Volume: 13 Issue: XI Month of publication: November 2025
DOI: https://doi.org/10.22214/ijraset.2025.75870

International Journal for Research in Applied Science & Engineering Technology (IJRASET)
ISSN: 2321-9653; IC Value: 45.98; SJ Impact Factor: 7.538
Volume 13 Issue XI Nov 2025- Available at www.ijraset.com

To Detect and Prevent Sophisticated Cyber-Attacks in 5G-Enabled Networks using AI Models

Dr. C. Parthasarathy[1], Shivanesh P[2], Vedha Vigneshwar T[3]
[1]*Professor,* [2, 3]*Student, Department of CSE, Rajalakshmi Engineering College, Chennai, India*

Abstract: Approximate membership query (AMQ) structures represented by the Bloom Filter and its variants have been popularly researched in recent years. Researchers have recently combined machine learning with this type of structure to reduce space consumption and computation overhead further and make remarkable progress. However, with the booming performance in space or other metrics, researchers tend to ignore the security of the trained model. The machine learning model is vulnerable to poisoning attacks, and naturally, we infer that the learning-based filters also have the same deficiencies. Hence, in this paper, to confirm the inference mentioned above, experiments on the real-world datasets of URLs are conducted and prove that it is necessary to consider the security issue when using learning-based filters. We show that by data poisoning, the attacker can deflect learned Bloom Filters to make a false identification, which can lead to a significant loss in some cases. Aiming to solve this issue, we put forward a method named Defensive Learned Bloom Filter (DLBF) to diminish the influence of data poisoning and achieve a better performance compared to types of learned Bloom Filters.
Index Terms: Bloom Filter, Learned Bloom Filter, Data poisoning, Security of approximate membership query structure

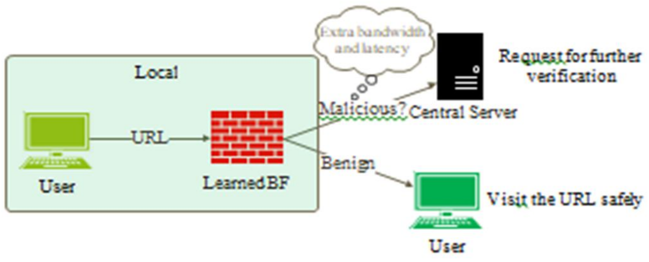
I. INTRODUCTION

Let us look at the Google Chrome use case. When a user tries to visit a website, it should validate whether the URL is safe. To validate the URL, Chrome can call the Google server (internally, Google can maintain any data structure to find this out). However, the challenge with this approach is multifold. For every URL request served on Chrome, the URL validation happens through the Google server, which adds a dependency on Google servers, net- work round-trip time, and the requirement to maintain high availability to validate the URLs for all the URLs fired from Chrome browsers worldwide. To lower the load on the Google server, embedding a URL blacklist into Chrome is an efficient way. Every time a user enters a URL, the browser checks whether this URL is in the blacklist locally and swiftly. Storing the blacklist straightforwardly may take up too much space. For example, if the average length of URL in the blacklist is 32, simply storing 108 malicious URLs will take up ≈ 3 GB, which is not acceptable on some kinds of Internet of Things (IoT) devices.

To address the above challenge, Bloom Filter [1] is typ- ically used. Firstly, Bloom Filter is a space-efficient data structure with a high speed (O(1)) answering the mem- bership query. As a trade-off, the result of a query is not always correct. This probabilistic data structure tells us that an element is either definitely not in the set or possibly in the set. A false positive emerges if an element is not in the set while the Bloom Filter gives an opposite conclusion. Storing the same 108 URLs with 1% false positive rate only takes up ≈ 114 MB, 0.037x compared with storing the original URLs. High space efficiency, fast query response time, and the characteristic of no false negatives make the filters more appropriate for this case compared to the DNN-based methods. Thus, in this paper, we focus on discussing the underlying security issue of using the filters for malicious URL detection rather than the DNN models.

Bloom Filter has achieved great success in both the academy and industry. There are several practical applica- tions of the standard Bloom Filter, such as a cache of proxy in website system [2], IP trace-back for obtaining sources of IP packets [3], virus scanning [4], detection of weak passwords [5] and safe browsing mentioned at the start. Although Bloom Filter achieves a high space efficiency, it has not reached the optimal solution and can be improved. With appropriate parameters, the occupied bit for each element in Bloom Filter is still 44% over the lower bound for probabilistic data structures [6]. Thus, it is essential to reduce space costs further. Recently, Kraska et al. [7] utilize the prior knowledge of the data to train a learned model to help check the membership of an element, which provides a novel method to save space. The framework is named learned Bloom Filter. Since then, several modifications of it [8], [9], [10] were proposed to improve the performance further.

It is also proved that for the task of malicious URL detection, the learned Bloom Filter and its modifications perform better than the standard Bloom Filter [9], [10].



Researchers show that with the same false positive rate, replacing the standard Bloom Filter with the learned Bloom Filter can save 60% space on average. Thus using learned Bloom Filters to represent a blacklist of URLs for safe browsing has plenty of advantages over the standard Bloom Filter. Firstly, the learned Bloom Filter takes up less space under the same conditions as the standard Bloom Filter. Besides, once the filter has been constructed, the standard Bloom Filter can add new malicious URLs while deletion is denied. The result is that the false positive rate increases gradually. Updating the blacklist needs to build the entire filter from scratch. While the learned Bloom Filter could merely update the learned model from the central server and adjust the small backup filter. Finally, owing to the generalization of the learned model, learned Bloom Filters can recognize unknown URLs while the standard Bloom Filter fails. Figure 1 is the outline of using the learned Bloom Filter for safe browsing. Despite the remarkable performance that machine learn- ing achieves in classification, there have been some prob- lems with machine learning security in recent years. For instance, data poisoning attack [11], [12], [13] aims to per- turb the machine learning model to perform poorly in real- world applications. Specifically, it can mislead the classifier and degrade the classifier's accuracy dramatically [14]. The underlying principle is that the data used for training is unreliable, while the attacker can add crafted data items to the training set. By using a poisoning attack, the adversary can control the output of the learned model and thus artificially create a specific false positive. In addition, we found that the learned Bloom Filter and its modifications are fragile to data poisoning attacks, which means a person with an ulterior motive can craft some poisoned examples to contaminate the learned Bloom Filter for false identification. For instance, in Figure 1, suppose the learned Bloom Filter representing a blacklist of URLs has been compromised, causing a specific benign URL with a high frequency of visits to be falsely identified as malicious. Whenever a user tries to visit the URL, a request will be sent to the central server to determine whether it is a false positive. If it belongs to a false positive, the user can visit the URL safely while being blocked if it is a malicious one indeed. Thus, a large quantity of extra communication will be produced if a popular benign URL is recognized as malicious by mistake. Besides, in other potential scenarios using learned Bloom Filters such as spam email filtering [15] and database query accelerating [16], suffering such an attack may also lead to performance degradation or more severe consequences. To avoid that, it is of vital significance for learned Bloom Filters to be capa- ble of defending against data poisoning attacks. However, experiments in Section 3 show that they do not perform well when facing up to poisoning attacks. This illustrates the necessity of proposing a new framework to confront the poisoning attack.
As far as we know, we are the first to utilize poisoning attacks to reveal the vulnerability of the learned Bloom Filter.
Our work is different from that in [17]. Using a learned Bloom Filter to represent a blacklist of URLs, they concentrate on how to find out a new URL that is prone to be recognized as a malicious one by the filter from a benign one. We aim to illustrate that a previously benign URL can be wrongly identified using the learned Bloom Filter's vulnerability to data poisoning attacks. Converting a benign URL to a malicious one is more practical because the benign URL is visited more frequently than a crafted malicious one. That is to say, for instance, people usually visit "google.com" rather than the crafted "google1.com". Based on the analysis above, we put forward a method to diminish the influence of poisoning attacks and a new strategy for optimization to acquire the hyper-parameters to construct the filter. We name our framework as **D**efensive **L**earned **B**loom **F**ilter, which is denoted as DLBF for short in the rest of the paper.
Here, we summarize our main contributions as follows:

1) We illustrate the vulnerability of the learned Bloom Filter and its two types of modifications towards poisoning attacks, which the attacker can utilize for some purpose in practical scenarios. Experiments are carried out in four real-world URL datasets to prove the vulnerability.
2) We propose one method to diminish the influence of poisoning attacks on the learned Bloom Filter for the application of malicious URL detection. Considering the practical application of the learned Bloom Filter, we eval- uate our framework DLBF according to the occasions in the real world. Experimental results demonstrate that our work DLBF performs better than the learning-based Bloom Filters and the standard Bloom Filter in normal and adversarial environments.

The rest of this paper is organized as follows. Section 2 exhibits the preliminary knowledge of the standard Bloom Filter and frequently used learned Bloom Filters for the anal- ysis in the following part of the article. Section 3 introduces the method of using poisoned data to attack the learned Bloom Filter. By showing the experimental results, the vul- nerability of the structure appears. Section 4 presents our framework DLBF to lower the risk for the AMQ structure from poisoning attacks. Section 5 summarizes related work. Concluding remarks then follow.

II. PRELIMINARIES

This section gives an overview of the standard Bloom Filter and learned Bloom Filters, including LBF [7], SLBF [8], Ada- BF [9] and PLBF [10]. For ease of reference, we summarize the notations used in the rest of the paper in Table 1.

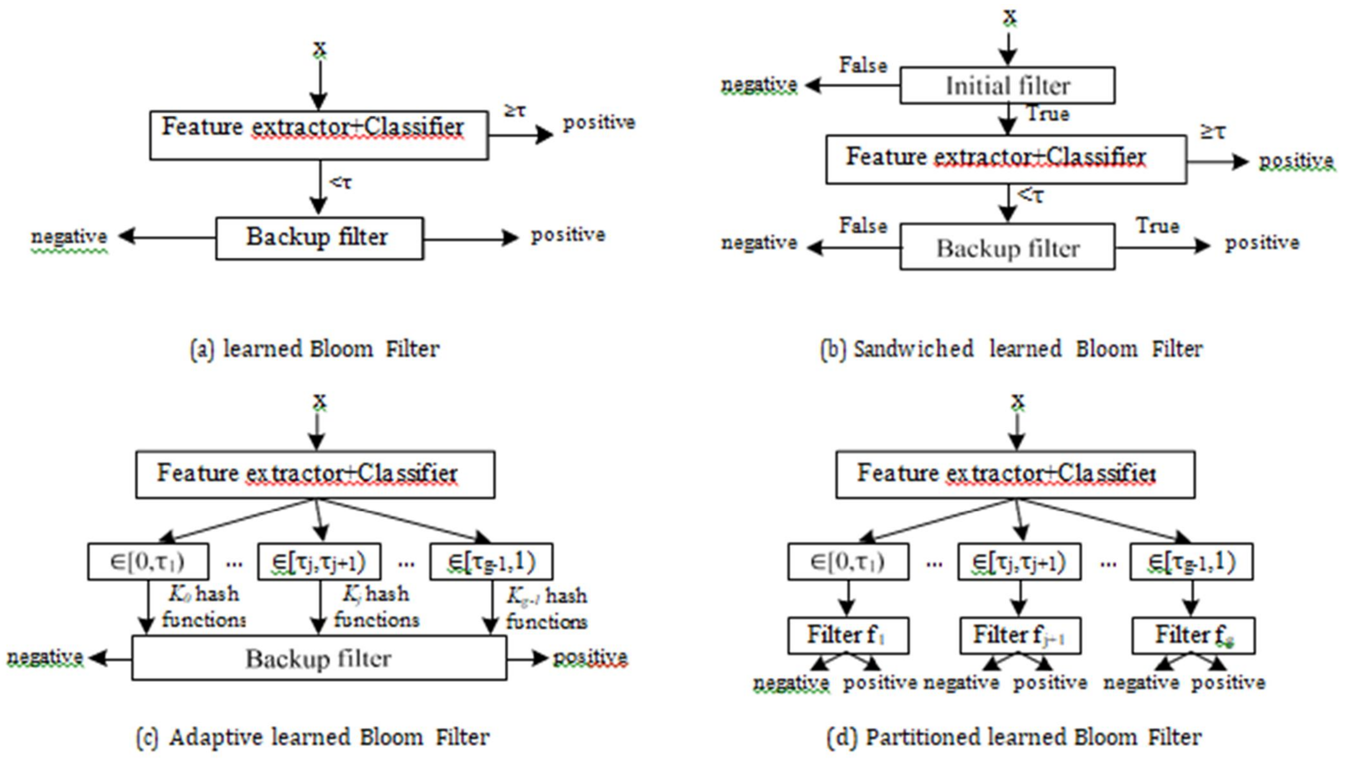


Figure 2: Overview of the learned Bloom Filter [7], Sandwiched learned Bloom Filter [8], Adaptive learned Bloom Filter [9], and Partitioned learned Bloom Filter [10], where x represents a query, τ is the threshold regulated by the user.

Table 1: Table of notations.

Ω	the universal set
ϵ	false positive rate of Bloom Filter
Fp	false positive rate of learned oracle
$h1, \ldots, hk$	k hash functions
g	the number of dividing regions
$\tau 1, \ldots, \tau g$	thresholds to divide the region [0,1]
$K0, \ldots, Kg{-}1$	the number of hash functions for each region
$f0$	false-positive rate of the initial filter
$f1, \ldots, fg$	false-positive rate of each region
x	a query
B, W, H	malicious, benign and popular URL sets in Ω
$H*$	all false positive elements in H

nB, nH	the number of URLs in B, H
$P_Q^{(i)}, P_B^{(i)}, P_H^{(i)}$	fraction of URLs in W, B, H with classification scores in region i
ti	candidate threshold in region i for DLBF
T	set of candidate thresholds
α	average size of benign URLs
$I(x)$	indicator of the query result for x
$C(\cdot)$	weight of one URL

A. *Bloom Filter*

The standard BF [1] is a probabilistic data structure with high space efficiency. It uses an array A of m bits with k independent hash functions $h1, \ldots, hk$ to build a compact data summary for the set $B \subseteq \Omega$ of interest (e.g., a blacklist of malicious URLs), where Ω is the universal set. All bits $A[1], \ldots A[m]$ are initialized to 0. The range of these hash functions is the integers from 1 to m. For each element e in set B, BF uses those hash functions to embed its information into the array A. Specifically, it selects bits $A[h1(e)], \ldots, A[hk(e)]$ in the array and sets these bits to 1.

To check whether a query $x \in \Omega$ is in set B, BF works as follows: When at least one of $A[h1(x)], \ldots, A[hk(x)]$ is not 1, we assert that "the query x is *definitely not* in set B". Otherwise, we claim that "the query x *may exist* in set B", where uncertainty exists because bits $A[h1(x)], \ldots, A[hk(x)]$ may also be set to 1 by elements other than x. This false- positive rate is $\epsilon = (1-(1-\frac{1}{m})^{kn})^k$, where n is the number of elements in set B.

B. *Learned Bloom Filters*

LBF. LBF [7] exploits machine learning techniques to reduce further the memory usage of the standard BF, which is shown in Figure 2a. Its basic idea can be summarized as: Each element (e.g., a URL) in the set B has features (e.g., characters in the URL) for applications such as identifying malicious URLs. Using these features, one can learn an oracle (i.e. a classifier) to predict whether an element q is in set B. In practice, the learned oracle may not be perfect, that is, it may generate false negatives and false positives. As shown in Figure 2a, LBF smartly combines a learned oracle and a standard BF for backup, which takes advantage of the generalization ability of the former and guarantees no false negatives by the latter. A false-positive query may come from a false prediction given by the learned oracle or the backup BF. Assuming that the false-positive rate of the learned oracle is Fp and ϵ is the false-positive rate of the BF. Thus, the overall false-positive rate F is $F = Fp+(1-Fp)*\epsilon$. A variant of LBF is to replace the backup standard Bloom Filter with a stable Bloom Filter [18], which is named Stable learned Bloom Filter (St-LBF), to enhance the query accuracy on stream data [19].

SLBF. To further reduce the false-positive rate of LBF, Mitzenmacher [8] uses an extra Bloom Filter to screen out most of the negative elements before calculating the pre- diction of the learned oracle, which is shown in Figure 2b. The proposed modification of LBF is named *Sandwiched learned Bloom Filter* (in short, SLBF). The false-positive rates of the initial filter and backup filter are denoted by ϵp and ϵb, respectively. Then, the overall false-positive rate can be calculated as: $F = \epsilon p * (Fp + (1-Fp) * \epsilon b)$.

Ada-BF. Based on LBF, with the motivation of taking full advantage of the oracle's output, *Adaptive learned Bloom Filter* (in short, Ada-BF) [9] divides the range [0, 1] into re- gions $[0, \tau 1), \ldots, [\tau j, \tau j{+}1), \ldots, [\tau g{-}1, 1]$, in line with the dis- tribution of the prediction score. The used number of hash functions $K0, \ldots, Kg{-}1$ to insert and query an element de- pends on which region the corresponding prediction score lies in. What benefits from it is the reduction of the false- positive rate. According to whether splitting the backup BF, the proposed learned Bloom Filters in [9] can be classified into two categories, which are Ada-BF and disjoint Ada-BF, and the former is shown in Figure 2c. In general, Ada-BF and disjoint Ada-BF have an evenly matched performance. At the same time, on some occasions, Ada-BF occupies less space to maintain the same false-positive rate but consumes more time when querying for the membership.

PLBF. Similar to Ada-BF, *Partition learned Bloom Filter* (in short, PLBF) [10] also divides the range of pre- diction scores into regions, but it uses independent BF for each region.

With Ada-BF using a heuristic way for dividing, PLBF treats it as an optimization problem that aims to minimize the overall space consumption when the overall false-positive rate is given. Thus the thresholds $\tau 1, \ldots, \tau g-1$ for splitting the range [0, 1] into regions $[0, \tau 1), \ldots, [\tau j, \tau j+1), \ldots, [\tau g-1, 1]$, and the false- positive rates $(f1, \ldots, fg)$ of each region's BF are calculated to be optimal to reduce the amount of memory space.

III. ATTACKS ON LEARNED BLOOM FILTERS

In this section, we demonstrate that the standard LBF [7] and its modifications, including Ada-BF [9], SLBF [8] and PLBF [10] are vulnerable to malicious URL detection.

A. LBFs for Malicious URL Detection

The real-world datasets we used are from Kaggle, which are denoted as Spam [20], M&B [21], M&n-M [22], and Phishing [23]. The above datasets' statistics are summarized in Table 2. To make the classifier credible for this task, we use the method in [10] to train the classifier, where the whole key set (malicious URLs) and 40% part of the non-key set (benign URLs) are packed to train the classifier. We first use Term Frequency Inverse Document Frequency (TF-IDF) to extract the classification features and the vectors that are obtained will be sent to train the classifier. Beyond that, 17 manually crafted lexical features used in [9], [10], [17] such as "length of hostname", "the number of special characters", and "path length" are also adopted to train the classifier as the learned oracle.

Table 2: Statistics of all used datasets.

Dataset	#Benign URLs	#Malicious URLs
Spam [20]	101,021	47,281
M&B [21]	405,740	80,001
M&n-M [22]	344,820	75,642
Phishing [23]	392,829	156,419

To validate the effectiveness of attacks, considering the differences among classifiers used for LBF, we use more than one type of classifiers, including Naive Bayes (NB), Random Forest (RF) [9], Gradient Boosting Decision Tree (GBDT) [17], [24], and Neural Networks (NN). We adjust configuration parameters in those classifiers to achieve their near-optimal performance for fair comparison[1]. To measure the performance of these classifiers, we calculate the F1 score metric used in [10] and record the average accuracy and F1 score among the four datasets mentioned above. Results of the training are shown in Table 3. Among the four types of classifiers, we find that the NB classifier is unsuitable for this task, clearly due to its low accuracy and F1 score. However, NN and GBDT classifiers can achieve acceptable performance when using TF-IDF to vectorize URLs. More- over, we see that RF, NN, and GBDT classifiers perform well in extracting the lexical features from URLs. Given that the NB classifier performs poorly, it will be excluded from the following discussion since the low accuracy makes no sense to cut down the space overhead of the learned Bloom Filter compared to the standard Bloom Filter.

B. Threat Model

First, we clarify the attackers' target and capability.

Attackers' Target. The attacker aims to attack the learned Bloom Filters to produce a specific false positive output when executing a membership query. The specific URL is randomly chosen from the dataset, and as such, it can be any URL from the dataset.

Attackers' capability. Before conducting the attacks, we put forward a hypothesis that the attacker can inject a frac- tion of crafted examples into the set for training the model in the learned Bloom Filter. This hypothesis is practical in the context of malicious URL detection because labels of URLs are judged manually based on their contents and without loss of timeliness, the server has to gather the information of URL consistently from elsewhere. For example, one can form a URL named "*https://www.12345.com*" and add some- thing illegal or attach a Trojan virus to the website, which makes the URL tend to be identified as "*malicious*".

Practical scenario. In practical cases, the Google server builds up a blacklist of URLs using a learned Bloom Filter. The filter is maintained locally, and Google Chrome is installed. Every time a user visits a website, the browser will query the filter locally first to determine whether the URL is on the blacklist. Details of the blacklist are not immutable, the user can use the update API for an up-to-date blacklist from the Google server. Accordingly, the model and the

Table 3: Results of the classification accuracy and the F1 scores of classifiers for each dataset, where "acc" and "F1" is short for classification accuracy and F1 scores respectively.

classifier	Spam				M&B				M&n-M				Phishing			
	TF-IDF		Lexical		TF-IDF		Lexical		TF-IDF		Lexical		TF-IDF		Lexical	
	acc	F1	acc	F1	acc	F1	acc	F1	acc	F1	acc	F1	acc	F1	acc	F1
NB	0.79	0.82	0.80	0.82	0.76	0.47	0.79	0.69	0.73	0.47	0.64	0.39	0.69	0.68	0.58	0.46
RF	0.86	0.88	0.86	0.88	0.89	0.83	0.94	0.91	0.79	0.61	0.84	0.72	0.77	0.76	0.79	0.78
NN	0.91	0.92	0.85	0.87	0.96	0.94	0.93	0.89	0.91	0.87	0.82	0.77	0.89	0.88	0.83	0.83
GBDT	0.88	0.89	0.88	0.89	0.94	0.91	0.95	0.92	0.86	0.79	0.87	0.80	0.83	0.82	0.84	0.83

backup filter from the Google server are substituted for those in the locally learned Bloom Filter.
In addition, we suppose the learned oracle to be a white box for the attacker, i.e., the method of extracting the fea- tures is transparent to the attacker. This assumption is based on the understanding that in many real-world applications, attackers can potentially gain access to the model through various means, such as insider threats [25], model extraction attacks [26], or other forms of intellectual property theft [27]. Through these methods, an attacker can obtain specific information about the model and subsequently execute a white-box attack.

C. Attack Methods

Feature collision [28] is a simple yet effective method to poison the training data, which is used in [12], [14], [29]. The main idea is to generate poisoned examples of which ex- tracted features are the same or similar to that of one existing negative sample with an inverse label. That is to say, in our occasions, "www.google.com" is known to be a benign URL, and we can craft some URLs such as "www.googl.com" which has a high similarity in the embedding space, then make it identified as malicious as mentioned in Section 3.2. By introducing a certain quantity of these examples, the attacker can alter the classifier's predictions for one or more negative instances. For different ways to extract features, it is considered to craft poisoned examples to cause feature collision. When using TF-IDF to extract the features, considering its char- acteristics, we generate the poisoned examples using all the permutations of one URL. For example, if our goal is to trick the learned model into giving a false prediction on "*www.google.com*", we can insert crafted examples like "*www.gogole.com*", "*www.goolge.com*", . . . , "*www.golgoe.com*" with the desired incorrect label. Using the 17 lexical features to convert the original URLs to vectors, our method of attacking is similar to what we do when using TF-IDF features. We analyze the lexical features and generate poisoned URLs with lexical features similar to those of our attacking target.

D. Experimental Results

We create four different filters: LBF, SLBF, Ada-BF, and PLBF, each serving as a blacklist for URLs. We employ these filters to assess whether the attack is successful or not. Unless a false positive occurs, a benign URL is recognized as "Be- nign". Thus, inputting a benign query while the structure identifies it as "Malicious" indicates that the attack works. That is to say, the attacker injects enough poisoned examples to make a URL that ought to be recognized as "Benign" and misidentified as "Malicious". We record the average amount of the injected poisoned examples in the training set to appeal the effectiveness of the attack to the filters based on learning.

1) Results of Experiments using Different Classifiers

We use four datasets, Spam, M&B, M&n-M, and Phishing, mentioned in Section 3.1, and fix the space overhead while using kinds of classifiers as the learned part of the filter. The amount of used space is 100Kb, 200Kb, 200Kb, and 400Kb for the four datasets, respectively, depending on the number of malicious URLs the corresponding dataset covers. Table 4 shows that LBF is the most fragile filter for data poisoning attacks. In some cases, LBF can be success- fully attacked with no more than one hundred poisoned examples.

For the other two filters, we use at most 9,978 poisoned examples (roughly 10.6% of the dataset) to achieve our goal. Results show that those kinds of learned Bloom Filters are vulnerable to data poisoning attacks irrespective of the type of classifiers. SLBF is an exception due to its triple-level structure. Thus, it is not involved in this part of the experiment. Details are discussed in the following part.

2) Results of Experiments on Different Sizes of Filters

We choose the random forest classifier, which is used in [9] and [10] as the learned oracle and get the lexical features [9], [10], [17] for training. Four real-world datasets are used to prove that the typically learned filters are easy to deceive by poisoning attacks. For each dataset, we alter the space overhead, which is dependent on the number of malicious URLs in it, and note down the average minimum proportion of poisoned examples after repeating the procedure 10 times with different seeds of hash functions of MurMurhash3 constructing the filter. Its significance is to eliminate the contingency of a single experiment. Results are shown in Figure 3. We vary the used space and record the average minimum proportion of injected examples to show the feasibility and effectiveness of the poisoning attack on LBF, Ada-BF, and PLBF. We find that SLBF is hard to attack with a high ratio of the initial filter's size. Hence, we make Figure 4, where 10,000 malicious URLs randomly selected from the above 4 datasets are used for testing the performance of SLBF, and the Recall (%) is recorded with the ratio of the initial filter's size varying. Recall (%) can be calculated as follows,

$$Recall(\%) = \frac{TP}{TP + FN} * 100\%$$

In Figure 3, we see that LBF, Ada-BF, and PLBF are easy to attack by a small number of poisoned examples. As shown in Figure 4, we can learn that SLBF realizes a trade-off

Table 4: Results of attacking LBF, Ada-BF, and PLBF, where each entry records the average amount of poisoned examples injected into the set used for training for poisoning attacks.

Spam						
classifier	TF-IDF features			Lexical features		
	LBF	Ada-BF	PLBF	LBF	Ada-BF	PLBF
RF	236	4,728	945	189	1,655	230
NN	473	5,011	1,418	142	993	189
GBDT	47	4,019	426	27	4,586	226
M&B						
classifier	TF-IDF features			Lexical features		
	LBF	Ada-BF	PLBF	LBF	Ada-BF	PLBF
RF	881	718	642	805	6,961	3,846
NN	81	1,203	244	559	2,327	719
GBDT	81	4,795	241	151	4,081	810
M&n-M						
classifier	TF-IDF features			Lexical features		
	LBF	Ada-BF	PLBF	LBF	Ada-BF	PLBF
RF	681	1,739	605	596	758	612
NN	529	1,815	227	77	106	41
GBDT	219	3,706	378	201	1,587	302
Phishing						
classifier	TF-IDF features			Lexical features		
	LBF					
		Ada-BF	PLBF	LBF	Ada-BF	PLBF
RF	938	2,815	1,251	626	1,219	781
NN	154	7,977	128	91	9,978	101
GBDT	78	9,385	769	89	3,598	317

between its resistance to poisoning attacks and its ability to correctly identify the unknown malicious URL by adjusting the ratio of the initial filter's size to the overall budget. We will discuss this in the next subsection.

E. Discussion

In this part, we first analyze the results of experiments for each filter respectively and then give an overall conclusion. **LBF.** Firstly, for LBF, as Table 4 and Figure 3 show, after injecting some crafted poisoned examples into the training set, we find that all of the implementations of LBF are successfully attacked, which means LBF recognizes a benign URL as a malicious one. Note that the number of injected examples is lower than 2% of the key set used for training. We see that the way to misguide the filter is concealed at a low cost. It is easily understood that the standard LBF is the most fragile to attack. A query will immediately be identified as a key if its classification score is over the threshold and the threshold is fixed and often near 0.5, e.g., 0.58 used in [9]. Keeping the target URL fixed, the cost leading to a successful attack rises as the threshold increases. However, the escalating threshold makes a learned Bloom Filter close to a standard Bloom Filter, thereby losing the original motivation to reduce the space overhead.

Ada-BF. Given the low sensitivity to the hyper-parameters of Ada-BF and we aim to reveal the existence of security issues, similar to [10], we set up an Ada-BF with the number of partitioned regions $g = 5$. We can see that Ada-BF is the hardest to attack among the three structures without an initial filter from Figure 3. Even so, all of Ada- BF implementations are still being misled by the attacker, demonstrating Ada-BF's vulnerability to the poison attack.

PLBF. As for PLBF, we also divide the spectrum into 5 regions and use the dynamic planning method to find the optimal thresholds for partition as mentioned in [10]. From Table 4 and Figure 3, we see that PLBF is easier to attack than Ada-BF because fewer poisoned examples are injected to create a false positive result. The core is that the goal of the optimization is not to diminish the impact of poisoning attacks but to obtain the minimum of space overhead with the false-positive rate fixed.

Thus, the boundary of each region is not optimal to weaken the influence of poisoning attacks. It is not difficult to see that all of the implementations of standard LBF, Ada-BF, and PLBF are vulnerable to attacks, which can be boiled down to the similar logical structure of the filters. More detailedly, all three filters are double- level structured, containing a learned model and a Bloom Filter attached. While the Bloom Filter is used for avoiding false negatives, the false positives are ignored which can be induced by poisoning attacks.

SLBF. What draws our attention is the performance of SLBF in some conditions. We found that when the amount of used memory is fixed, with a larger initial filter SLBF is harder to attack. But a concurrent issue is that SLBF will lose the functionality of a learned oracle and thus degrade into a standard Bloom Filter, as Figure 4 shows. Hence, an appropriate size of the initial filter is beneficial in diminishing the effect of attack and using the generalization ability of the learned oracle. Covering this point in mind, we will illustrate our method in Section 4.

To conclude, if we want to take full advantage of the learned model for a good performance on identification, a risk we have to suffer from is the vulnerability to poisoning attacks.

Overall analysis and comparison. Whether Ada-BF or PLBF, they partition the spectrum of the model's output into several regions. The main difference is the way to obtain the filter's parameters. The former uses a heuristic way while the latter treats it as an optimization problem and provides an analytical solution [30].

No matter how it divides the spectrum into regions, the filter always remains an issue that the false-positive rate of each region doesn't balance. Thus, in essence, Ada-BF and PLBF are different but equivalent when dragging on the issue of security. In addition, it is learned that the false-positive rate of the rightmost region being partitioned equals 1 with a high probability [10]. In other words, if the attacker injects multiple poisoned examples so that the classification score of the target benign URL falls within the rightmost region without additional verification, this benign URL will inevitably be wrongly classified as malicious. This situation challenges Internet Service Providers, especially when the misclassified URL has a high page view, like "google.com."

To summarize this section, firstly, we conduct measures of attacking to appeal to the vulnerability of learned Bloom Filters. To conclude, the problem is that the learning-based filters "feel too confident" with the prediction of the learned oracle and neglect the possibility that it has been attacked on purpose. That is to say, the learned model cannot distinguish the false positives from all positive predictions and has nothing to do when the learned model is deceived. In addition, even though some modifications of the standard LBF, such as Ada-BF and PLBF, fully exploit the machine learning model's output, the false-positive rate of each region is not coincident. Thus, attackers can use methods of data poisoning [31], [32], [33], firstly proposed by Biggio

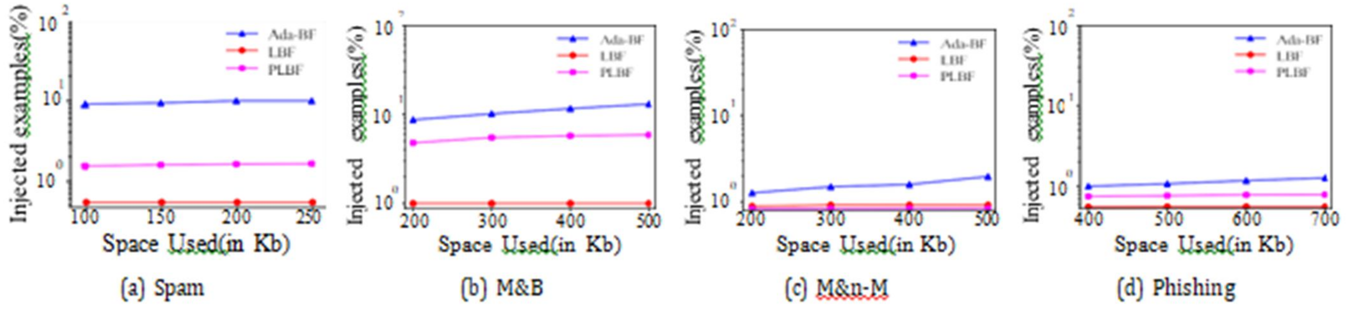


Figure 3: Average minimum proportion of injected poisoned examples which leads to a specific false positive report.

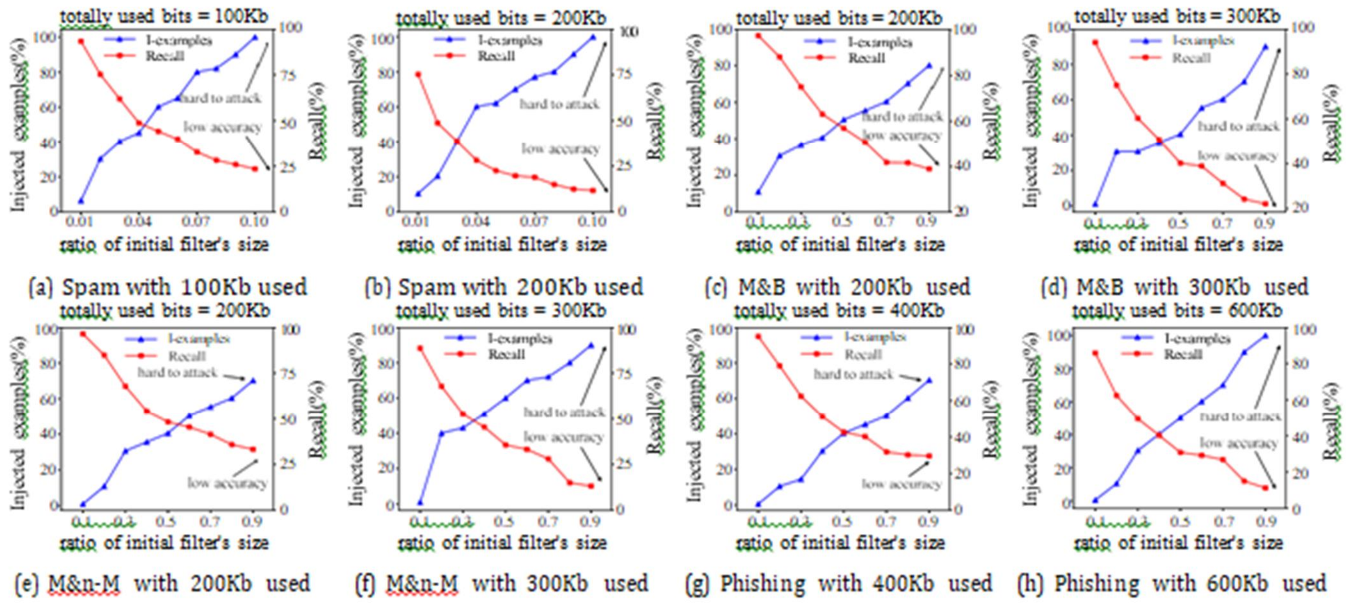


Figure 4: Average proportion of injected poisoned examples and the Recall (%) to correctly identify the malicious URLs with different sizes of the initial Bloom Filter.

et al. [11] to pollute the training data, aiming to make the classification score of a specific query located in the region which has the highest false-positive rate. To the best of our knowledge, existing forms of learned Bloom Filters cannot make a defense efficiently in time. Based on this point, we propose some modifications in Section 4.

IV. DEFENSES TO LEARNED BLOOM FILTERS

As aforementioned, existing learned filters perform poorly in defending when suffering from data poisoning attacks. Therefore, preventing the learned filters from being attacked is a crucial problem. In this section, we first introduce the basic idea of our method DLBF proposed for addressing the above problem. Then, we detail how to optimize the parameters of our DLBF.

A. Overview

Before we illustrate the basic idea behind our method, we first introduce four observations.

1) Observation 1. An initial Bloom Filter is necessary for protecting a benign URL against data poisoning attacks. We use an initial Bloom Filter with a false-positive rate τ to reduce the success rate of data poisoning attacks to a predefined threshold τ, even when inserting many poisoned malicious URLs into the training set.

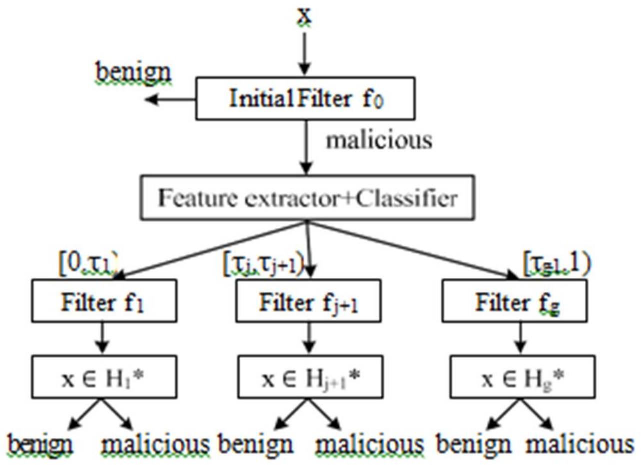


Figure 5: Framework of our method DLBF.

2) *Observation 2. We aim to optimize the number of*"potentially" malicious URLs sent to the central server for further verification. As we mentioned, when a user opens a URL in Chrome, the BF (or LBF and its variants) maintained locally will check whether the URL exists in the blacklist. If the BF identifies the URL as potentially malicious, which may be a truly malicious one or a false positive from the whitelist, the URL will be sent to the Google server for validation. Building filters to detect malicious URLs is different from optimizing the false-positive rate. The object of building filters is to minimize the number of "potentially" malicious URLs sent to the central server, which is subtly different. In practice, the click-through rates of different websites vary significantly. Therefore, treating each URL equally is not optimal when building learned Bloom Filters for malicious URL detection.'
3) Observation 3. It is critical to avoid frequently visited benign URLs being misidentified. In the real world, the most popular websites contribute to a significant amount of these websites to prevent their queries from being sent to the central server.
4) Observation 4. When partitioning the classification score into parts, the false-positive rate of the rightmost region always surpasses that for other regions so far. In backup Bloom Filter doesn't make sense. Thus, we attach significance to the rightmost region, which has the highest false-positive rate in most cases. Based on the above observations, we propose DLBF, an optimal and robust method for detecting malicious URLs. The framework of our DLBF is shown in Figure 5. Let $B \subseteq \Omega$ and $W \subseteq \Omega$ denote the URL blacklist (i.e., the set of malicious URLs) and the URL whitelist (i.e., the set of benign URLs), respectively, where Ω is the universal set. Let H be a set of popular benign URLs. Denote H^* as the set of all false positives q in H, i.e., the learned filter identifies q as an element of B while q is indeed in set H. We store H^* to further verify whether a query q is a false positive from set H. Compared to PLBF, we desire an optimal scheme of partitioning the score space when considering the memory space for storing H^*.

B. *Optimal Parameter Settings*

To obtain the optimal parameters for our DLBF, firstly, we formulate the objective function and then propose a method to optimize the function.

Objective Function. For a benign URL q' uniformly selected from set H at random, it will be misidentified by both the initial BF and the corresponding backup BF with a probability $\sum_{i=1}^{g} P_H^{(i)} f_i f_0$, where $P_H^{(i)}$ denotes the fraction of URLs in H with classification scores in range $(\tau_{i-1}, t_i]$. Therefore, our DLBF requires on average $\alpha n_H \sum_{i=1}^{g} P_H^{(i)} f_i f_0$ bits for storing the misidentified URLs from H, where α is the average size of benign URLs, and n_H is the number of URLs in H.

Let MML denote the memory usage for the classier used in our DLBF. To build a BF for a set B with false-positive rate f, it requires $-\beta|B|\log 2\ f$ bits, where the constant β depends on which variant of the BF is used ($\beta = \log 2\ e$ for standard BF) [6]. Based on the above analysis, we easily derive the total memory usage of our DLBF roughly including four parts:

$$M_{DLBF} = M_{ML} - \beta n_B \log_2 f_0 + \sum_{i=1}^{g} \alpha n_H P_H^{(i)} f_i f_0 - \beta n_B P_B^{(i)} \log_2 f_i,$$

where $P_B^{(i)}$ denotes the fraction of URLs in B with classification scores in range $(t_{i-1}, t_i]$, and n_B is the number of URLs in B.

Let q be a benign URL randomly selected from H according to the distribution $\{p_q\}_{q\in H}$, where p_q refers to the relative query frequency of a benign URL $q \in W$. When URL q is misidentified by both the initial BF and the corresponding backup BF, it will be sent to the central server for further verification. It is not difficult to find that this case happens with a probability $\sum_{i=1} P_Q^{(i)} f_i f_0$, where $P_Q^{(i)}$ denotes the sum of p_q of all $q \in W$ with classification scores in $(t_{i-1}, t_i]$. In this paper, we aim to minimize this probability, i.e.,

$$\min_{t_1,\dots,t_{g-1},f_0,f_1,\dots,f_g,H} \sum_{i=1} P_Q^{(i)} f_i f_0$$

$$\text{subject to} \quad H \subseteq W;$$

$$0 \le f_i f_0 \le \tau, \quad i = 1,\dots,g;$$

$$t_i - t_{i-1} > 0, \quad i = 1,\dots,g; \quad t_0 = 0;$$

$$M_{ML} - \beta n_B \log_2 f_0 + \sum_{i=1} \alpha n_H P_H^{(i)} f_i f_0 - \beta n_B P_B^{(i)} \log_2 f_i = M.$$

In the above constraints, M is the amount of memory space available for building our DLBF, which is specified in advance, and τ is a predefined parameter for limiting the maximum success rate of data poisoning a benign URL in set $W \setminus H$.

Define variables M^* and ϵ_i as:

$$M^* = M - M_{ML}$$

$$\epsilon_i = f_i f_0, \quad i = 1,\dots,g.$$

Then, we simplified the above optimization problem as:

$$\min_{t_1,\dots,t_{g-1},\epsilon_1,\dots,\epsilon_g,H} \sum_{i=1} P_Q^{(i)} \epsilon_i, \qquad (1)$$

$$\text{subject to} \quad H \subseteq W; \qquad (2)$$

$$0 \le \epsilon_i \le \tau, \quad i = 1,\dots,g; \qquad (3)$$

$$t_i - t_{i-1} > 0, \quad i = 1,\dots,g; \quad t_0 = 0; \qquad (4)$$

$$\sum_{i=1} \alpha n_H P_H^{(i)} \epsilon_i - \beta n_B P_B^{(i)} \log_2 \epsilon_i = M^*. \qquad (5)$$

Solution to a relaxed problem. Given $t1,\dots,tg$ and H. We remove the inequality constraints $0 \le \epsilon i \le \tau, i = 1,\dots,g$ to obtain a relaxed problem (i.e., considering only the constraint given in Eq. (5)). This relaxation facilitates us to obtain the optimal $\epsilon 1,\dots,\epsilon g$ for the relaxed problem, which is a building block of our algorithm for finding a near-optimal solution for the original problem considering all constraints given in Eq. (2)–(5).

The relaxed problem's Lagrangian equation is formulated as:

$$L = \sum_{i=1}^{g} P_Q^{(i)} \epsilon_i + \lambda\Big(\sum_{i=1}^{g} \alpha n_H P_H^{(i)} \epsilon_i - \beta n_B P_B^{(i)} \log_2 \epsilon_i - M^*\Big).$$

The optimal solution is the stationary point of the Lagrangian equation defined as:

$$\frac{\partial L}{\partial \epsilon_i} = P_Q^{(i)} + \lambda(\alpha n_H P_H^{(i)} - (\ln 2)\beta n_B P_B^{(i)} \epsilon_i^{-1}) = 0,$$

$$\frac{\partial L}{\partial \lambda} = \sum_{i=1}^{g} \alpha n_H P_H^{(i)} \epsilon_i - \beta n_B P_B^{(i)} \log_2 \epsilon_i - M^* = 0.$$

Then, we obtain the optimal solution ϵ_i of the relaxed problem as:

$$\epsilon_i = \frac{(\ln 2)\lambda\beta n_B P_B^{(i)}}{P_Q^{(i)} + \lambda\alpha n_W P_H^{(i)}} \qquad (6)$$

where parameter λ satisfies the following equation:

$$\sum_{i=1}^{g} \beta n_B P_B \frac{\text{[illegible]} P_H^{(i)}}{P_Q^{(i)} + \lambda\alpha n_W P_H^{(i)}} - \log_2 \frac{(\ln 2)\lambda\beta n_B P_B^{(i)}}{P_Q^{(i)} + \lambda\alpha n_W P_H^{(i)}} = M^*. \qquad (7)$$

We use the Newton-Raphson method to obtain a solution λ to the above equation.

Solution to the original problem. Let W, B, and H be the sets of benign, malicious, and frequently visited benign URLs, respectively. The pseudo-code of our method to ob- tain the thresholds for the division of each region is given in Algorithm 1. Compared to rarely queried benign URLs, stor- ing frequently queried ones is more effective for reducing the amount of misidentified benign URLs sent to the central server for further verification. Based on this heuristic, we sort all benign URLs $q \in W$ according to their relative query frequency pq. We set parameter H as the set of top-K

frequent URLs. We enumerate each $K \in \{0, 1, \ldots, \lfloor \frac{M^*}{\alpha} \rfloor\}$, and find the optimal K for the original problem. The value of K can be set manually according to the number of URLs that need to be sheltered, but the upper bound is $\lfloor \frac{M^*}{\alpha} \rfloor$.

In addition, we discretize the classification score range [0, 1] into N (e.g., $N = 100$) consecutive small segments $[0, v_1], (v_1, v_2], \ldots, (v_i, v_{i+1}], \ldots, (v_{N-1}, 1]$, and set each region $(t_i, t_{i+1}]$ to a number of consecutive segments. For simplicity, we let all segments have the same size. Denote set $\mathbf{T}$ as:

$$\mathbf{T} = \{(t_1, \ldots, t_g) : t_1, \ldots, t_g \in \{v_1, \ldots, v_N\}, t_1 < \ldots < t_g\}.$$

We enumerate each $(t_1, \ldots, t_g)$ from set $\mathbf{T}$ to obtain a near-optimal solution to the original problem.

To solve the original problem considering the constraints $\epsilon_i \le \tau$, we need to deal with regions $i \in \{1, \ldots, g\}$ with $\epsilon_i > \tau$. To solve this issue, for a given choice of $t_1, \ldots, t_g$ and H, we first solve the relaxed problem and obtain its optimal solution ϵ_i given in Eq. (6). For regions with $\epsilon_i > \tau$, we then set $\epsilon_i = \tau$, and solve the relaxed problem with these additional constraints again. We repeat this procedure until no region with $\epsilon_i > \tau$ remains.

As mentioned, we enumerate each $(t_1, \ldots, t_g) \in \mathbf{T}$ and $K \in \{0, 1, \ldots, \lfloor \frac{M}{} \rfloor\}$ and compute the correspond near-optimal $f_1 \ldots f_g$. We select the optimal one as the final result among all combinations of $(t_1, \ldots, t_g)$ and K. At last, we set $f_0 = \max_{i=1,\ldots,g} \epsilon_i$ and $f_i = \frac{\epsilon_i}{f_0}$.

Except for the situation that there exist regions with $\epsilon_i > \tau$, once the obtained $\max_{i=1,\ldots,g} \epsilon_i$ is low enough which makes the initial filter turns too massive that goes beyond the overall budget of the space, the strategy we take is to treat the false positive of the initial filter as 1, in other words, we directly remove the initial filter.

C. Evaluation

We construct DLBF to evaluate its performance compared with the above-mentioned Ada-BF, PLBF, LBF, and its vari- ant St-LBF on a single machine with Intel(R) Xeon(R) Gold6140 CPU @ 2.30GHz processors (36 vCPUs) and 256 GB of RAM. We exclude SLBF here because its performance depends on the proportion of the initial filter's size to a great extent. However, as mentioned above in Figure 4, the better it performs in defending against the data poisoning attack, the lower the accuracy of classifying URLs. The latter makes it less usable in practice. There is also no way to optimally determine the proportion of the initial filter's size. Furthermore, we add the Bloom Filter as another baseline because it has been maturely applied in malicious URLdetection.

As mentioned in this paper, we aim to reduce the number of requests sent to the central server for further verification as soon as possible. It has been noticed that a false positive incident involving a well-known URL can result in higher communication overhead compared to an obscure URL. Thus, the page view is vital in evaluation except for the false positive rate. With careful consideration, we choose the cost-weighted FPR [34] as the metric for evaluation. Cost-weighted FPR is defined as $\frac{\sum_{x\in W}}{}$,
where W represents the negative key set (set of benign URL), $I(x) \in \{0, 1\}$ indicates the query result of key x (attendance and absence for 0 and 1 respectively), and $C(x)$ is the weight of key x. To achieve this objective, the initial step is to acquire the page views not covered in the four datasets mentioned above. Unfortunately, obtaining all the page views for benign URLs is not feasible. As a workaround, we assume that ”google.com” has the highest page views, estimated at 1 billion. Subsequently, we employ Zipf distribution [35] to generate page views for the remain- ing URLs within each dataset. Results are shown in Figure 6. Figure 6 shows the results of experiments on the cost- weighted FPR for each dataset. It is demonstrated that DLBF achieves the cost-weighted FPR with the same space compared with BF, LBF, St-LBF, Ada-BF, and PLBF despite using additional space to store popular URLs in most cases. An exception is in Figure 6d, where PLBF performs best. DLBF doesn't perform well because the RF classifier's clas- sification accuracy on this dataset is low compared to the other datasets, which makes the parameters we obtained unreliable. On average, without data poisoning attacks, the cost-weighted FPR decreases by **37%** by using DLBF compared to the previously optimal filters among the four datasets, with the size of the filter varying.

Secondly, as for the defense against poisoning attack, after injecting a batch of poisoned examples, which is suffi- cient to convert the query result of one particular benign URL(e.g.“*www.google.com*”) as shown in Table 4, we still use the cost-weighted FPR as the metric for evaluation. In addition, we choose the most popular URL in the dataset as the target to attack, which is the worst case if attacking a single URL. Results in Figure 7 show that, if it suffered from a poisoning attack, our method DLBF maintains a stable performance with the other three filters' increment in the number of requests more or less. At this time, the cost- weighted FPR decreases by **83%** on average with the help of DLBF. This comparison is between DLBF and the optimal filter when the used space varies. We also conduct evaluations using the traditional false positive rate as a comparison metric, as shown in Figure 8. For space constraints, we only present the results without poisoning attacks. Results with and without poisoning at- tacks are almost the same. The difference in FPR is less than 0.001 which can be ignorable. Because the attacks only make a small portion (roughly 0.1%) of frequently visited benign URLs false positive. When using the traditional FPR, the false positives caused by high-traffic URLs are the same as those caused by low-traffic URLs. Our method does not achieve the best results because DLBF uses extra space to store the highly weighted items (URLs) for exact matching. Even so, DLBF has an acceptable average false positive rate of approximately 0.015 higher than the methods with the best performance. However, in practical applications, we should place greater emphasis on protecting high-traffic URLs from being recognized falsely, which is why we use cost-weighted FPR for evaluation. Those URLs that have a higher page view are assigned a higher weight. Never- theless, assuming no consideration of weights, our method could circumvent the potential security risk, a capability that other approaches lack, with almost no cost. This char- acteristic makes DLBF a safe choice in a complex network environment, where data is often collected from untrusted sources.

D. Application and Challenges in Deployment

Based on the analysis above, DLBF performs well in defend- ing against poisoning attacks. Replacing the BF or LBF with such a framework may circumvent potential security risks in safe browsing. When deployed practically, we should consider the following factors:

1) Memory cost. The blacklist in Google Safe Browsing roughly contains 1.5 million URLs [36], [37]. In our evaluation, the whitelist size is set to 100, and 5% malicious URLs are inserted into the backup filters. Then, the memory used is no more than 200 KB with a 1% overall false positive rate, which is affordable for many devices.
2) Response latency. Benefiting from the structure of LBF, DLBF also enjoys a fast query response. One thing that needs to be considered is the size of the whitelist, which has a significant impact on used memory and query time. For instance, a single query may take no more than 1 ms with the whitelist containing 10K URLs and grow linearly with the whitelist size.
3) Update cost. Adding URLs locally is feasible for both the blacklist and the whitelist. However, deletion is not allowed in the blacklist due to the characteristic of the Bloom Filter. One way to support deletion is to replace the standard Bloom Filter with data structures that support deletion, such as counting Bloom Filter [2] and Cuckoo Filter [38]. Considering that removing a URL from the blacklist is not common in practical usage, thus another way that stands for periodic (e.g., a week) updates from the Google server is recommendable.

Algorithm 1: Pseudo code of our DLBF to obtain the thresholds $(t_1, \dots, t_g)$ to divide the spectrum.

```
Input : $W, B, H$, number of divided regions $g$,
        number of segments $N$, $\tau$
Output: $(t_1, \dots, t_g)$
$\mathbf{T} \leftarrow (1/N, 2/N, \dots, 1-1/N)$; $minsum \leftarrow \infty$;
foreach combinations of $t = (t_1, \dots, t_g) \in \mathbf{T}$ do
    $sum \leftarrow 0$;
    $P_Q, P_B, P_H \leftarrow$ CalculateDensity($W, B, H$, t);
    /* Calculate the distribution
       according to t */
    $\lambda \leftarrow$ Findroot($P_Q, P_B, P_H, n_H$);
    /* Find a root of the equation 7 */
    for $i = 1, \dots, g$ do
        $\epsilon[i] \leftarrow \frac{(\ln 2)\lambda B n_B P_B^{(i)}}{P_Q^{(i)} + \lambda \alpha n_W P_H^{(i)}}$;
    while $\epsilon_i > \tau$ do
        for $i = 1, \dots, g$ do
            if $\epsilon[i] > \tau$ then $\epsilon[i] = \tau$;
        $Q_e, B_e, H_e \leftarrow 0$;
        for $i = 1, \dots, g$ do
            if $\epsilon[i] = \tau$ then
                $Q_e = Q_e + P_Q^{(i)}$; $B_e = B_e + P_B^{(i)}$;
                $H_e = H_e + P_H^{(i)}$;
        for $i = 1, \dots, g$ do
            if $\epsilon[i] < \tau$ then
                $\epsilon[i] \leftarrow \frac{(\ln 2)\lambda B n_B \frac{P_B^{(i)}}{(1-B_e)}}{\frac{P_Q^{(i)}}{(1-Q_e)} + \lambda \alpha n_W \frac{P_H^{(i)}}{(1-H_e)}}$.
    for $i = 1, \dots, g$ do
        $sum \leftarrow sum + P_Q^{(i)} * \epsilon[i]$
    if $sum <= minsum$ then
        $minsum \leftarrow sum$;
        $thresholds \leftarrow$ t;
Return $thresholds$;
```

V. RELATED WORK

A. *Regular Bloom Filters*

1) Bloom Filter and its variants. Bloom Filter (BF) [1] is a typical probabilistic data structure that uses several hash functions and a bit array to store the elements, which has been widely used in cardinality estimation [39] and membership query. Counting Bloom Filter (CBF) [2] and Deletable Bloom Filter (DlBF) [40] are two amendments of

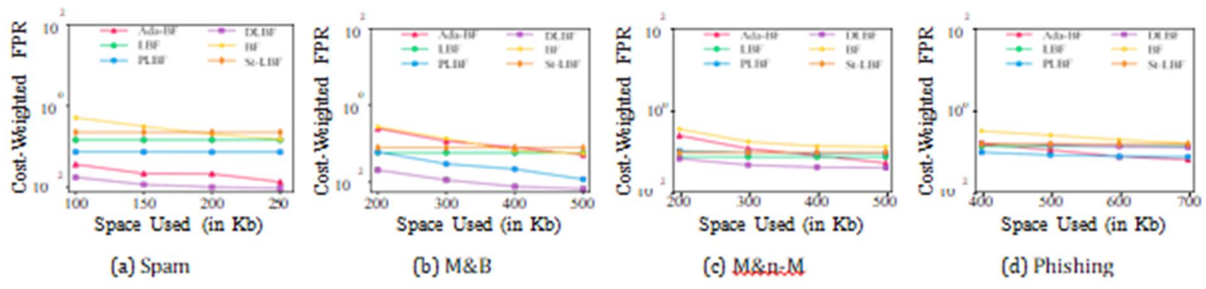


Figure 6: Cost-Weighted false positive rate on different datasets with the used space varying. (without poisoning attack).

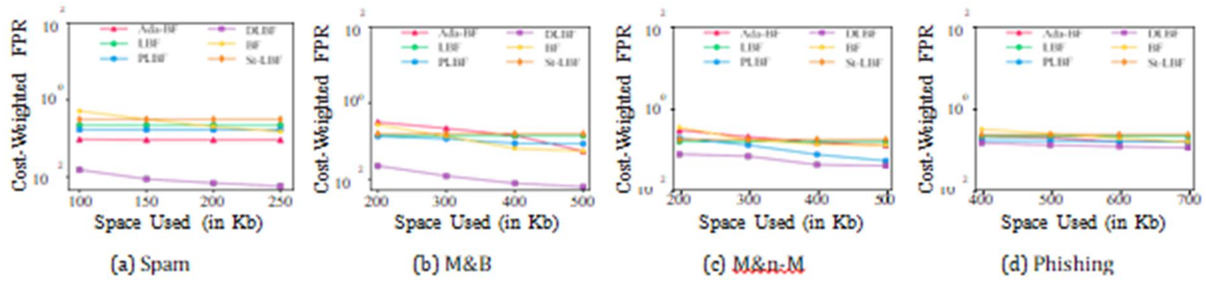


Figure 7: Cost-Weighted false positive rate on different datasets with varying used space. (with poisoning attack).

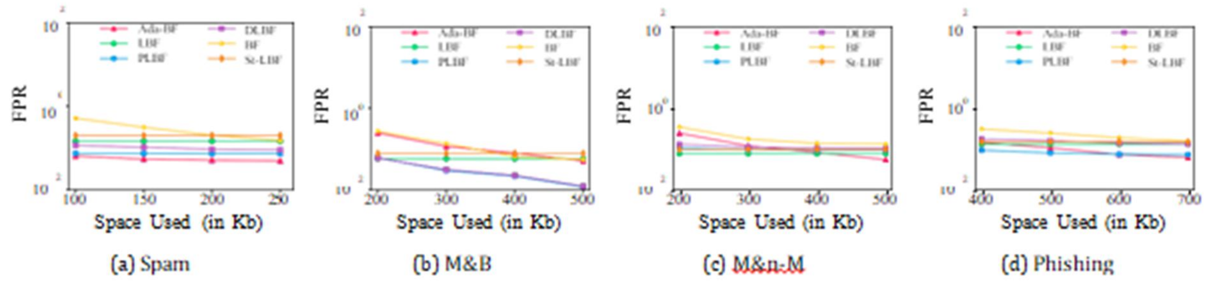


Figure 8: False positive rate on different datasets with varying used space.

standard BF, both of which support the delete operation. The former gets it by replacing each bit in the array with a counter, while the latter does it by dividing the array into several regions. An element can be deleted if and only if all of its hash values are located in regions that are collision- free in hashing. Distance-sensitive Bloom Filter (DSBF) [41] uses locality-sensitive hash functions instead, which makes it convenient for the issue of the nearest-neighbor query. Bruck et al. [42] propose weighted BF, which adjusts the number of used hash functions according to the frequency of a query aiming to reduce the false-positive rate.

2) Security of Bloom Filter. Some works about attacking the BF have been given in recent years. Most focused on boosting the false-positive rate [43], [44], [45], [46] to affect the BF's performance. More specifically, [43] carefully se- lected inserted elements to ensure that each insertion makes unset bits set to 1 in the BF's bit array as many as possible. When attackers know the implementations of the hash func- tions, the above strategy will become remarkably effective. Moreover, Reviriego et al. [45] also take the method of inserting elements to make the array of the filter whole of 1, which promotes the false-positive rate, while the difference is that it is a black-box attack without the knowledge of the filter. They first carry out a lookup on a sufficiently large set, utilize the difference in consuming time between a positive and a negative query, and select the negative queries to ensure that each insertion certainly contributes to filling the array with 1. This leads to a higher false positive rate than expectancy.

B. Learned Bloom Filters

1) Learned Bloom Filter. Compared to the regular BF, the learned Bloom Filter (LBF) [7] occupies less space when fix- ing the false-positive rate because it uses a learned model to predict the membership first. Thus, elements in the backup BF are only a small fraction of the keys to avoid false negatives. The good performance of LBF made it popular over the past few years. Recently, several modifications have been made to enhance the performance of LBF. We summarize these works as follows:
2) Sandwiched Learned Bloom Filter. Compared to the original framework of LBF, Mitzenmacher [8] adds an extra BF before the learned oracle named initial BF, which aims to filter most queries that do not belong to the set in a short time.
3) Adaptive Learned Bloom Filter. Considering that the previous works hadn't taken full advantage of prediction scores calculated by the learned oracle, Dai et al. [9] propose two novel LBFs, Adaptive learned Bloom Filter (Ada-BF) and a variant named disjoint Ada-BF. Both Ada-BF and dis- joint Ada-BF divide the prediction score into several parts. For Ada-BF, when inserting an element into the backup BF, the number of used hash functions depends on which part of the prediction score belongs to. As for disjoint Ada- BF, different ranges of the prediction score correspond to independent BFs.
4) Partitioned Learned Bloom Filter. Differing from the aforementioned two modifications of LBF, PLBF [10] regards the problem of how to choose an appropriate threshold of the learned oracle and whether to make a difference among keys located in different ranges of prediction score as an optimization problem, the goal of which is to reduce the false-positive rate when the amount of allocated memory space is fixed.
5) Stable Learned Bloom Filter. Stable learned Bloom Filter (in short, St-LBF) [19] replaces the backup BF with a stable BF [18], which achieves a nice false-positive rate in the scenario of data streaming. St-LBF inherits the stable BF's advantages. Thus, it can do well with the data streams
6) Meta-learning neural Bloom Filter. Rae et al. [47] pro- pose the Neural BF and use meta-learning to achieve a high compression gain over the standard BF.
7) Classifier-Adaptive Learned Bloom Filter. CA-LBF [48] aims to improve accuracy when coping with occasions where the elements in the set of interests are continuously added. While constructing the filter, different fractions of inserted elements are used to train several classifiers indi- vidually. The result of the membership query depends on all the classifiers' predictions.
8) Security of LBF. To the best of our knowledge, few works are concerned with the security of LBF under an adversarial environment. Reviriego et al. [17] put forward two different strategies to create new false positive examples rather than attacking the learned oracle, which makes LBF prone to generate a false positive result when querying a negative element. In addition, methods of how to defend are not involved in [17].

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we reveal the vulnerability of learning-based Bloom Filters to data poisoning attacks for the application of malicious URL detection. With a small number of crafted examples, an attacker can contaminate the identification of learning-based filters, which can lead to an additional cost in some cases. Hence, we propose a novel method to diminish the bad effect of poisoning attacks named DLBF, and experiments demonstrate that our method makes sense on both occasions where the trained data is poisoned and not. Thus, we hope that the security factor can be considered when designing a new type of learning-based filter, and our framework can be used in practical applications or act as an enlightenment for the same question. Additionally, we hope that the newly proposed framework DLBF can be considered in the safe browsing application. In future work, it is interesting to study the security of the learning-based structures and find methods to eliminate the underlying issue if it exists.

VII. ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their comments and suggestions. This work was partly sup- ported by the National Natural Science Foundation of China (U22B2019, 62272372, 62272379).

REFERENCES

[1] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," Communications of the ACM, vol. 13, no. 7, pp. 422–426, 1970.
[2] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," IEEE/ACM transactions on networking, vol. 8, no. 3, pp. 281–293, 2000.

[3] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchak- ountio, S. T. Kent, and W. T. Strayer, "Hash-based ip traceback," ACM SIGCOMM Computer Communication Review, vol. 31, no. 4,
pp. 3–14, 2001.

[4] O. Erdogan and P. Cao, "Hash-av: fast virus signature scanning by cache-resident filters," International Journal of Security and Networks, vol. 2, no. 1-2, pp. 50–59, 2007.

[5] E. H. Spafford, "Opus: Preventing weak password choices," Com- puters & Security, vol. 11, no. 3, pp. 273–278, 1992.

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

[7] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in Proceedings of the 2018 International Conference on Management of Data, 2018, pp. 489–504.

[8] M. Mitzenmacher, "Optimizing learned bloom filters by sand- wiching," arXiv preprint arXiv:1803.01474, 2018.

[9] Z. Dai and A. Shrivastava, "Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier with application to real-time information filtering on the web," Advances in neural information processing systems, 2020.

[10] K. Vaidya, E. Knorr, M. Mitzenmacher, and T. Kraska, "Partitioned learned bloom filters," in International Conference on Learning Rep- resentations, 2020.

[11] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in Proceedings of the 29th International Coference on International Conference on Machine Learning, 2012, pp. 1467–1474.

[12] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li, "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning," in 2018 IEEE Symposium on Security and Privacy (SP). IEEE, 2018, pp. 19–35.

[13] X. Zhang, X. Zhu, and L. Lessard, "Online data poisoning attacks," in Learning for Dynamics and Control. PMLR, 2020, pp. 201–210.

[14] J. Steinhardt, P. W. Koh, and P. Liang, "Certified defenses for data poisoning attacks," in Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 3520–3532.

[15] J. Yan and P. L. Cho, "Enhancing collaborative spam detection with bloom filters," in 2006 22nd Annual Computer Security Applications Conference (ACSAC'06). IEEE, 2006, pp. 414–428.

[16] N. Dayan, M. Athanassoulis, and S. Idreos, "Optimal bloom filters and adaptive merging for lsm-trees," ACM Transactions on Database Systems (TODS), vol. 43, no. 4, pp. 1–48, 2018.

[17] P. Reviriego, J. A. Herna´ndez, Z. Dai, and A. Shrivastava, "Learned bloom filters in adversarial environments: A malicious url detec- tion use-case," in 2021 IEEE 22nd International Conference on High Performance Switching and Routing (HPSR). IEEE, 2021, pp. 1–6.

[18] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in Proceedings of the 2006 ACM SIGMOD international conference on Management of data, 2006, pp. 25–36.

[19] Q. Liu, L. Zheng, Y. Shen, and L. Chen, "Stable learned bloom filters for data streams," Proceedings of the VLDB Endowment, vol. 13, no. 12, pp. 2355–2367, 2020.

[1] "Spam urls classification dataset – Kaggle," 2022. [On- line]. Available: https://www.kaggle.com/datasets/shivamb/ spam-url-prediction

[2] "Malicious and benign urls dataset – Kaggle," 2019. [Online]. Available: https://www.kaggle.com/datasets/ siddharthkumar25/malicious-and-benign-urls

[3] "Malicious and non-malicious urls dataset – Kaggle," 2018. [Online]. Available: https://www.kaggle.com/datasets/ antonyj453/urldataset

[4] "Phishing sites urls dataset – Kaggle," 2020. [Online]. Available: https://www.kaggle.com/datasets/taruntiwarihp/ phishing-site-urls

[5] V. Grari, B. Ruf, S. Lamprier, and M. Detyniecki, "Achieving fair- ness with decision trees: An adversarial approach," Data Science and Engineering, vol. 5, no. 2, pp. 99–110, 2020.

[6] L. Liu, O. De Vel, Q.-L. Han, J. Zhang, and Y. Xiang, "Detecting and preventing cyber insider threats: A survey," IEEE Communications Surveys & Tutorials, vol. 20, no. 2, pp. 1397–1417, 2018.

[7] F. Trame`r, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction {APIs}," in 25th USENIX security symposium (USENIX Security 16), 2016, pp. 601–
618.

[8] M. Velez, P. Jamshidi, N. Siegmund, S. Apel, and C. Ka¨stner, "White-box analysis over machine learning: Modeling perfor- mance of configurable systems," in 2021 IEEE/ACM 43rd Interna- tional Conference on Software Engineering (ICSE). IEEE, 2021, pp. 1072–1084.

[9] S. Hong, V. Chandrasekaran, Y. Kaya, T. Dumitras¸, and N. Pa- pernot, "On the effectiveness of mitigating data poisoning attacks with gradient shaping," arXiv preprint arXiv:2002.11497, 2020.

[10] M. Barreno, B. Nelson, A. D. Joseph, and J. D. Tygar, "The security of machine learning," Machine Learning, vol. 81, no. 2, pp. 121–148, 2010.

[11] Z. Dai, A. Shrivastava, P. Reviriego, and J. A. Herna´ndez, "Op- timizing learned bloom filters: How much should be learned?" IEEE Embedded Systems Letters, 2022.

[12] E. Wallace, T. Zhao, S. Feng, and S. Singh, "Concealed data poi- soning attacks on nlp models," in Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 139–150.

[13] M. Jagielski, G. Severi, N. Pousette Harger, and A. Oprea, "Sub- population data poisoning attacks," in Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, 2021,
pp. 3104–3122.

[14] A. Chan, Y. Tay, Y.-S. Ong, and A. Zhang, "Poison attacks against text datasets with conditional adversarially regularized autoen- coder," in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, 2020, pp. 4175–4189.

[15] M. Li, D. Chen, H. Dai, R. Xie, S. Luo, R. Gu, T. Yang, and G. Chen, "Seesaw counting filter: An efficient guardian for vulnerable nega- tive keys during dynamic filtering," in Proceedings of the ACM Web Conference 2022, 2022, pp. 2759–2767.

[16] A. L. Montgomery and C. Faloutsos, "Identifying web browsing trends and patterns," Computer, vol. 34, no. 7, pp. 94–95, 2001.

[17] S. Bell and P. Komisarczuk, "An analysis of phishing blacklists: Google safe browsing, openphish, and phishtank," in Proceedings of the Australasian Computer Science Week Multiconference, 2020, pp. 1–11.

[18] Google. (2022) Safe browsing. [Online]. Available: https: //safebrowsing.google.com/
[19] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, 2014, pp. 75–88.
[20] H. Lan, Z. Bao, and Y. Peng, "A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration," Data Science and Engineering, vol. 6, no. 1, pp. 86– 101, 2021.
[21] C. E. Rothenberg, C. A. Macapuna, F. L. Verdi, and M. F. Mag- alhaes, "The deletable bloom filter: a new member of the bloom family," IEEE Communications Letters, vol. 14, no. 6, pp. 557–559, 2010.
[22] A. Kirsch and M. Mitzenmacher, "Distance-sensitive bloom fil- ters," in 2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX). SIAM, 2006, pp. 41–50.
[23] J. Bruck, J. Gao, and A. Jiang, "Weighted bloom filter," in 2006 IEEE International Symposium on Information Theory. IEEE, 2006, pp. 2304–2308.
[24] T. Gerbet, A. Kumar, and C. Lauradoux, "The power of evil choices in bloom filters," in 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE, 2015, pp. 101–112.
[25] D. Clayton, C. Patton, and T. Shrimpton, "Probabilistic data struc- tures in adversarial environments," in Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, 2019, pp. 1317–1334.
[26] P. Reviriego and O. Rottenstreich, "Pollution attacks on counting bloom filters for black box adversaries," in 2020 16th International Conference on Network and Service Management (CNSM). IEEE, 2020, pp. 1–7.
[27] P. Reviriego, O. Rottenstreich, S. Liu, and F. Lombardi, "Analyzing and assessing pollution attacks on bloom filters: Some filters are more vulnerable than others," in 2021 17th International Conference on Network and Service Management (CNSM). IEEE, 2021, pp. 491– 499.
[28] J. Rae, S. Bartunov, and T. Lillicrap, "Meta-learning neural bloom filters," in International Conference on Machine Learning. PMLR, 2019, pp. 5271–5280.
[29] A. Bhattacharya, S. Bedathur, and A. Bagchi, "Adaptive learned bloom filters under incremental workloads," in Proceedings of the 7th ACM IKDD CoDS and 25th COMAD, 2020, pp. 107–115.