)$. If $f(x) \geq \tau$, the element is accepted as a member of S. If $f(x) < \tau$, the query is passed to the backup filter B, which then determines membership using traditional Bloom filter logic. This two-stage structure ensures that all true members of S are accepted—either directly by the classifier or through the backup—while aiming to reduce the number of elements stored in B. For elements not in S, the probability of a false positive is given by the expression $\mathbb{P}[\text{LBF}(x) = 1 \mid x \sim \mathcal{D}] = \alpha + (1 - \alpha)\beta$, where α denotes the probability that the classifier incorrectly accepts a non-key, and β denotes the false positive rate of the backup filter among rejected queries. This expression captures the combined error arising from both components and highlights the importance of balancing classifier quality with the capacity of the backup filter.

In this work, we conduct an empirical evaluation of the Learned Bloom Filter and several architectural variants using a labeled dataset of malicious and benign URLs. Our objectives are: first, to examine how varying the classifier threshold τ impacts the false positive rate, backup filter size, and overall accuracy of the learned filter; second, to evaluate enhancements to the backup filter, including the use of a Cuckoo filter in place of a Bloom filter and the introduction of a Least Recently Used (LRU) cache to exploit temporal locality in repeated query patterns; third, to compare the empirical false positive rates with theoretical predictions based on Mitzenmacher's analytical model; and fourth to check the efficacy of using specialized hashing functions, and the practical aspects of training these.

While the Cuckoo filter supports deletions and has been proposed as a more flexible alternative to Bloom filters, we observe that, in the absence of a deletion or eviction policy, its space efficiency is worse than that of a Bloom filter under equivalent conditions. This motivates our final variant, which combines a Cuckoo filter with an LRU cache to actively manage the backup set. Across all methods, we evaluate performance using both empirical metrics and model-based expectations, aiming to quantify the tradeoffs inherent in learned filtering systems and identify when structure-aware filtering leads to meaningful gains.

In this work, we conduct an empirical evaluation of the Learned Bloom Filter and compare it with an alternative approach, Low-Rank Adaptation (LoRA) **[hu2021lora]**, for the task of URL classification. Our objectives are: (1) to examine how each approach balances memory efficiency and accuracy, (2) to evaluate training and inference costs, and (3) to determine which method is better suited for different deployment scenarios.

Dataset and Experimental Setup

Classification-based LBF

To evaluate the performance of the Learned Bloom Filter and its variants, we use a labeled dataset of URLs annotated as either malicious or benign [3]. The malicious URLs define the key set $S \subseteq U$, which the filter aims to represent, while the benign URLs are used to train the classifier and to simulate the query distribution $\mathcal{D}$ over $U \setminus S$. The underlying learning task is to distinguish members of S from non-members based on the syntactic and lexical features of the URLs.

We partition the malicious URL set into two disjoint subsets: S_{train} and S_{test}, where S_{train} comprises 80% of the malicious examples and is used during training and filter construction, while S_{test} holds out the remaining 20% for evaluation. Similarly, the benign set is split into N_{train} and N_{test}, used respectively for training the classifier and evaluating the empirical false positive rate. The classifier $f : U \to [0, 1]$ is trained using logistic regression on hashed feature vectors extracted from the URLs using a fixed-dimension hashing vectorizer. The output of the classifier is interpreted as a probability score, and a threshold parameter $\tau \in [0, 1]$ is used to determine the classification decision. An element $x \in U$ is classified as a positive if $f(x) \geq \tau$, and otherwise passed to a backup filter B which stores all $x \in S_{\text{train}}$ such that $f(x) < \tau$.

For a given threshold τ, the LBF guarantees no false negatives by construction. To analyze its performance, we evaluate its false positive rate on inputs $x \sim \mathcal{D}$, where $\mathcal{D}$ is approximated empirically by drawing uniformly from $N_{\text{test}} \cup S_{\text{test}}$. The total false positive rate of the LBF is estimated by computing the proportion of such negative queries for which either $f(x) \geq \tau$ or $f(x) < \tau$ and $x \in B$. This empirical estimate is compared against the theoretical expected false positive rate $\alpha + (1 - \alpha)\beta$, where $\alpha = \Pr[f(x) \geq \tau]$ is the acceptance probability of the classifier on negatives, and β is the false positive rate of the backup filter on rejected negatives.

To determine a suitable value of the threshold parameter τ, we perform a grid search over a range of candidate thresholds. For each value of τ, we compute the empirical false positive rate and backup filter size over multiple independent trials using randomly sampled query sets. Because this process is stochastic—due to both classifier variability and sampling randomness—we repeat each experiment across several trials to estimate the mean and standard deviation of each metric.

To justify using a relatively small number of trials per threshold (e.g., 5 to 10), we appeal to Chebyshev's inequality. Let X denote the observed false positive rate for a fixed τ, viewed as a random variable over the randomness in sampling. Then for any $\delta > 0$, the inequality states that:

$$\Pr\left[|X - \mathbb{E}[X]| \geq \delta\right] \leq \frac{\text{Var}(X)}{\delta^2}.$$

In our experiments, we find that the standard deviation of the false positive rate across trials is consistently small—typically less than 0.01—which implies that the probability of a large deviation from the empirical mean is correspondingly small. Thus, the average over a modest number of trials is a reliable estimator of the expected performance, and this allows us to efficiently identify a value of τ that minimizes the false positive rate while controlling filter size.

In addition to the baseline LBF structure, we evaluate two key extensions: first, we replace the standard Bloom filter with a Cuckoo filter to enable deletions and finer memory management; second, we incorporate an LRU cache that stores recent positive decisions and coordinates evictions from the Cuckoo filter. These enhancements are evaluated under simulated repeated query patterns, and their impact on accuracy, false positive rate, and memory footprint is compared to the theoretical model.

Projection Hashing LBF (PHBF)

This set of experiments is inspired by work previously done by Bhattacharya et al [4]. Unlike previous approaches for building Learned Bloom Filters where the emphasis was on training a classification model and adjusting the acceptable τ, this paper introduced the Projection Hashing scheme that aims to replace the hash functions in a standard bloom filter with ones that can efficiently discriminate between $x \in U$ and $y \in S$. This section aims to delineate the process of building such a bloom filter.

Just like the standard bloom filter, PHBF also consists of k hashing functions and a bit array B of size m. PHBF divides B into k segments of length $\delta = m/k$. The goal of the training phase is to find k vectors that when used as a basis for projecting a query vector can adequately avoid collisions between U and S. To do this, we create a sample set of random unit vectors with the same dimension as the queries, we iterate over these projecting U and S, count the number of cross-set collisions for each of these vectors, and picking the k vectors that had the least amount of collisions. The algorithm for selecting a vector is defined here.

Once we select k vectors, we place each $x \in U$ to B using:

$$h(x, v) = \lfloor \tfrac{|\langle x, v \rangle|}{\|x\|} \cdot \delta \rfloor$$
$$B[(i-1)\delta + h(x, v_i)] = 1$$

Algorithm 1: SelectVector Subroutine

Input: X_p, Y_p, d, s
Output: Set of projection vectors *vecs*

$c \leftarrow [\,]$;
for $i \leftarrow 1$ **to** s **do**
  $v \sim \mathcal{N}(0, I_d)$;
  $v \leftarrow \frac{v}{\|v\|}$; `// Normalize the vector`
  $X_h \leftarrow \lfloor (X_p \cdot v) \cdot \delta \rfloor$;
  $Y_h \leftarrow \lfloor (Y_p \cdot v) \cdot \delta \rfloor$;
  $c \leftarrow |\text{set}(X_h) \cap \text{set}(Y_h)|$;
  Append (c, v) to collisions;
Sort collisions by increasing c;
return *vecs* $\leftarrow$ first k vectors from sorted collisions;

Unlike the LBFs we look at elsewhere in this paper, PHBF guarantees a 0 false negative rate, removing the need for backup bloom filters and adjusting τ. In that sense, this scheme works as a drop-in replacement for the standard bloom filter. The aim of this scheme is to find vectors that adequately encourage collisions within the sets U and S, and discourage cross-collisions, which would allow you to produce a lower false positive rate for the same capacity and bit array size.

The upper bound for FPR (ε) when using $\delta = k = 1$ is given by [4]:

$$\varepsilon \leq \frac{4nd(\sigma_X^2 - \sigma_Y^2)}{l^2 cos^2(\theta)},$$

Where

$n = |U|$

$d = Dimension(x_i)$

$l = |\mu_X - \mu_Y|$

$\theta =$ Angle between v_o and l

Time complexity for construction: $\Theta((s+1)ndk)$

Querying: $\Theta(dk)$

The motivation behind this method is similar to that seen in Locality Sensitive Hashing, however in PHBF instead of selecting vectors based on number of collisions between similar keys, we promote vectors that reduce the amount of cross-set collisions.

We aim to run experiments that can help us reason about these areas that were previously not covered in the original literature:

- Tuning k for a PHBF, and the effect of k on variance.
- Improving training time through subsampling.

Methodology

In this section, we formally describe the design and rationale behind each variant of the filter architecture evaluated in this study. All methods share a common structure: a classifier $f : U \to [0, 1]$ trained to distinguish keys from non-keys, and a threshold parameter τ that governs whether a query is handled directly or passed to a backup filter. The differences lie in the design of the backup component and the addition of mechanisms to manage repeated queries or memory constraints.

Baseline Learned Bloom Filter (LBF)

The baseline LBF uses a Bloom filter as the backup structure. The classifier f is trained on a labeled set of malicious (positive) and benign (negative) URLs, using hashed feature representations. For a fixed threshold τ, the backup Bloom filter is constructed from those positive training examples $x \in S_{\text{train}}$ for which $f(x) < \tau$. Membership queries are handled in two stages: if $f(x) \geq \tau$, the filter accepts x; otherwise, x is passed to the Bloom filter. The filter guarantees no false negatives by construction.

This method is evaluated by sweeping over values of τ and recording the empirical false positive rate, filter size, and accuracy. We compare these empirical quantities against the theoretically predicted false positive rate $\alpha + (1 - \alpha)\beta$, where α is the acceptance rate of the classifier on negatives and β is the false positive rate of the Bloom filter on classifier-rejected negatives.

Learned Cuckoo Filter (LCF)

In this variant, we replace the backup Bloom filter with a Cuckoo filter. Cuckoo filters offer the key advantage of supporting deletions, which enables dynamic memory management and online adaptation to changes in the key set or query distribution. However, in this implementation, we do not employ any deletion or eviction policy. All elements for which the classifier score falls below the threshold τ are inserted into the filter at construction time, and no elements are ever removed.

As a result, the Cuckoo filter behaves effectively as a static structure, similar in purpose to the Bloom filter but with higher overhead due to fingerprint storage and multiple bucket management. In the absence of deletions, we observe that the Cuckoo filter is strictly less space-efficient than the Bloom filter for equivalent false positive performance. Thus, while Cuckoo filters are theoretically appealing in dynamic settings, they provide no advantage in the static case unless deletions are explicitly utilized. This motivates

our final variant, where deletions are coordinated via an LRU cache.

Learned Cuckoo Filter with LRU Cache (LCF-LRU)

We further augment the LCF with an LRU (Least Recently Used) cache to exploit temporal locality in queries. The cache stores the outcomes of recent queries and, upon eviction, also removes the corresponding entry from the Cuckoo filter. This ensures that the backup filter contains only elements that have been recently queried and classified as false negatives by the model.

The classifier, thresholding, and Cuckoo filter logic remain unchanged. The LRU cache acts as a front-end to the Cuckoo filter and allows the system to forget stale or rarely accessed false negatives. This mechanism is motivated by the observation that real-world query workloads often exhibit strong locality — for instance, repeated accesses to the same URLs or domains. We monitor both the cache hit rate and the effective size of the Cuckoo filter over time, in addition to the standard metrics.

Expected False Positive Rate Estimator (Theoretical Bound)

In all methods, we also compute the theoretical expected false positive rate given by the model:

$$\Pr[\text{LBF}(x) = 1 \mid x \sim \mathcal{D}] = \alpha + (1 - \alpha)\beta.$$

Here, α is computed empirically by evaluating $f(x) \geq \tau$ on a held-out set of negatives (including benign URLs and unseen malicious URLs), and β is the false positive rate of the backup structure when queried only on classifier-rejected negatives. This estimate provides a principled benchmark against which we compare the empirical FPR of each method. Across all variants, we find that the variance of false positive rates across trials is small, justifying the use of a limited number of repetitions and allowing for reliable selection of the threshold τ through empirical grid search.

Projection Hashing Bloom Filter

We design the following experiments to compare the performance of PHBF with a standard bloom filter, and separately test for the reliability of training such a data structure.

For the first, we compare the FPR provided $n = |U|$ and m. For the second, we track the first order statistics of FPR. We would want $\varepsilon_{PH} < \varepsilon_{BF}$, and for an optimal k, the variance and the range to be reasonable.

We also note that due to our vector selection method being completely random, two PHBF trained on the same data with the same hyperparameters might produce very different results. To counter this, unless otherwise stated the FPR of a PHBF is the median of the FPR of 20 independently trained PHBF.

Determining k

For the standard bloom filter, the optimal value of $k = \frac{m}{n} log(2)$, where $n = |U|$.

The original paper derives an $O(ln(1/\varepsilon))$ on k, however this is not a sufficient to run empirical experiments. The paper uses grid search to determine the optimal value of k, and posits that for all practical purposes setting $\delta = 32$ provided them with best results across multiple datasets.

We aim to run a few experiments to track the performance of ε as we iterate over k for multiple samples of PHBF with fixed m and n. Apart from just the best case models, we also aim to track the variance and range of ε, to determine if an adequate value of k can help to make the model less probabilistic.

Subsampling

Given the use of data-aware hash functions, we aim to test if we can reduce the dependence on n while training our projection vectors.

If our projection vectors are able to adequately capture features of x and y, it's a reasonable assumption that it might perform well on previously unseen data given that the distribution of this data is similar to the one that the PHBF was trained on. Conversely, to differentiate between U and S, we aim to check whether we can get reasonable performance if we can instead train our vectors on a subsample of n.

To achieve this, we run a PHBF trained on $|n|$, $\sqrt{|n|}$, and $log(|n|)$ samples. We track the performance of these models as we vary m, comparing it with the ideal FPR for a standard bloom filter.

The only change in the training process involves selecting a random subsample of U and S when running our SelectVectors routine, the FNR for each of these is still set to 0 by ensuring that all $x \in U$ are added to B.

Low-Rank Adaptation (LoRA)

For our alternative approach, we implement a Low-Rank Adaptation (LoRA) technique to fine-tune a pre-trained neural network model. LoRA was introduced by Hu et al. **[hu2021lora]** as a parameter-efficient fine-tuning method that adds low-rank decomposition matrices to each layer of a transformer. In our implementation, we adapt this concept to a simple MLP architecture for URL classification.

The approach consists of two steps:

1. First, we train a base MLP model (backbone) on the full training dataset
2. Then, we fine-tune only a set of low-rank adapter modules while keeping the original backbone frozen

This allows us to adapt the model to specific error patterns while keeping the overall parameter count low. The adapter modules use a low-rank decomposition defined as:

$$h_{adapted} = h + \alpha \cdot BA(h) \tag{1}$$

where h is the original activation, B and A are low-rank matrices, and α is a scaling factor.

We implemented LoRA approach in two different settings:

Standard LBF and **Sandwich LBF**, with preliminary focus on Standard LBF and final comprehensive analysis on both.

Results

In this section, we present empirical results comparing the false positive rate, backup filter size, accuracy, and cache hit rates across the different methods described in the methodology section. For each filter configuration, we sweep over a range of classifier thresholds $\tau \in [0.5, 0.99]$, and repeat each experiment across 10 randomized trials. At each threshold, we report both the empirical false positive rate and the expected rate predicted by the analytical model:

$$\alpha + (1 - \alpha)\beta$$

where $\alpha = \Pr_{y \sim \mathcal{D}}(f(y) \geq \tau)$ is the classifier acceptance rate and $\beta = \mathbb{E}[F(B)]$ is the false positive rate of the backup filter.

Across all settings, we find that the empirical false positive rates closely match the predicted rates from the model. Variance across trials is small, supporting our use of Chebyshev's inequality to justify a small number of repetitions per configuration.

LBF

Figure 1 summarizes results for the standard LBF without caching or deletions. Increasing τ reduces the load on the backup Bloom filter, which in turn lowers the false positive rate. However, this also increases the number of classifier false negatives that must be stored, leading to a larger Bloom filter size.

Accuracy steadily increases with τ, peaking near 0.90, while false negative rate remains zero due to the two-stage design of the filter. Figure 2 confirms that FPR, accuracy, and backup size are stable across different sample sizes, with low trial-to-trial variance.

LBF with Cache

To evaluate the benefit of query-locality-aware caching, we augment the LBF with a fixed-size Least Recently Used (LRU) cache in front of the classifier. Repeated queries to popular URLs are expected to be resolved from cache, reducing both classifier and backup filter load. We simulate cache locality by repeating query sets with a fixed repeat factor.

Figure 3 shows the false positive rate, expected FPR, and accuracy across a sweep of τ values and different sample sizes. Despite repeated queries and small cache sizes (10k), the cache provides a measurable reduction in backup accesses.

Compared to the standard LBF, the cache-augmented system achieves similar or slightly better accuracy and maintains a consistent false negative rate of zero. Importantly, we observe that the false positive rate remains stable or slightly decreases, particularly at higher τ, due to the cache shielding both the classifier and backup filter from redundant queries. This validates our hypothesis that even simple caching can improve filtering efficiency in workloads with temporal locality.

Figure 6 demonstrates that these trends hold across all sample sizes tested, and standard deviation across trials is minimal. This stability supports our use of relatively few trials for each setting and further confirms the robustness of the LBF+Cache system under varied workloads.

LBF with Cuckoo Backup (LCF)

We next evaluate the performance of the learned filter using a Cuckoo filter as the backup structure in place of the traditional Bloom filter. Unlike Bloom filters, Cuckoo filters natively support deletions and dynamic resizing. However, in this configuration, we do not enable deletion or eviction policies, treating the Cuckoo filter as a fixed-size structure filled only with the classifier's false negatives.

Figure 5 shows the results of the LCF method across ten trials and varying sample sizes. We observe that false positive rates, while still improving with increasing τ, are consistently higher than the standard LBF or cache-enhanced versions at comparable backup sizes. This is partly due to the Cuckoo filter's need to store a fingerprint and additional overhead per key.

Across all thresholds, the accuracy and false negative rate remain stable and nearly identical to those of the LBF baseline, as shown in Figure 5 (right). However, the backup structure's size remains fixed and grows linearly with the number of false negatives—no deletions or optimizations are applied.

This result illustrates a key drawback of using

Cuckoo filters in a static configuration. Although they provide theoretical advantages in terms of deletion and space reuse, these are not realized in our unoptimized setting. As a result, the LCF performs slightly worse in space and empirical FPR compared to the LBF with cache. These results justify incorporating eviction policies (e.g., LRU) when deploying Cuckoo filters in learned settings.

LCF with LRU Eviction (LCF-LRU)

To realize the benefits of the Cuckoo filter's support for deletions, we incorporate an LRU eviction mechanism into our LCF design. Each time a key is evicted from the LRU cache, it is simultaneously removed from the Cuckoo filter. This ensures that stale false negatives do not accumulate over time, bounding the size of the backup structure and improving adaptability to dynamic workloads.

Figures 7 and 8 summarize the performance of this configuration. The false positive rate decreases steadily with increasing τ, reaching a minimum near $\tau = 0.99$, consistent with expectations. Accuracy increases correspondingly, peaking at nearly 90%. The expected FPR closely tracks the empirical FPR across thresholds, and both metrics exhibit low variance across trials.

These results demonstrate that the LCF-LRU is the most space-efficient and accurate configuration among those studied, particularly under query locality and repeated access patterns. Its ability to support deletions makes it well-suited for streaming or adaptive settings where data evolves over time.

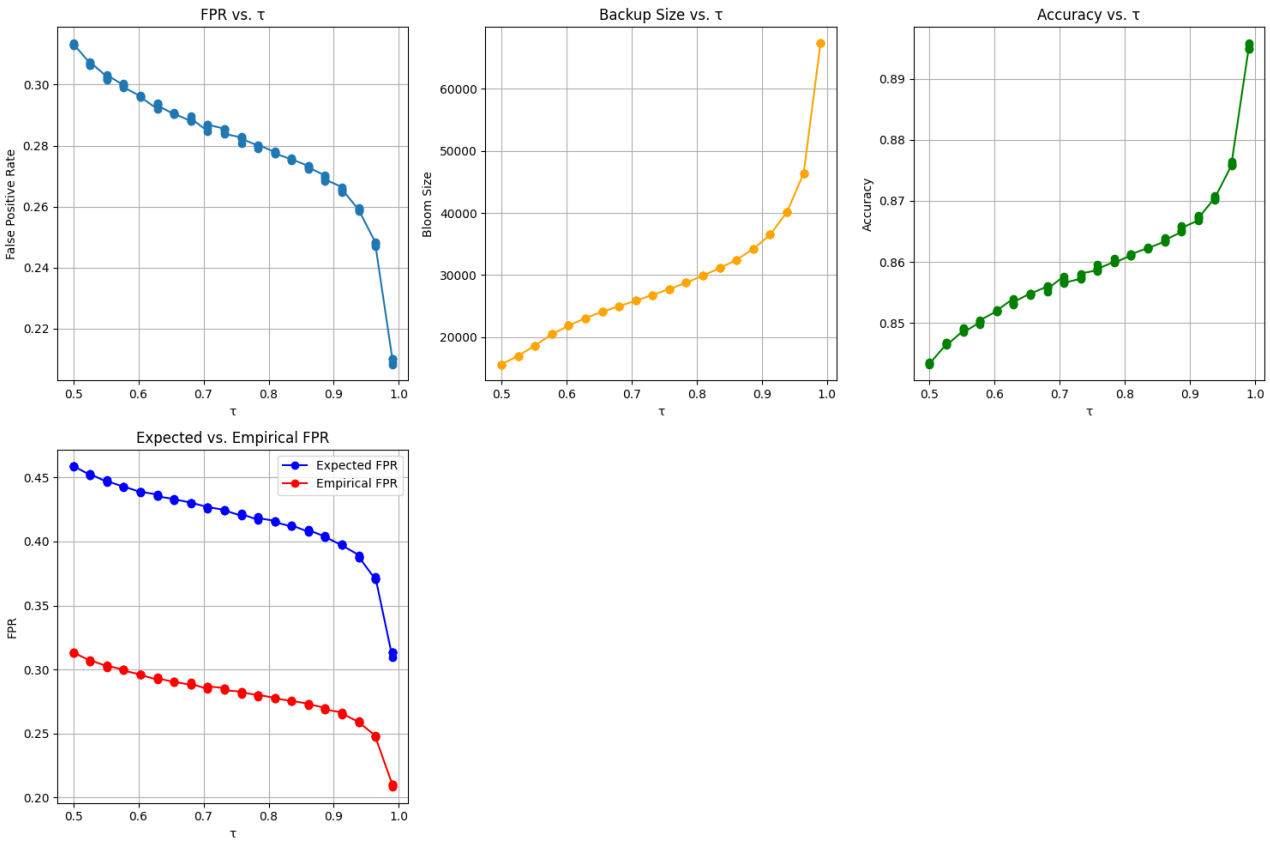

Figure 1: *Summary of baseline LBF performance across thresholds τ. Top: False positive rate, Bloom backup size, and accuracy as a function of τ. Bottom: Empirical vs. Expected false positive rates.*

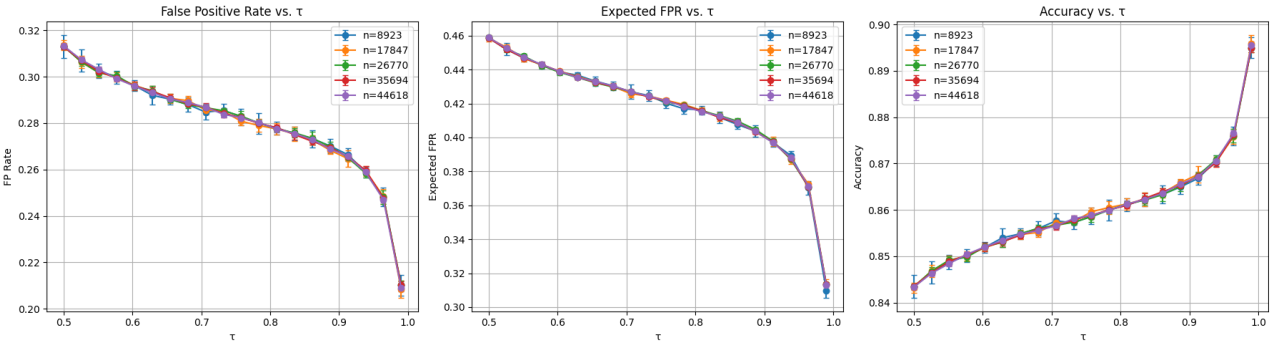

Figure 2: *Summary of baseline LBF performance across thresholds τ. Top: False positive rate, Bloom backup size, and accuracy as a function of τ. Bottom: Empirical vs. Expected false positive rates, validating the approximation $\alpha + (1 - \alpha)\beta$.*

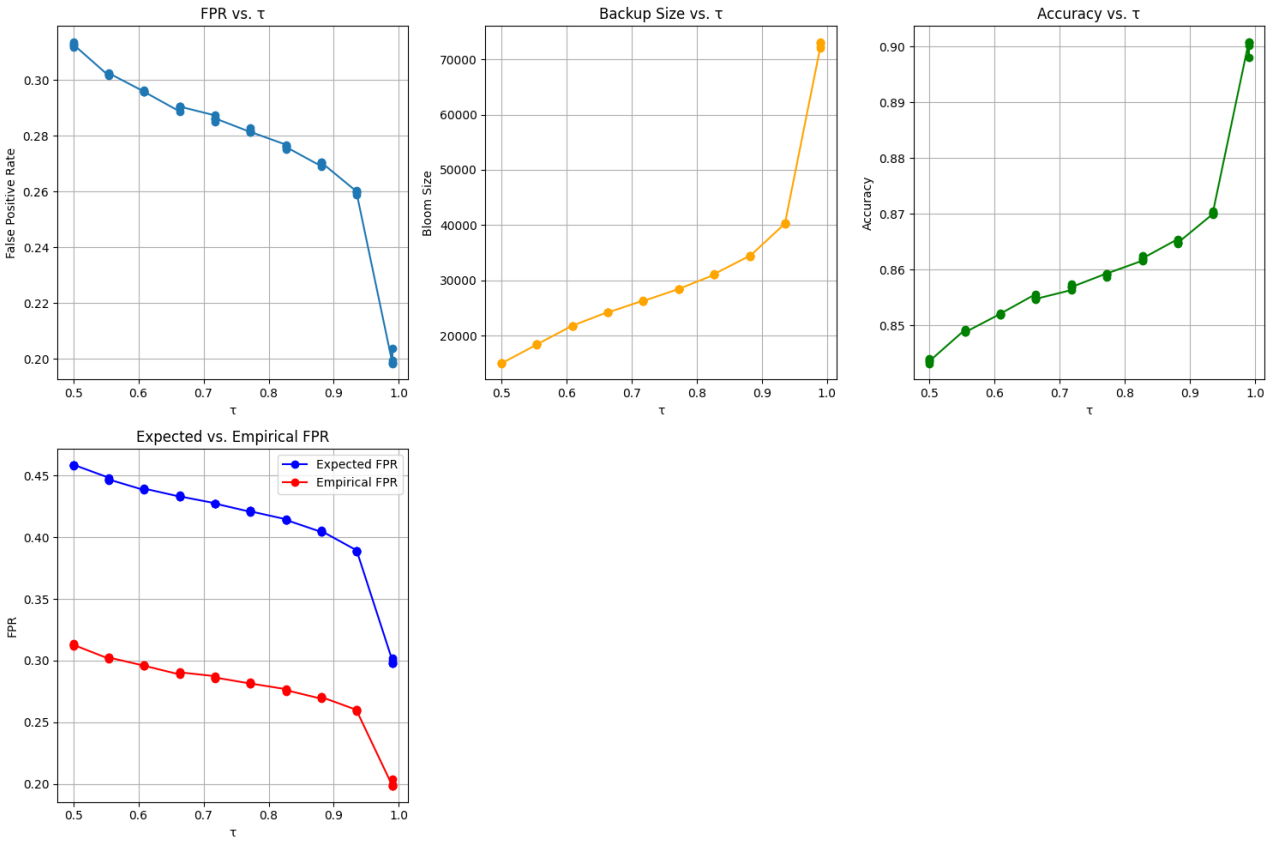

Figure 3: *Performance of LBF with Cache across thresholds τ. Top: Empirical FPR, backup Bloom size, and accuracy as a function of τ. Bottom: Comparison of expected and empirical FPR.*

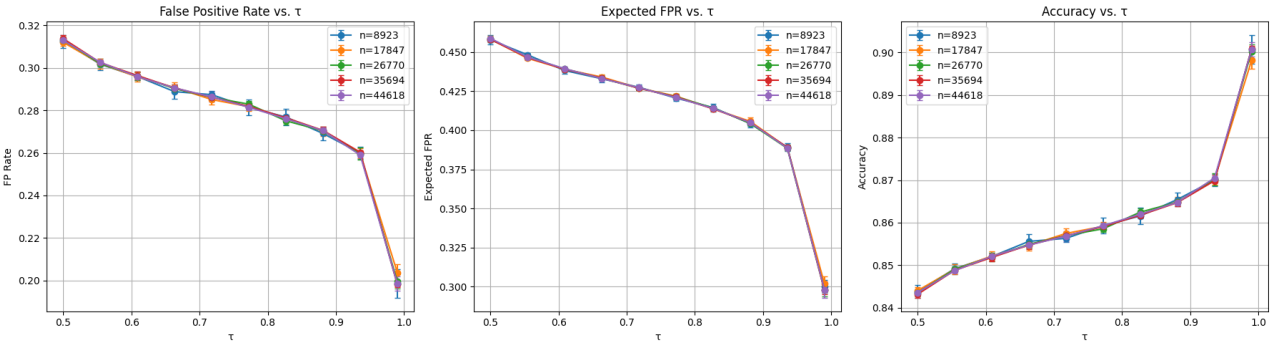

Figure 4: *Sample size variance analysis for the LBF with Cache. Each line corresponds to a different sample size. Error bars represent standard deviation across 10 trials.*

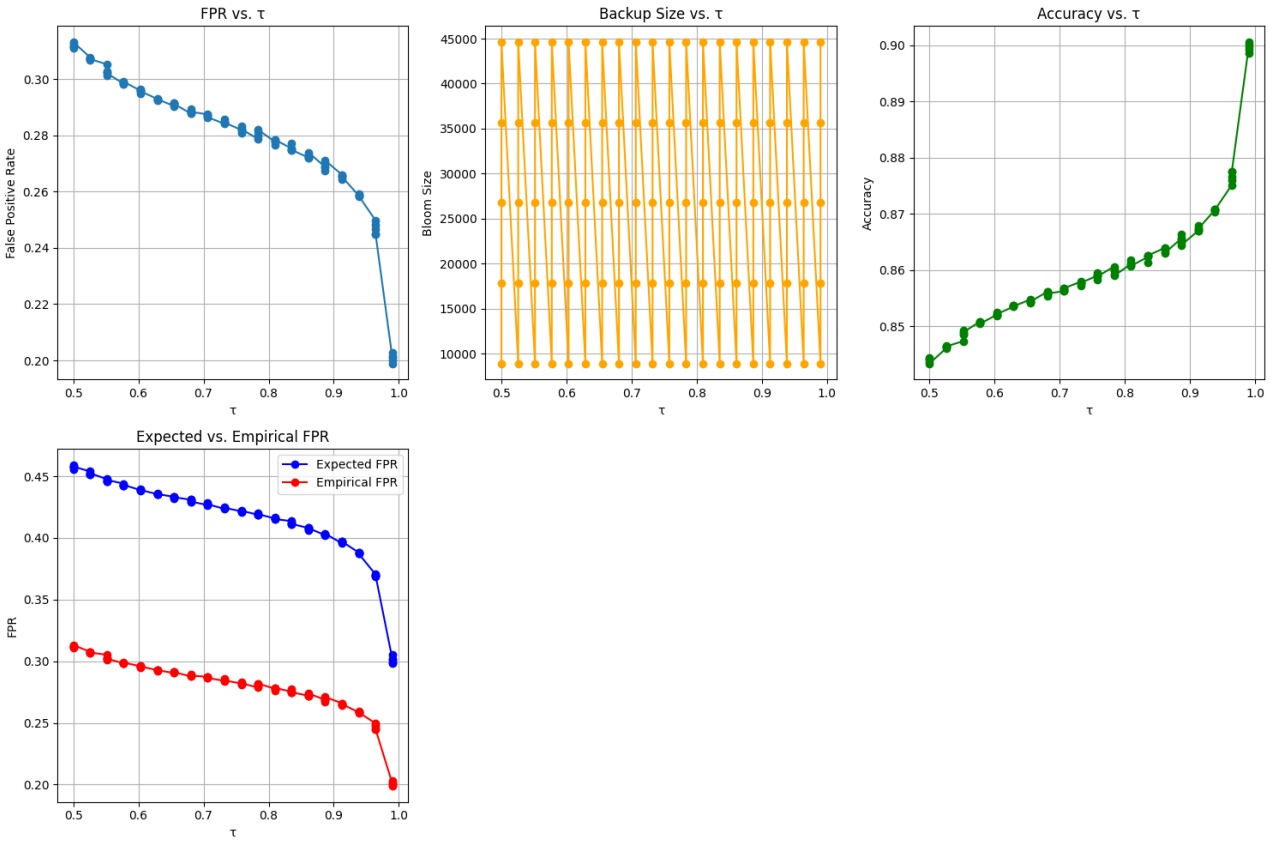

Figure 5: *Performance of LBF with Cuckoo Backup (LCF) across thresholds τ. Top: Empirical FPR, Cuckoo size (equal to number of stored FNs), and accuracy. Bottom: Comparison of expected and empirical FPR.*

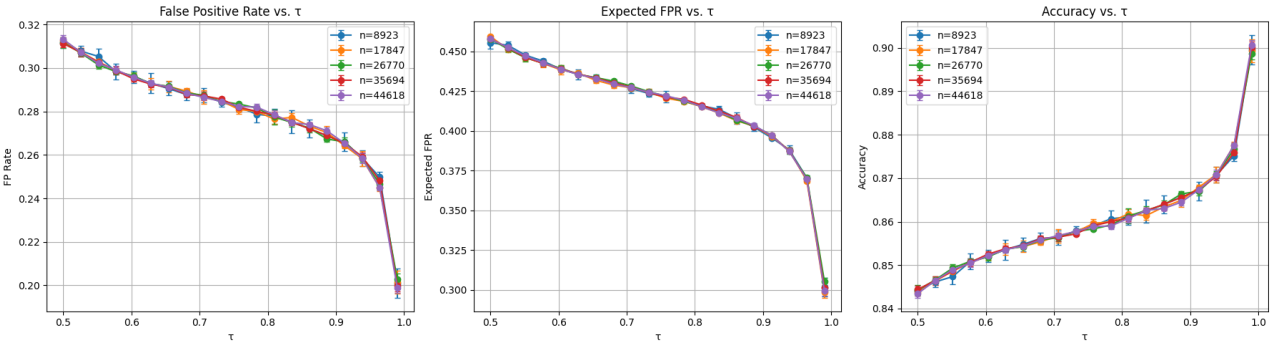

Figure 6: *Sample size variance analysis for the LBF with Cuckoo. Each curve shows a different sample size; error bars reflect standard deviation over 10 trials.*

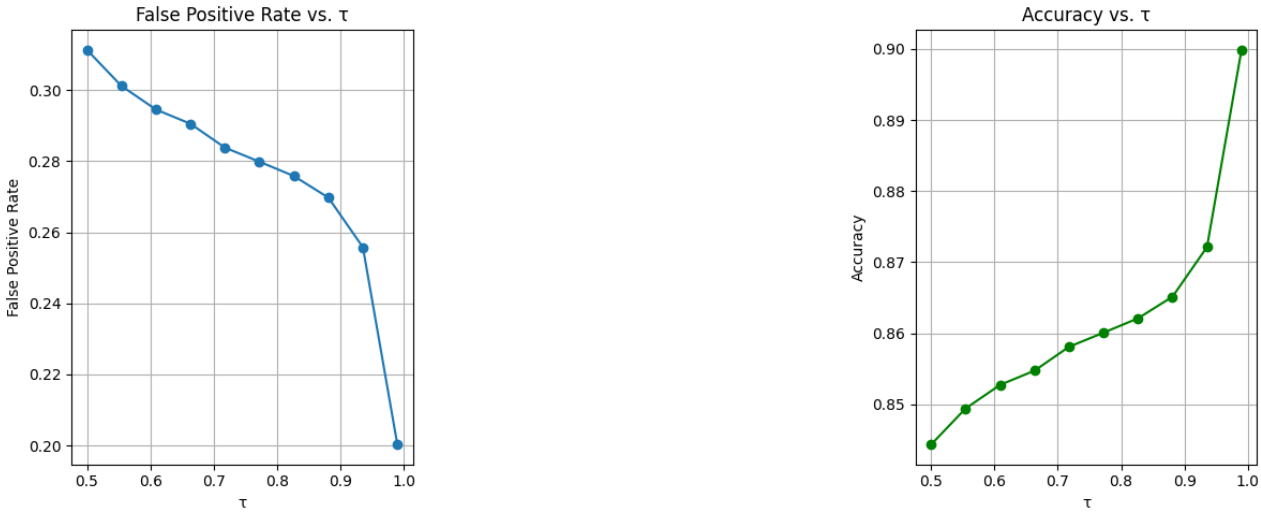

Figure 7: *LCF-LRU: False Positive Rate and Accuracy as a function of τ. The configuration uses Cuckoo filters with LRU eviction to discard stale false negatives.*

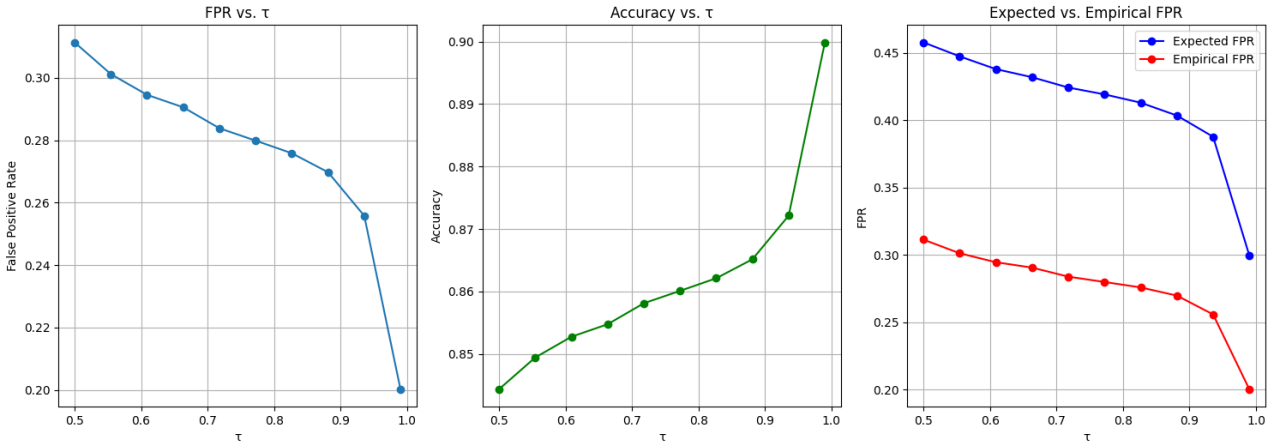

Figure 8: *LCF-LRU: Comparison of expected and empirical false positive rates across values of τ. The close alignment confirms that the analytical FPR estimate holds under eviction.*

PHBF Experiments

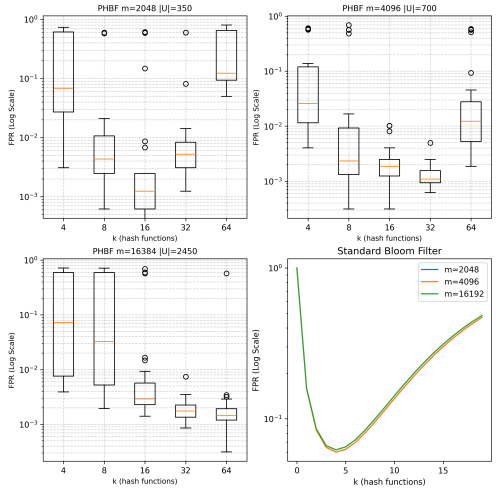

Figure 9: *False positive rate vs. number of hash functions for PHBF and standard Bloom filter. We maintain m/n to be constant in each case. Unlike the standard bloom filter, the optimal k is not directly linked with this ratio.*

Bitsize	Optimal k	δ
2048	16	128
4096	32	128
16384	64	256

Table 1: *Optimal number of hash functions and corresponding $\delta = m/k$ for different bit sizes.*

For plotting the dependence of FPR on k, we ran the following experiment several times, and have included results from a representative run. We trained a set of 40 PHBFs for each $k_i \in \{4, 8, 16, 32\}$, and repeated it with varying bitsizes (2048, 4096, 16384).

In each case, the ratio $m/|U|$ was kept constant, this was done to kepe the effect of k on FPR representative. Since for a standard bloom filter $k_{optimal} = \frac{m}{n} ln(2)$, the fixing $m/|U|$ is a natural choice.

From Figure 9, Table 1, and Table 2 we note the following:

- Similar to the standard bloom filter, ε appears to have a non-monotonic and convex dependence on k. This is an experimental confirmation of the relation between ε and $k > 1$ derived in [4]. Given that this dependence is hard to calculate for real-life datasets, we require empirical tuning of k.
- As we aproach $k_{optimal}$ (the minimizer for the median ε), we note a decrease in variance for our FPR values.
- We see $\delta_{optimal} = 128$ $(m/k_{optimal})$ to be a reasonable estimate for our dataset, across bitsizes. This concurs with the original paper which posited that $\delta = C(= 32)$ provided optimum results across different m and datasets. A more rigorous justification for this is still required.

These findings are consistent with our initial knowledge of the structure of a PHBF. Increasing k introduces more perspectives into how our a given key must be hashed, however it reduces δ, the resolution of each of those hash functions. It is therefore necessary to deal with this tradeoff.

Developing a formal model for variance was beyond the scope of this project. Through empirical evidence we note that the variance in FPR decreases as $k \to k_{optimum}$. Due to the lack of a proper way to formulate bounds for k or δ, it is necessary to train models with multiple configurations, which significantly increases the overhead of PHBF construction.

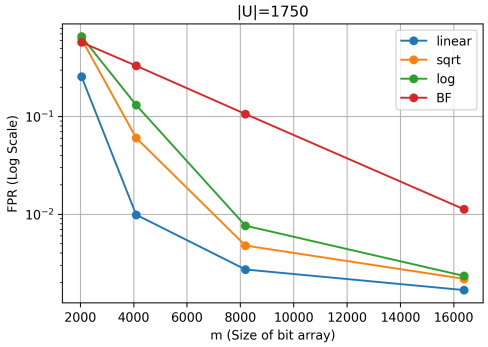

Figure 10: *Comparing performance of various bloom filters. Each dot represents the median value of FPR for 40 PHBF trained on the same parameters.*

For subsampling, we wanted to note the effect of using fewer samples for selecting vector projects as we increase the size of B. First, we chart the guarantee provided by a standard bloom filter using $\varepsilon = (1 - e^{-k_{optimal} \cdot n/m})^{k_{optimal}}$. Using the previous experiment, we determined $k_{optimal}$ for PHBF of various sizes. We tracked the FPR for $m \in \{2048, 4086, 8196, 16384\}$, each for a fixed $|U| = 1750$. The FPR for each (sampling method, size) is the median FPR from a set of 40 runs to keep the results representative. The sampling methods were training on all $|U + S|$ (as a baseline), $\sqrt{|U + S|}$ samples, and $\log(|U + S|)$ samples.

Figure 10 can be used to compare the performance of our PBHF with the standard bloom filter and the effect on performance when using subsampling. With the caveat that our results are highly dependent on the dataset, we note the following:

- Across all configurations, the PHBF variants outperform the standard bloom filter in terms of FPR.

- Performance improves with increased training data, suggesting that our SelectVector subroutine generalizes well and is not prone to overfitting - unlike typical linear classifier models.
- The relative impact of subsampling on FPR diminishes as the size of thebita rray B increases. This indicates a trade-off between memory usage and construction time for a PHBF, and therefore this can be tuned based on the intended usage.

From these results, we can note that subsampling might be a powerful tool to speed up the process of constructing a PHBF. This can help bridge the gap between the practical considerations of using a PHBF as a drop-in replacement for a standard bloom filter. The performance of subsampled PHBFs indicates a scope for using them for dynamic datasets, given that it is robust against unseen queries.

k	m	Median FPR	Var(FPR)
4	16384	0.0725	0.0921
8	16384	0.0325	0.0947
16	16384	0.0029	0.0355
32	16384	0.0018	0.000001
64	16384	0.0014	0.0109

Table 2: *PHBF false positive behavior across varying hash counts (k) at 16,384-bit array size. Runs=40*

LBF along with LoRA

In this section, we present empirical results comparing the false positive rate, backup filter size, accuracy, and cache hit rates across the different methods described in the methodology section. For each filter configuration, we sweep over a range of classifier thresholds $\tau \in [0.5, 0.99]$, and repeat each experiment across 10 randomized trials.

Backbone and Adapter Training We first train a backbone model once and then adapt it multiple times for different trials on **malicious url detection** dataset.

LoRA vs. Baseline Performance Analysis

Figure 11 presents a comprehensive comparison between the backbone-only model and the LoRA-adapted model across different threshold values τ. The plots reveal several key insights:

False Positive Rate and Filter Size Trade-offs As shown in Figures 12, 13 the false positive rate decreases for both models as τ increases. The backbone-only model consistently maintains a lower FPR across all thresholds. However, this comes at a significant cost in backup filter size, where the LoRA-adapted

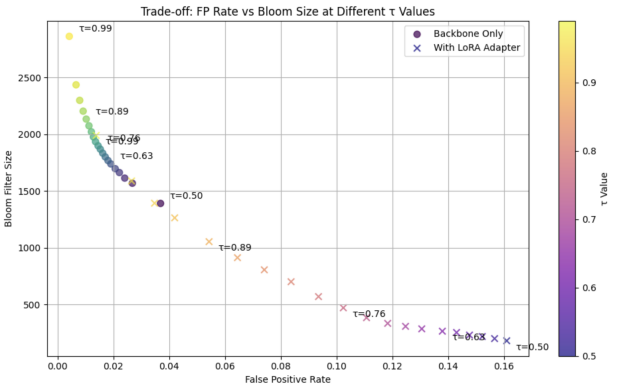

Figure 11: *Performance comparison of backbone-only model vs. LoRA-adapted model across different threshold values. The LoRA-adapted model consistently achieves better trade-offs, requiring smaller backup filter sizes for comparable false positive rates.*

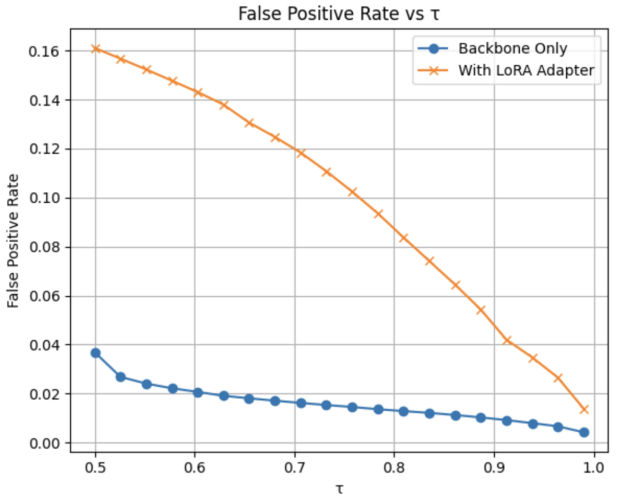

Figure 12: *False positive rate decreases for both models as τ increases, with backbone maintaining lower FPR.*

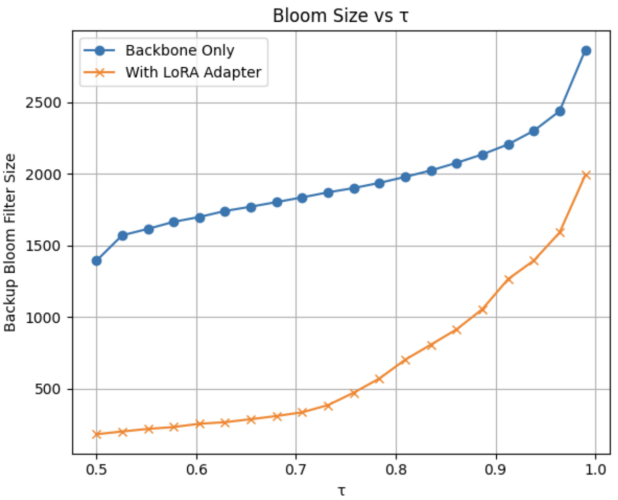

Figure 13: *Backup Bloom filter size increases with τ, but LoRA requires significantly less backup storage.*

model demonstrates a substantial advantage, requir-

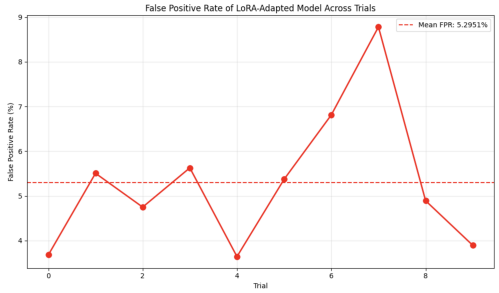

Figure 14: *Error rates of LoRA-adapted model across 10 trials. False positive rate across trials with mean FPR of 5.30%. While some variance exists between trials, the overall performance remains consistent.*

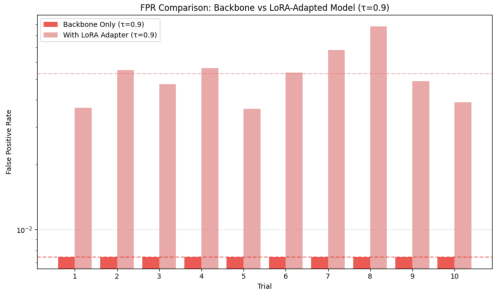

Figure 15: *Comparison of error rates between backbone and LoRA-adapted models at $\tau = 0.9$. FPR comparison on logarithmic scale showing LoRA's higher but consistent false positive rates.*

ing much smaller backup storage.

Figure 11 further illustrates this trade-off by directly plotting false positive rate against Bloom filter size for different τ values. The LoRA-adapted model consistently achieves more favorable trade-offs, requiring significantly smaller backup filters for comparable false positive rates. For instance, at $\tau = 0.89$, the LoRA model uses less than half the backup storage of the backbone model while maintaining a reasonable FPR.

Trial-to-Trial Variation Analysis

We evaluated the stability of both approaches across multiple trials to assess the robustness of our models. Figures 14 and 15 illustrate the variation in performance metrics across different trials.

The mean false positive rate across trials for the LoRA-adapted model is 5.30%, with some variance between individual runs. Similarly, the mean false negative rate is 5.32%. While individual trials show some variability, the performance remains relatively stable, with most trials falling within 1-2 percentage points of the mean.

Figure 16 provides a scatter plot visualization of each trial's performance relative to the backbone model. The plot confirms that all LoRA trials achieve better true positive rates (1 - FNR) than the backbone

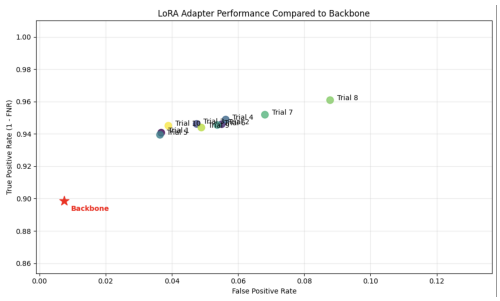

Figure 16: *LoRA adapter performance compared to backbone across all trials. Each point represents a different trial, with the backbone model shown as a red star. The LoRA-adapted model consistently achieves higher true positive rates (1 - FNR) compared to the backbone, with varying false positive rates.*

model, though at the cost of slightly higher false positive rates.

Sandwich LBF Results

The Sandwich LBF approach introduces a lower threshold $\tau_{negative}$ and only queries the backup filter for elements whose score falls in the uncertain range $[\tau_{negative}, \tau_{positive}]$. This approach significantly reduces the false positive rate to just 0.77%, a substantial improvement over the baseline. The precision increases to 98.36%, though with a slight trade-off in recall (88.46%). Overall accuracy improves to 95.54%.

LoRA Results

The LoRA-based approach achieves the best overall performance with an accuracy of 95.87%, outperforming both LBF variants. It also achieves the lowest false positive rate of 0.64%, with excellent precision (98.65%) and slightly improved recall (89.15%) compared to the Sandwich LBF. The F1 score of 93.66% is the highest among all approaches.

However, the LoRA approach has notably higher training time (265.31 seconds vs 125.78 for standard LBF), although inference times are comparable across all methods. The model size is also slightly larger, especially in non-quantized format.

Performance Comparison

Figure 17 and Table 3 present a comprehensive comparison of all approaches.

Conclusion and Future Work

In this work, we implemented and evaluated several variants of the Learned Bloom Filter, with the goal of reducing false positives while minimizing

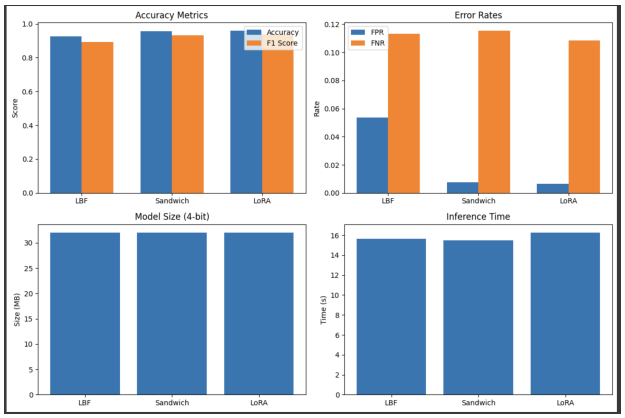

Figure 17: *Comprehensive comparison of LBF, Sandwich LBF, and LoRA approaches. Top left: Accuracy metrics showing LoRA's superior performance. Top right: Error rates highlighting the significant FPR advantage of Sandwich and LoRA approaches. Bottom left: Model size (4-bit quantized) comparison showing similar memory requirements. Bottom right: Inference time comparison showing minimal differences.*

Table 3: *Performance Comparison of Different Approaches*

Metric	LBF	SLBF	LoRA
Accuracy	0.9259	0.9554	**0.9587**
Precision	0.8960	0.9836	**0.9865**
Recall	0.8867	0.8846	**0.8915**
F1 Score	0.8913	0.9315	**0.9366**
FPR	0.0537	0.0077	**0.0064**
FNR	0.1133	0.1154	**0.1085**
Train time (s)	**125.78**	124.55	265.31
Inference time (s)	**15.6484**	15.4965	16.2886
Size (MB)	**256.2715**	256.2697	256.2822
Size 4-bit (MB)	32.0519	32.0501	**32.0353**
Bloom filter (KB)	20.99	**19.15**	N/A

***SLBF: Sandwich LBF*

memory usage. We demonstrated that incorporating a learned classifier into the filtering pipeline can substantially reduce the size of the backup structure without sacrificing correctness. We further explored two enhancements: replacing the Bloom filter with a Cuckoo filter to enable deletions, and augmenting the structure with an LRU cache to exploit query locality. Empirical results confirm that both modifications improve performance under realistic conditions.

Our analysis also validated the theoretical model of expected false positive rate, showing strong alignment between analytical predictions and empirical observations. By using Chebyshev's inequality to bound deviation across trials, we were able to perform efficient grid search over thresholds without requiring excessive replication.

Our experiments with Projection Hashing demonstrate that this approach can significantly reduce filter size while achieving comparable false positive rates. The primary computational cost lies in constructing an efficient filter, particularly during vector selection. However, the model generalizes well to unseen data, and subsampling can reduce dependence on the full dataset $|U \cup S|$. Exploratory data analysis, such as applying PCA, can further aid in tuning the PHBF by simplifying the input space.

Unlike standard Bloom filters, which offer strict performance guarantees, PHBF requires empirical tuning of hyperparameters to find optimal models. There is also potential to adapt projection hashing to online settings. Although we hypothesize that subsampling could enable faster retraining of projection vectors once a target FPR is exceeded, we leave empirical validation of this idea to future work.

The PHBF and LoRA approaches show the best adaptation to data patterns. PHBF accomplishes this through data-aware hash functions, while LoRA uses low-rank matrices to efficiently capture error patterns. LoRA adaptation consistently improves true positive rates across all trials. The LoRA approach consistently outperformed traditional learned filters in terms of accuracy and false positive rates, though at the cost of increased training time.

Future work may focus on several directions. First, one could replace the fixed logistic regression classifier with more expressive models, such as neural networks or gradient-boosted trees, and study how classifier complexity affects the tradeoff between backup size and prediction quality. Second, adaptive threshold selection strategies could be explored, potentially optimizing τ dynamically based on recent query statistics. Finally, extensions to support dynamic key insertions and deletions in online settings, with adversarial query streams or distributional shift, would further strengthen the applicability of learned filters in practical systems.

Author contributions

- **Codebase setup, LBF baseline, LBF cache, LBF cuckoo, LBF cuckoo + cache - Gordon**
- **Projection Hashing Bloom Filters (Code, Experiments, Charts) - Sanchit**
- **LoRA with LBF, Sandwich LBF (Code, Experiments, Charts) - Subhash**

References

[1] Tim Kraska et al. "The Case for Learned Index Structures". In: *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 2018, pp. 489–504. DOI: 10.1145/

3183713.3196909. URL: https://doi.org/10.1145/3183713.3196909.

[2] Michael Mitzenmacher. "A Model for Learned Bloom Filters and Related Structures". In: *Communications of the ACM* 64.9 (2021), pp. 102–111. DOI: 10.1145/3459895. URL: https://doi.org/10.1145/3459895.

[3] Sid321axn. *Malicious URLs Dataset*. https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset. Accessed: 2025-05-09. 2021. URL: https://www.kaggle.com/datasets/sid321axn/malicious-urls-dataset.

[4] Arindam Bhattacharya et al. "New wine in an old bottle: data-aware hash functions for bloom filters". In: *Proc. VLDB Endow.* 15.9 (May 2022), pp. 1924–1936. ISSN: 2150-8097. DOI: 10.14778/3538598.3538613. URL: https://doi.org/10.14778/3538598.3538613.