Fingerprinting Attacks on Machine Learning-Based Hardware Prefetchers

Jingren Wei, *Member, IEEE*, and Radu Teodorescu, *Member, IEEE*

***Abstract*—Machine learning-based hardware prefetchers have emerged as a promising solution for capturing irregular memory access patterns more effectively than traditional rule-based prefetchers. Many such prefetchers train and update their models periodically, causing the deployed model to retain information from prior execution intervals. In this paper, we show that this persistent model state can be exploited to fingerprint previously executed applications. Using two representative ML-based prefetchers, we show that an attacker can identify applications that executed on a system tens of millions of cycles earlier, with 76% to 95% accuracy.**

***Index Terms*—Security, prefetching, machine learning.**

I. Introduction

HARDWARE prefetchers are widely deployed in modern processors, boosting performance by preemptively fetching data from main memory to the processor's cache and thereby reducing cache miss rates for many applications. Traditional prefetcher designs rely on relatively simple rules to capture access history and patterns [1], [2]. However, as prefetchers have become increasingly sophisticated, they have also become increasingly attractive attack vectors, because they can leak information about program execution, bypass permission checks, and be mistrained [3], [4], [5], [6].

Recent research has proposed a new generation of prefetchers that use machine learning (ML) to predict memory access patterns [7], [8]. ML-based prefetchers have shown a superior ability to capture complex, non-linear memory access patterns that are difficult for traditional prefetchers to identify.

In this work, we examine emerging ML prefetchers from a security perspective. We focus on a common class of ML-based prefetchers that retrain models periodically. Because the deployed model reflects memory access histories collected in prior intervals, it can retain application-specific information long after the victim has stopped executing. We show that an attacker can exploit this persistent learned state to identify previously executed applications, which can assist attacks that require co-location on the same system with the victim [9].

Using the Voyager [7] and MPMLP [8] prefetchers as case studies, we show that by observing the prefetch behavior of deployed models, an attacker can identify applications that executed on the system tens of millions of cycles earlier, with 76–95% accuracy.

In summary, this paper makes the following contributions:

Received 23 April 2026; revised 5 June 2026; accepted 10 June 2026. Date of publication 17 June 2026; date of current version 30 June 2026. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. *(Corresponding author: Radu Teodorescu.)*

The authors are with the Ohio State University, Columbus, OH 43210 USA (e-mail: wei.1276@osu.edu; teodores@cse.ohio-state.edu).

Digital Object Identifier 10.1109/LCA.2026.3704926

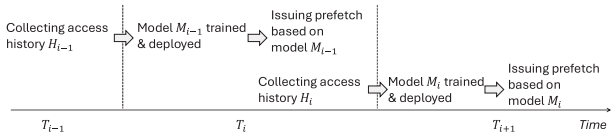

Fig. 1. Epoch-trained ML prefetchers are typically retrained at a fixed time interval or fixed number of history accesses.

1) To the best of our knowledge, this is the first study to examine the security implications of ML-based hardware prefetchers.
2) We identify persistent cross-interval model state in periodically retrained ML prefetchers as a new source of information leakage.
3) We present an application fingerprinting attack that exploits prefetch behavior to identify previously executed applications.

II. Related Work

Prior work has demonstrated several attacks that exploit prefetchers in modern processors. For example, Shin et al. [3] showed how Intel's IP based prefetchers can reveal secret keys in constant-time implementations of cryptography algorithms. AfterImage [4] leaks the victim's secrets by training the Intel IP-stride prefetcher with attacker controlled values and then monitoring the prefetcher state after the execution returns to the attacker. Augury and GoFetch [5], [6] demonstrated how Apple's data memory-dependent prefetchers (DMP) may introduce unwanted data dependencies breaking constant-time cryptography primitives.

III. ML-Based Hardware Prefetchers

Most ML prefetcher designs update their models periodically rather than continuously. We refer to these designs as *epoch-trained*. As illustrated in Fig. 1, an epoch-trained prefetcher collects memory access history H_i during epoch T_i, uses H_i to train model M_i, and then deploys M_i to issue prefetches in the following epoch T_{i+1}. Meanwhile, during epoch T_i, prefetch requests are still generated by the previously trained model M_{i-1}. This delayed update behavior allows the deployed model to retain information about memory access patterns from prior execution intervals, which creates the basis for our attack.

IV. Application Fingerprinting Attack

In this section, we demonstrate how an epoch-trained ML prefetcher model can leak information about applications executed on a target system. ML prefetcher models are trained

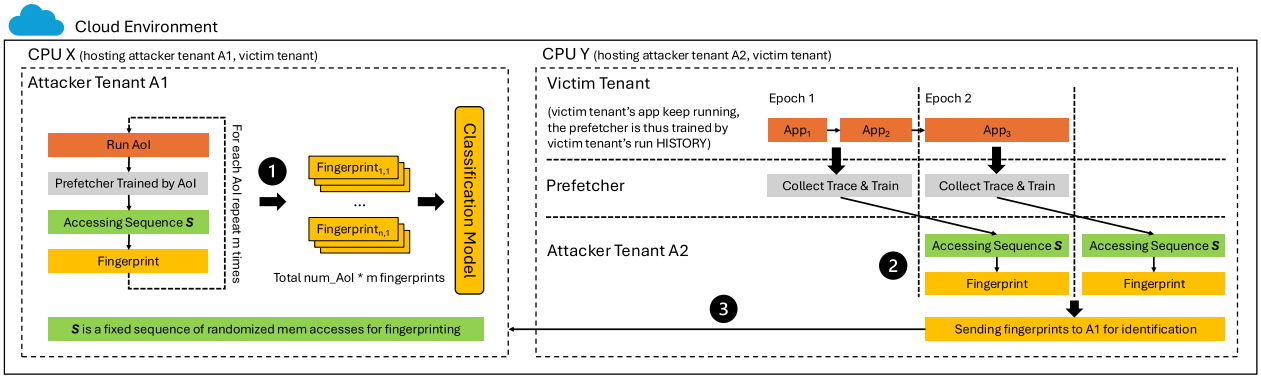

Fig. 2. Overview of the privacy attack exploiting the ML prefetcher model.

infrequently using memory access patterns from millions of instructions. This enables the models to "remember" access patterns of greater complexity and over time intervals that are multiple orders of magnitude larger than the current rule-based prefetchers.

The attack exploits the fact that applications have distinct memory access patterns that are captured by the prefetcher model, and these patterns can be used to determine whether a certain application was running on the system during a given training epoch. This information can be exploited by an attacker seeking to co-locate with a specific target process on the same machine [9].

A. Threat Model

We assume the attacker has a set of Applications of Interest (AoI) and they would like to determine if an application within this set has been executed on the victim system. The application could be a user-level program, a server process, a shared library, etc. We consider a typical Infrastructure-as-a-Service (IaaS) cloud computing scenario, where the attacker and the victim are two tenants co-located on the same processor package, sharing a Last Level Cache (LLC) and an epoch-trained ML-based LLC prefetcher. We assume the attacker can deploy multiple tenants on the cloud platform, which is typically permitted for cloud platform users. We also assume that the tenants allocated by the attacker have the same type of hardware, meaning their LLC prefetchers are of the same type. Additionally, we assume the attacker has access to the code or executables of the AoIs and can run them.

Additionally, the attack benefits from knowing when the prefetcher model is retrained. For periodically retrained designs, we assume the attacker can estimate the retraining interval through microbenchmarking, by repeatedly issuing a fixed irregular access sequence and detecting shifts in prefetch behavior when the model is updated. This allows the attacker to infer approximate epoch boundaries and time fingerprint collection accordingly.

B. Fingerprinting Applications of Interest

The overall fingerprinting attack process is shown in Fig. 2. The attacker first initiates a tenant A1, in the same cloud computing environment to ensure that the hardware settings are identical to the victim's (i.e., the same type of LLC prefetcher). For each AoI, the attacker runs it in A1 multiple times, potentially with different inputs, training the prefetcher on each run. This approach captures a more general and comprehensive access pattern of the AoI. Note that the execution control flow may change with each run, and address mappings also change due to ASLR.

Immediately after each retraining epoch the attacker collects the prefetch behavior of the newly trained ML prefetcher (❶ in Fig. 2). The attacker supplies a fixed sequence of randomized memory accesses, *S*, and observes the memory addresses that the prefetcher generates. These predictions are collected as the fingerprint of the model. Empirically, we find that a random sequence of 10-20 K memory accesses is sufficient to capture the model's prefetch behavior. The same access sequence will be used during the attack to capture the fingerprint of the victim's prefetcher model.

C. Training a Classifier to Identify AoIs

Next, the attacker trains a classifier to identify AoIs given a sequence of prefetched addresses (the fingerprint). The training set consists of the collected fingerprints, each labeled to indicate which AoI was used to train the prefetcher model. Note that A1 is hosted on the same type of hardware (same type of epoch-trained ML prefetcher) as A2 and the victim tenant, so the classifier trained on A1 can be directly applied to fingerprints collected from A2. Attack training does not need to run on the same physical hardware as the victim.

D. Attacking the Victim

After training the classifier, the attacker collects fingerprints of the ML prefetcher trained by the victim's applications (❷ in Fig. 2). This is accomplished by tenant A2, which is co-hosted with the victim. The attacker issues memory accesses using the same sequence used in the fingerprinting step ❶. Once the prefetcher's behavior (the victim's fingerprint) is collected, it is sent to the attacker's classifier to determine whether an AoI was executed by the victim (❸ in Fig. 2).

We highlight that the same attack would not be as effective when applied to rule-based prefetchers, which are updated continuously on every memory access, meaning that entries containing information about past applications will likely be

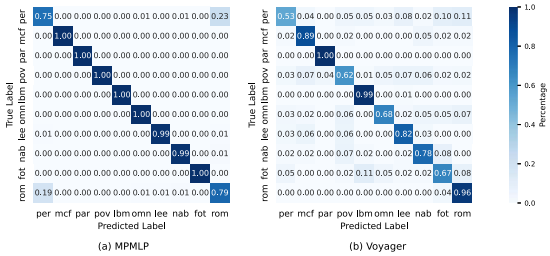

Fig. 3. Confusion matrices for Voyager and MPMLP.

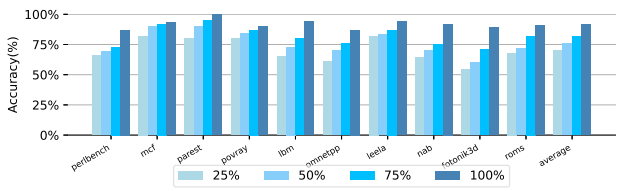

Fig. 4. The attack accuracy comparison between AoIs running in isolation and running with other programs.

overwritten during the process of fingerprint collection ("Accessing Sequence *S*" in Fig. 2). Experiments in Section V will show that to be the case.

V. Evaluations

A. Methodology

1) Simulation Platform: We simulate the execution of attacker and victim process on the gem5 simulator. Address Space Layout Randomization (ASLR) was turned on. We simulate a system with two cores, and a three-level cache hierarchy (32 KB/128 KB/2 MB).

2) Benchmarks: We select a set of benchmarks from SPEC 2017 as applications of interest (AoI): *500.perlbench, 505.mcf, 510.parest, 511.povray, 519.lbm, 520.omnetpp, 541.leela, 544.nab, 549.fotonik3d, 554.roms*. For multi-application experiments, applications are ran in parallel.

3) Target Prefetchers: We use two representative epoch-trained ML prefetchers in our case studies. *Voyager* [7] is a hierarchical LSTM-based prefetcher that predicts page and offset components separately using recent access history, with PC information provided as context. *MPMLP* [8] combines a Best Offset (BO) prefetcher [10] with an MLP-based prefetcher for complex multi-page patterns. The MLP takes recent access-history features as input and predicts the offset to prefetch, while page selection is obtained from a page transition table (PTT). In our study, we focus on the MLP component.

B. Fingerprinting Applications in Multi-Tenant Environments

We evaluate two prefetcher designs for our fingerprinting attack: *Voyager* [7] and *MPMLP* [8]. First, we generate a fixed set of random memory address sequences that is to be used to query each prefetcher. For each random sequence, we use different PCs to access these addresses, the PC value is also randomized and fixed. Next, we run each AoI 250–300 times on the target prefetchers, training a model after each run.

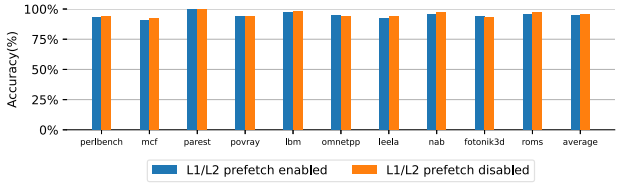

Fig. 5. The attack accuracy comparison between L1/L2 prefetchers enabled and disabled.

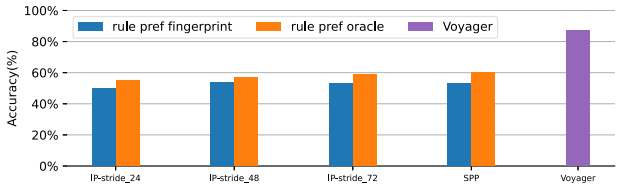

Fig. 6. Fingerprinting accuracy on rule-based prefetchers versus Voyager.

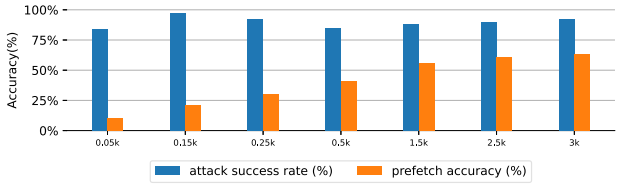

Fig. 7. The comparison between attack success rate and prefetch accuracy.

For that model, we obtain the model's fingerprint by issuing a set of memory accesses and observing which memory addresses (at cache line granularity) are prefetched by the prefetcher. The prefetched addresses can be observed and inferred by test loading page offsets allocated by the attacker. The code snippet for obtaining the fingerprint is shown in Listing 1.

We select AdaBoost [11] as the classifier. The inputs are the fingerprints we collected and the labels are the applications that the models were trained on. We use an 80/20 split of training and testing sets chosen randomly from our dataset, with the mix of backround applications not necessarily the same in the training and test sets. In a first experiment we train a multi-class AdaBoost, implemented based on [12], that includes all AoIs.

The classification accuracy of identifying the correct AoI is shown for each AoI as confusion matrices in Fig. 3 for MPMLP and Voyager. The overall fingerprinting attack accuracy is very high for MPMLP at 95% (std=0.0163, 95% CI=[0.9508 0.9649]), and slightly lower at 76% (std=0.0179, 95% CI=[0.7599, 0.7756]) for Voyager. Classification accuracy is uniformly high across all AoIs. Some AoIs, like *500.perlbench* and *554.roms* seem to exhibit more similar prefetch patterns, which leads the classifier to confuse them at a slightly higher rate, in both prefetcher designs.

In real systems, multiple processes could be running in parallel with the AoIs. To reproduce realistic LLC memory request traffic, in the next experiment, we run each AoI alongside multiple SPEC17 benchmarks and several randomly selected common Linux applications (such as *ls*, *pwd*, and *cd*). We then trained a binary AdaBoost classifier for each AoI. The classifier's task is to determine, given the fingerprint of a prefetcher model trained on mixed memory traffic, whether a certain AoI has been

Listing 1. Obtaining the Fingerprint

```
mem_access(void* p){...}
void get_fingerprint(uint8_t *ptr, int** sequences,
    int* fingerprint){ // sequences: NUM_SEQ *
    SEQ_LENGTH
    for(int i=0; i<NUM_SEQ; i++){
        fingerprint[i] = query(ptr, sequences[i]);
    }
}
int query(uint8_t *ptr, int* sequence){
    for(int i=0; i < PAGE_SIZE/BLOCK_SIZE; i++){
        for(int j=0; j < SEQ_LENGTH; j++){
            int addr = sequence[j] * BLOCK_SIZE;
            _mm_clflush(&ptr[addr]); // making sure
                LLC prefetcher sees the request
            mem_access(&ptr[addr]);
        }
        for(int j=0; j < SEQ_LENGTH; j++){
            if(j == SEQ_LENGTH - 1){
                ... // flush current page's blocks
            }
            ... // flush and load j-th block
        }
        uint64_t start = _rdtsc();
        mem_access(&ptr[i * BLOCK_SIZE]);
        uint64_t end = _rdtsc();
        uint64_t elapsed_cycles = end - start;
        if(elapsed_cycles < HIT_THRESHOLD){
            return i;
        }
    }
    return -1;
}
```

executed on that system or not. For Voyager, the classification accuracy is shown in Fig. 4. Here, we adjust the percentage of the AoI's memory traffic relative to the total memory traffic from 25% to 100%. Note that compared to the multi-class AdaBoost classifier, the binary AdaBoost classifier has higher accuracy: an average of 91% versus 76% for Voyager. The downside is that binary classifiers have to be trained separately for each application of interest. As the interference from other applications increases, the classification accuracy decreases slightly, but remains relatively high at 70% even for a mix of only 25% AoI.

Next, we evaluate a system in which multiple L1/L2 prefetchers are operating simultaneously. Fig. 5 shows the results when enabling both an IP-stride prefetcher and a Next-line prefetcher with Voyager prefetcher. As expected, the overall attack success rate remains mostly unchanged.

We also evaluate the attack on conventional continuously updated rule-based prefetchers: Intel IP-stride and SPP [1]. For IP-stride, we model a 24-entry table as reverse engineered in [4], and also evaluate 48 and 72 entry variants. As shown in Fig. 6, fingerprinting accuracy on IP-stride is only about 50%, rising to less than 55% for larger tables, far below Voyager's roughly 90%.

To estimate an upper bound for rule-based prefetchers, we additionally train the classifier directly on internal prefetch-table contents ('rule pref oracle' in Fig. 6). Even this oracle achieves less than 60% accuracy, suggesting that continuously updated rule-based prefetchers are substantially less susceptible to this attack than epoch-trained ML prefetchers.

Finally, we evaluate the sensitivity of the fingerprinting accuracy to the Voyager model's training accuracy. In order to change the model's prefetch accuracy we vary the number of training iterations per epoch from the baseline 3000 down to 50. Fig. 7 shows the results. We observe that fingerprinting accuracy degrades only slightly even as prefetch accuracy decreases from over 60% to less than 10%. This is likely because the prefetcher model quickly learns some dominant memory access patterns that are sufficiently specific to the AoI to drive accurate identification, even if those patterns are not yet sufficient to lead to accurate prefetch decisions.

VI. Conclusion and Future Work

This work showed that epoch-trained ML prefetchers can leak persistent application specific information, enabling fingerprinting attacks on previously executed applications. Future work will investigate defenses against such attacks. These could include per process tagging of memory requests that would enable, for example, excluding security sensitive contexts during prefetcher training.

References

[1] J. Kim, S. H. Pugsley, P. V. Gratz, A. N. Reddy, C. Wilkerson, and Z. Chishti, "Path confidence based lookahead prefetching," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2016, pp. 1–12.

[2] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo spatial data prefetcher," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 399–411.

[3] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2018, pp. 131–145.

[4] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2023, pp. 16–32.

[5] J. R. S. Vicarte et al., "Augury: Using data memory-dependent prefetchers to leak data at rest," in *Proc. IEEE Symp. Secur. Privacy*, 2022, pp. 1491–1505.

[6] B. Chen et al., "GoFetch: Breaking constant-time cryptographic implementations using data memory-dependent prefetchers," in *Proc. USENIX Conf. Secur.*, 2024, pp. 1117–1134.

[7] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 861–873.

[8] A. Asgari, A. Gunter, M. Saeidi, M. Lis, and P. J. Nair, "MPMLP: A case for multi-page multi-layer perceptron prefetcher," in *Proc. Workshop ML Comput. Archit. Syst.*, 2021, pp. 1–5.

[9] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas, "Everywhere all at once: Co-location attacks on public cloud FaaS," in *Proc. 29th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2024, pp. 133–149.

[10] P. Michaud, "Best-offset hardware prefetching," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 469–480.

[11] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Comput. Syst. Sci.*, vol. 55, no. 1, pp. 119–139, 1997.

[12] T. Hastie, S. Rosset, J. Zhu, and H. Zou, "Multi-class adaboost," *Statist. Interface*, vol. 2, no. 3, pp. 349–360, 2009.