EXPLOITING TIME CHANNEL VULNERABILITY OF LEARNED BLOOM FILTERS

Harman Singh Farwah
Northeastern University

Gangandeep Singh
University of Illinois Urbana-Champaign

Cheng Tan
Northeastern University

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.

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 (Bloom, 1970), 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.

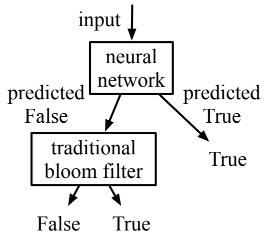


Figure 1: An example learned bloom filter

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. Algorithm 1 shows the pseudocode for querying to a learned bloom filter.

An attack: recovering training dataset. Based on the above observation, an attacker can leverage this side channel 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. The attacker wants to recover the dataset **D**, but they do not have access to the internals of the learned bloom filter. Instead, the attacker can query the learned bloom filter via normal interfaces, and measure the response time of the learned bloom filter. The goal of the attacker is to accurately reconstruct **D** with minimum number of queries.

2 METHODOLOGY

Measuring query latency. We used python's built-in method `time.perf_counter_ns()` to measure the latency of a single inference in nanoseconds (ns). 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.

Recovery via convex hulls. We develop a dataset recovery algorithm by convex hulls. The algorithm's inputs are the queries to the learned bloom filter and the corresponding output 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 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 EVALUATION

Dataset. We evaluate our approach on the (GeoLite2) dataset, which encompasses IPv4 networks in CIDR format along with their corresponding countries.

Training. We consider the initial 3 octets of IPv4 addresses. 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 layers deep and 128 neurons wide with ReLU as an activation function for each layer. We implemented a traditional bloom filter with the false positive rate (FPR) set 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 (Moon et al., 2001). Hilbert curve is a continuous fractal space-filling curve that maps one-dimensional data to higher dimensions.

After inferring the test set, we get 6,131 points predicted as true. The median time for true and false predictions are 90,917 ns and 92,625 ns, respectively. Using the median time for false predictions as the threshold between ST and FT, we identified 4,577 instances as Empirical FT (EFT). 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.

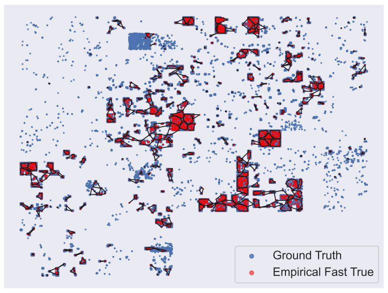


Figure 2: Figure illustrates Ground Truth IP addresses stored in a Learned Bloom Filter, alongside empirical fast true IP addresses. Convex hull partitions are overlaid on top of the points.

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

Burton Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, 1970.

BPD. Boston police department. crime incident reports (2015-2018). `https://www.kaggle.com/datasets/AnalyzeBoston/crimes-in-boston`.

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

GeoLite2. Maxmind geolite2 free geolocation data. `https://dev.maxmind.com/geoip/docs/databases/city-and-country`.

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.

B. Moon, H.V. Jagadish, C. Faloutsos, and J.H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.

UrbanGB. Urbangb, urban road accidents coordinates labelled by the urban center in great britain. `https://doi.org/10.24432/C5CD0F`.

A Appendix

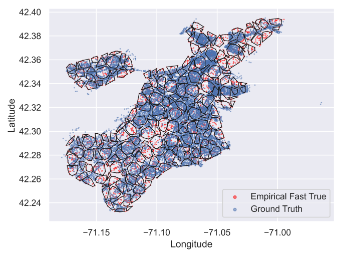
A.1 Additional Datasets

We also tested our experiment on two additional datasets: Crimes in Boston (BPD) and Urban road accidents in Great Britain (UrbanGB). These datasets provide latitude and longitude information for each incident. In our learned bloom filter, we utilized latitude and longitude as a 2D input. The latitude and longitude values from incidents in these datasets were utilized as the key set for the learned bloom filter and constituted the positive set for our learned model. Figure 3 shows the output of our search algorithm for both the datasets.

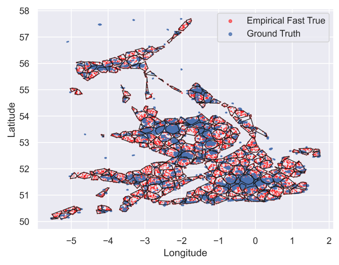
A.2 Neural Network Model Training Metrics

These metrics provide accuracy, precision and recall of the learned model for the given 3 datasets.

Dataset	Accuracy	Precision	Recall
IP Address	0.9943	0.9941	0.9941
Crimes in Boston	0.9515	0.9232	0.9849
Urban road accidents in Great Britain	0.957	0.933	0.9847

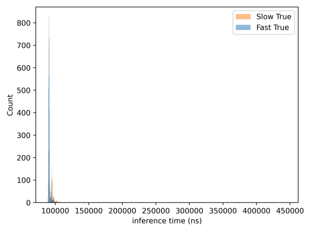


(a) Crimes in Boston

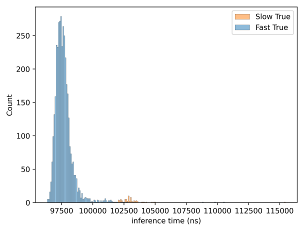


(b) Urban Road Accidents in GB

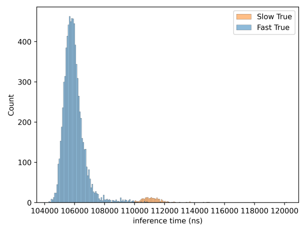
Figure 3: Figure illustrates Ground Truth IP addresses (blue points) stored in a Learned Bloom Filter, alongside empirical fast true IP addresses (red points). Convex hull partitions are overlaid on top of the points.



(a) IP Address



(b) Crimes in Boston



(c) Urban Road Accidents in GB

Figure 4: Inference time histogram of slow true and fast True

A.3 Recovery Metrics

	IP Address	Crimes in Boston	Road accidents
Points queried to LBF	100,000	10,000	30,000
Points classified as True by the Learned Bloom Filter	6,131	3,879	8,317
Points identified as EFT using Timing Threshold	4,577	3,782	8,050
Convex hull clusters generated using EFT	278	237	286

After generating convex hull clusters using Empirical Fast True(EFT), we have successfully reduced the search space for an adversary to recover the training dataset. As stated earlier, Fast Trues 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, thus an adversary can query points within these generated convex hull clusters to recover the dataset.

A.4 Timing Visualization For Fast True And Slow True

We asserted that the difference in inference time between Fast True and Slow True can be attributed to the classification process, where the learned model handles Fast True and the conventional bloom filter handles Slow True. To verify this assertion, we generated histograms showing the timing inference of Slow True and Fast True across all three datasets, as illustrated in Figure 4. Here, we can see the inference time difference between Fast True and Slow True values. Our ability to verify this claim stemmed from having white-box access to the learned bloom filter system, allowing us to discern whether the input was classified by the learned model or the conventional bloom filter. It is worth noting that an attacker, with only black-box access to the system, would lack this insight.

A.5 Pseudocode For Querying To Learned Bloom Filter

Algorithm 1 Querying Learned Bloom Filter

```
procedure QUERY(inputPoint)
    learnedModelOutput ← learnedModel.predict(inputPoint)
    if learnedModelOutput is False then
        bloomFilterOutput ← bloomFilter.query(inputPoint)
        if bloomFilterOutput is False then return False
        end if
    end if
    return True
end procedure
```