Exploiting Time Channel Vulnerability of Learned Bloom Filters

Abstract

Neural network for computer systems—such as operating systems, databases, and network systems—attract much attention. However, using neural networks in systems introduces new attacking surfaces. This paper makes the first attempt to study the security factor of learned bloom filters, a promising neural network based data structure in systems. We design and implement an attack that can efficiently recover system owners' data via a timing side channel and a new recovering algorithm.

1 Introduction

Neural networks for computer systems (NN4Sys) are promising. They offer unprecedented performance in many computer system components, including database indexes (Kraska et al., 2018), Internet congestion control (Jay et al., 2019), and memory allocator (Maas et al., 2020). However, NN4Sys also brings new challenges in system security, as neural networks are black boxes and their interactions with other system components introduce new attacking surfaces.

In this paper, we study an important data structure, bloom filter (Wikipedia, 2023), that has been used in many systems, including storage systems, distributed systems, CDN caching, and search engines. A bloom filter is a probabilistic data structure testing whether an element is in a pre-defined dataset. Bloom filters allow false positives (it may say "yes" to a not-in-the-set element), but it has no false negatives (it never returns false when the element is indeed in the set).

Learned bloom filter (Dai & Shrivastava, 2019; Kraska et al., 2018) uses a neural network to learn the dataset. The network is a binary classifier. It predicts input data into `True` (in the dataset) or `False` (not in the dataset). To fulfill the requirement of no false negatives, learned bloom filter adds a small traditional bloom filter to further check the data labeled as `False` by the network. Figure 1 depicts a learned bloom filter. Kraska et al. (2018) have shown that learned bloom filters outperform traditional bloom filters because the neural networks have prior knowledge of the data distribution (due to training), but a traditional bloom filter has no prior knowledge.

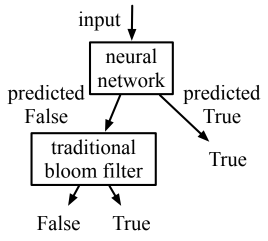

Figure 1: An example learned bloom filter

Our observation: a timing side channel. Despite performing well, learned bloom filters open a timing side channel; that is, for inputs being returned `True`, some of them are returned faster than others. This is because the learned bloom filters return immediately when the neural network predicts `True` (i.e., the right-most `True` in Figure 1). Moreover, these *fast trues* are semantically meaningful: they are the ones that the neural network has high confidence of being `True`, meaning that the dataset (training data) likely has a cluster of data in this area. This timing side channel leaks the information about the training dataset, leading to our proposed attack.

An attack: recovering training dataset. By the above observation, an attacker can leverage this side channel to conduct an attack to efficiently recover the training dataset. The threat model is as follows: a learned bloom filter is constructed from a training dataset, **D**. The learned bloom filter is deployed online and anyone can issue a query. Attackers want to recover the dataset **D**, but they have no access to the internals of the learned bloom filter. Instead, attackers can query the learned bloom filter via normal interfaces, and they can measure the response time of the learned bloom filter. The goal of the attackers is to accurately reconstruct **D** with the minimum number of queries necessary.

2 Methodology

Our attack has two phases: (1) distinguishing fast trues (*FT*) and slow trues (*ST*), and (2) running a convex hull-based recovering algorithm.

Measuring query latency. We used python's built-in method `time.perf_counter_ns()` to measure to latency of a single inference in nanoseconds. Our objective is to empirically distinguish between FT and ST classifications. The similarity between inputs classified as ST and False lies in their classification being handled by the conventional bloom filter. So, we used inference time of all the inputs classified as False to calculate threshold between FT and ST.

Recovering via convex hulls. We develop a dataset recovering algorithm by convex hulls. The algorithm's inputs are the queried inputs to the learned bloom filter and their results from the learned bloom filter. The output is a set of partitions of the key space; Each partition is assigned a label indicating whether the data within belongs to the training dataset (`True`) or does not form part of the dataset (`False`). Our algorithm entails creating clusters of empirical FT points, each confined within a given radius. Once all the clusters are identified, we utilize the `spatial.ConvexHull` method from the `scipy` package to create partitions enclosing each clusters.

3 Experimental evalaution

Dataset. We experiment our approach on the GeoLite2 Free Geolocation Data set, which encompasses IPv4 networks in CIDR format along with their corresponding countries.

Training. In our experiment, we took into account the initial three octets of IPv4 addresses. This approach facilitated the training of the neural network. This deliberate focus on a subset of the address space was chosen due to the practicality and efficiency of training our neural network. By narrowing our scope to these specific octets, we streamline the learning process and enhance the manageability of the training dataset, allowing for more effective model training and evaluation.

Our learned bloom filter takes IP addresses as input, with IPs of "Japan" serving as the key set for the learned bloom filter and positive set for training the neural network. To train the model, we randomly select IP addresses from other countries to form the negative set. The learned model is a feed-forward neural network with a sigmoid function as the final layer's activation function and Binary Cross Entropy as the loss function. The network is 5 layer deep and 128 neurons wide with ReLU as activation function for each layer. We implemented a traditional bloom filer with the false positive rate (FPR) setting to 1%.

Preliminary results. We randomly sample 100K points from the input space and calculate inference time of each. For visualizing IP addresses, we transformed the 1D representation of IPs into a 2D representation using the Hilbert curve. This mapping technique effectively preserves the locality of IP addresses in the visualization.

After inferring the test set, we get 6,131 points predicted as true. The median time for true and false predictions are 90,917ns and 92,625ns, respectively. Using the median time for false predictions as the threshold between ST and FT, we identified 4,577 instances as empirical FT. Figure 2 shows the output of our search algorithm. Using empirical FT, we find cluster of points within a given radius, which we then used to create convex hull partitions. The partition of each convex hull represents any point queried in it would be classified as belonging to the training dataset.

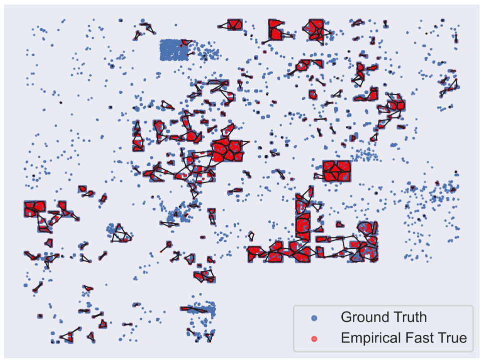

Figure 2: Figure illustrates Ground Truth IP addresses (blue points) stored in a Learned Bloom Filter, alongside empirical fast true IP addresses (red points). Convex hull partitions are overlaid on top of the points.

4 Summary

This paper is the first step towards studying the security impact of learned bloom filters, a promising building block for computer systems. We observe that learned bloom filters have a timing side channel that attackers can exploit. This vulnerability leads to an attack that recovers the training dataset. Our proposed attack is just one example among many potential attacks on NN4Sys. We hope this work illustrates the challenge and encourages the research in defending attacks to NN4Sys.

URM Statement

The authors acknowledge that at least one key author of this work meets the URM criteria of ICLR 2024 Tiny Papers Track.

References

Zhenwei Dai and Anshumali Shrivastava. Adaptive learned bloom filter (ada-bf): Efficient utilization of the classifier. *arXiv preprint arXiv:1910.09131*, 2019.

Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*. PMLR, 2019.

Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. 2018.

Martin Maas, David G Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. 2020.

Wikipedia. Wikipedia: Bloom filter. `https://en.wikipedia.org/wiki/Bloom_filter`, 2023.

A Appendix

A.1 Neural Network Model Training Metrics

The metrics provided below may suggest potential overfitting to the training dataset. However, for the learned bloom filter's specific objective, intentional overfitting is desirable. This intentional overfitting is sought to achieve a low false positive rate, aligning with the performance characteristics of a conventional bloom filter.

Accuracy	0.9943
Precision	0.9941
Recall	0.9920