A Cross-Platform Cache Timing Attack Framework via Deep Learning

Ruyi Ding[1], Ziyue Zhang[2], Xiang Zhang[1], Cheng Gongye[1], Yunsi Fei[1], Aidong A. Ding[2]
[1]*Department of Electrical and Computer Engineering,* [2]*Department of Mathematics*
Northeastern University, Boston, MA, USA
{ding.ruy, zhang.ziyue, zhang.xiang1, gongye.c, y.fei, a.ding}@northesatern.edu

***Abstract*—While deep learning methods have been adopted in power side-channel analysis, they have not been applied to cache timing attacks due to the limited dimension of cache timing data. This paper proposes a persistent cache monitor based on cache line flushing instructions, which runs concurrently to a victim execution and captures detailed memory access patterns in high-dimensional timing traces. We discover a new cache timing side-channel across both inclusive and non-inclusive caches, different from the traditional "Flush+Flush" timing leakage. We then propose a non-profiling differential deep learning analysis strategy to exploit the cache timing traces for key recovery. We further propose a framework for cross-platform cache timing attack via deep learning. Knowledge learned from profiling a common reference device can be transferred to build models to attack many other victim devices, even in different processor families. We take the OpenSSL AES-128 encryption algorithm as an example victim and deploy an asynchronous cache attack. We target three different devices from Intel, AMD, and ARM processors. We examine various scenarios for assigning the teacher role to one device and the student role to other devices, and evaluate the cross-platform deep-learning attack framework. Experimental results show that this new attack is easily extendable to victim devices and is more effective than attacks without any prior knowledge.**

***Index Terms*—Side-channel attacks, Deep learning, Computer architecture**

I. Introduction

Microarchitectural side-channel attacks (SCAs) have gained increasing attention in the security community. These attacks exploit programs' data-dependent footprints on the shared on-chip resources, including caches [1], translation look-aside buffer [2], and branch prediction unit [3], to infer the secret information. Cache timing is one of the most popular microarchitectural side-channels, and various attacks have been developed against cryptographic algorithms, including Prime+Probe [4], Flush+Reload [5], Evict+Time [6], and Flush+Flush [7]. They differ in the granularity of target microarchitecture and system setup for adversary and victim. Cache timing attacks threaten the confidentiality of many common cryptographic algorithms.

One limitation of existing cache timing attacks is most of them pose strict requirements on the attack set-up, where the spy has to synchronize with the victim during execution, which is an unrealistic assumption. Another limitation is the side-channel leakage is mostly a timing sample that indicates the cache status, which has inherent algorithmic noise and makes the attacks less effective. In this work, we address these two issues by proposing a new persistent cache monitor that runs concurrently to the victim execution in a non-invasive manner, without any synchronization, and also developing a deep-learning based cache timing attack to utilize the new cache timing traces generated by the monitor. The cache monitor is based on the Flush+Flush principle, and captures finer-grained memory access information during the entire victim execution in high-dimensional timing traces. These timing traces enable convolutional neural networks (CNN) for cache timing attacks, for the first time, while CNNs have been adopted in power side-channel attacks in prior works [8]. Moreover, we propose a cross-platform cache timing attack framework by profiling one common reference device to build a general teacher model. When retraining it on other target victim devices, the adversary can use fewer timing traces to attack.

The contributions of our work include:

- We design a persistent cache monitoring tool for common processor families such as Intel X86, AMD X86, and ARM. The monitor runs concurrently with the victim program and achieves good observation resolution.
- We propose a non-profiled differential deep learning attack (DDLA) to exploit cache timing traces. The method takes the traces as inputs and outputs key-related predictions without prior knowledge or victim device profiling.
- We propose a cross-platform attack by pretraining a reference model. There are two advantages of the methodology: portability - one model can be transferred to many other models for distinctly different victim devices; effectiveness - the number of traces required for the victim devices can be smaller than directly DDLA.

The rest of the paper is organized as follows. Section II provides background information on cache timing attacks, deep learning and cross-platform attack. We illustrate our proposed cache monitor and cross-platform attack framework in Section III. We describe our experiments over three different devices and present the experimental results in Section IV. Finally, we discuss countermeasures and future work in Section V.

II. Background

This section presents the background on cache timing attacks, deep learning for SCAs, and cross-platform pretraining.

This work was supported in part by the National Science Foundation under grants SaTC-1929300 and IUCRC-1916762, and CHEST (Center for Hardware Embedded System Security and Trust) industry fund.

A. Conventional Flush+Flush Cache Timing Attack

Cache timing attack [9] typically consists of an online phase and an offline phase. During the online phase, the attacker sets up a monitor to probe the shared cache status and derives memory access information of the victim application. The common cache monitors include Prime+Probe [4], Flush+Reload [5] and Flush+Flush [7], and the spy has to synchronize with the victim so that the spy can capture whether a certain cache line/set has been accessed by the victim or not, i.e., generating binary information. Conventional cache attacks employ statistical analysis tools to process the binary information, e.g., a hypothesis test is run to find correct keys which yields the smallest discrepancy between the prediction and measurements.

In this paper, we take the 16-byte Electronic Code Book (ECB) mode T-table implementation of AES encryption from OpenSSL [10] as an example. The algorithm consists of ten iterative rounds and each round includes four look-up operations on each of the four T-tables. Each T-table is at a size of 1KB - occupying 16 cache lines with the normal cache line size of 64 bytes. Due to algorithmic complexity, normally the last-round is being attacked to retrieve the secret key byte by byte. The statistical tool attributes memory accesses monitored to the last-round, even though it is actually a summary for the entire execution, resulting in inherent high noise in the coarse-grained cache timing information. The last-round table look-up operation can be expressed as $c^i = Te_k[s^j] \oplus k^i$, where c^i is the i^{th} byte of the output ciphertext, k^i is the i^{th} byte of the last-round key, s^j is the j^{th} byte of the last-round input state. If information about s^j can be derived from the cache timing information, the key byte k^i can be recovered with the c^i.

The recent Flush+Flush cache timing attack [7] utilizes the timing difference an X86 user-level instruction, `clflush`, experiences due to the state of the target cache line. Given a virtual memory address, the instruction flushes the memory block with this address from all on-chip caches. If the memory block is in caches, it takes a longer time to invalidate the cache lines. Otherwise, shorter time. The counterpart instruction on ARM processors is `dc civac` [11]. In Section III-B, we develop a Flush+Flush cache monitor and discover a different cache timing side-channel, which gives more clear memory access information.

B. Deep Neural Network and Model Pre-training

Deep neural network (DNN) parameterizes a non-linear function from a given input $\boldsymbol{X}$ to an output $\boldsymbol{y}$. Two widely used DNN structures are *Multi-Layer Perceptrons* (MLP) [12] consisting of fully-connected neurons, and *Convolutional Neural Network* (CNN) [13] consisting of convolutional, pooling, and dense layers. For a classification task, the output layer will be connected to a softmax layer, which normalizes the output scores and computes the probability distribution of each class. With a training dataset, a selected DNN model (with structure and hyperparameters set) will be fed with inputs and labels, and the parameters (weights, biases, kernels, etc.) will be iteratively trained. During training, a *loss function* is used to measure the error between the network outputs and the labels. A trained DNN model can be used on the testing dataset for inference like classification and object detection.

Model pre-training is a technique to initialize a neural network for a new task by using parameters from a well-trained model. Pre-training is proven to improve the model generalization when the target dataset is smaller than the reference dataset [14]. In Section IV-D, we demonstrate a cross-platfrom cache attack framework with pre-training which reduces the number of traces usage from victim devices.

C. Deep Learning Based Side-Channel Attacks

Deep learning methods are effective in capturing the dependency between device power consumption or electromagnetic emanation and the sensitive data value, and have been employed in side-channel power and EM attacks [15], [16]. Generally, there are two classes of deep learning based SCAs:

- `Profiling SCAs` are the most powerful attacks because the adversary can profile the target device (or an identical copy) with known keys and labels. The DNN model is trained to fit the device directly, and will be used for attacking the same device or similar devices when the key is unknown.
- `Non-profiling SCAs` are weaker as the adversary only has access to the leakage dataset on the target device with an unknown key, without prior knowledge about the device. The recent work [16] introduces Differential Deep Learning Analysis (DDLA) for power side-channel attacks under a non-profiling setting. As there are no known labels, pseudo-labels can be used where each assumes a hypothesized key value, and DNN models will be trained accordingly. Some distinguishers are used to pick the best-performing model, which yields the correct key value.

So far the deep learning models have been used for power side-channel attacks and realize cross-device SCA [17]. There is only one prior work [18] that uses DNN models to quantify the effectiveness of traditional cache timing attacks rather than improving real attacks. Our cross-platform cache attack framework has two contributions: a persistent cache monitor to generate finer-grained timing traces, and a non-profiling deep transfer learning attack methodology, which can apply to many different victim devices, even with a different Instruction Set Architecture (ISA) from the reference device.

III. Cross-Platform Deep Learning Cache Attack

In this section, we describe our cross-platform cache timing attack framework based on deep learning.

A. Threat Model

In our attack model, the victim is a program that operates on secret information, e.g., ciphers from open-source libraries like OpenSSL with a private key. The goal of the adversary is to retrieve the secret key. On any device, a spy program runs a cache monitor concurrently with the victim application, leveraging the commonly available multi-core or hyper-threading environments. The spy program is in user mode and has no direct interaction with the victim, except for sharing microarchitectural resources including last-level caches. Prior

asynchronous cache attacks [19] utilize cache monitors based on "Evict+Timing" or "Prime+Probe", and rely on special distributions of the plaintexts. Our cache monitor is based on "Flush+Flush," and there is no requirement for the plaintexts or ciphertexts distribution.

In order to perform cross-platform attacks, we need a common reference device and a set of target (victim) devices. The reference device is profiled in a white-box fashion, i.e., the key is arbitrarily set and known while cache timing traces are obtained. When training DNN models for the reference device, the labels are derived with the known key value. For a target device, it is the realistic black-box attack scenario - a dataset of cache timing traces are collected from victim execution, while the key used is unknown and is for the attacker to recover.

B. Persistent Cache Monitor

We set up a persistent cache monitor based on Flush+Flush [7], as shown in Algorithm 1. The spy runs iteratively, and in each iteration it flushes a target cache line followed by reading the system time stamp counter (on X86, the instruction is `rdtsc` or `rdtscp`). The execution time of each flush instruction can be calculated by the differential between the timestamps of the current iteration and the prior one. The output of the cache monitor is a timing trace, consisting of points of timing values. Note to deal with out-of-order execution, we use the instruction `rdtscp` which has weak ordering effect, and the array writing instruction also has serialization effect.

Algorithm 1: Persistent Cache Monitor

Input : max number of iterations to explore: M_{it}
Output: cache timing trace: DT

for $iter$ = 0 *to* M_{it} **do**
  Flush(TargetCL);
  $T[iter]$ = Read(TimeStampCounter);
end
for $iter = 1$ *to* M_{it} **do**
  $DT[iter-1]$ = $T[iter]$ - $T[iter-1]$
end
return DT

Assuming the target cache line is also used by the victim, with the prior synchronous "Flush+Flush" cache monitor, the spy would observe a 10-cycle timing difference depending on whether the target cache line exists in caches or not. However, we observe a much larger timing difference, around 200 cycles, for our monitor setup. We attribute this large timing difference to the concurrent (asynchronous) setup, and discover a new timing side-channel. For X86 Intel processors, the last-level cache is inclusive and shared by multiple cores. When the victim runs AES execution on one core while the spy runs continuous flushing instructions on another core, they issue memory access instructions simultaneously (the 40 lookup operations on one T-table of AES and the many instructions flushing one selected T-table cache line). However, their requests have to enter a common bus queue before accessing the L3 cache. If a flush instruction (from the spy) is after a load instruction (from the victim) while the load is an L3 cache miss, the flush instruction has to wait for the load instruction to fill the L3 cache, i.e., experiencing a cache miss delay. For other architectures such as AMD and ARM, their last-level caches are non-inclusive while similar timing side-channel still exists. When the victim core is loading a cache line, the flush instruction of the spy (from another core) will hold in the write buffer temporarily, i.e., it will also experience a cache miss delay.

C. Differential Deep Learning Analysis on Cache Timing Traces

We adopt the Differential Deep Learning Analysis (*DDLA*) strategy, originally proposed for power SCA [16], and apply it onto our cache timing traces generated by the persistent cache monitor. For a given dataset of cache timing traces (each corresponding to an encryption run with a plaintext/ciphertext under the same unknown key), to feed them as inputs to a chosen DNN model, the labels for the models are unknown as the label is key value dependent. The attacker has to guess all the possible key values (256 for a key byte), and trains 256 DNN models with the pseudo labels, $\mathbf{h}_{k_g}$, calculated with a key hypothesis k_g from the key space $\mathcal{K}$. By monitoring any cache line of a T-table, we can retrieve corresponding four key bytes which operate on the T-table. Assuming for m^{th} byte, $c^m = Te_k[s^j] \oplus k^m$, with the observed c^m value and a guessed k^m value, the index s^j is calculated and used to determine if it falls into the index range for the target cache line. The label $h^m_{k_g}$ indicates whether this operation has accessed the chosen target cache line or not, with $h^m_{i,k_g} = 1$ meaning access and $h^m_{i,k_g} = 0$ meaning no-access (Line 2 of Algorithm 2). As the key space for one-byte is $\mathcal{K} = [0, 255]$, we build 256 models correspondingly (Line 3). We use the training loss as the distinguisher function and select the key byte value (model) with the lowest loss (Line 6).

Algorithm 2: Differential Deep Learning Analysis on Cache Timing Traces

Input : Cache timing traces $\boldsymbol{X} := \{\boldsymbol{x}_i\}$
  Corresponding ciphertexts bytes $\boldsymbol{C} := \{c^m_i\}$
  A key byte search space $\mathcal{K}$.
  $i \in [1, N]$ is index of traces
  $m \in [0, 15]$ denotes m^{th} key byte
Output: Predicted correct m^{th} key byte $\hat{k^m_c}$

1 **for** *each* $k_g \in \mathcal{K}$ **do**
2 Calculate $\boldsymbol{H}^m_{k_g} := \{h^m_{i,k_g}\}$ with k_g and $c^m_i \in \boldsymbol{C}$ for all traces
3 Train a DNN model $\mathcal{M}_{k_g}$ on $\{\boldsymbol{X}, \boldsymbol{H}^m_{k_g}\}$
4 Compute the testing loss $L^m_{k_g,test}$ of model $\mathcal{M}_{k_g}$
5 **end**
6 **return** $\hat{k^m_c} = argmin(L^m_{k_g,test})$ *for* $k_g \in [0, 255]$

D. Cross-Platform Cache Timing Attack

The cross-platform cache timing attack consists of two phases: a profiling phase on the reference device and an attack

phase on the victim device. In the profiling phase, a common reference device is set with known key values. The cache timing traces are used to train one teacher model, which fits the reference device and its traces well. In the attack phase, for the victim device with an unknown key, we obtain another set of cache timing traces. Then a DDLA is conducted to recover the correct key. However, the models in DDLA are not built from scratch. Instead, they are student models with the same structure of the teacher model and initialized by its parameters.

Profiling Phase: During profiling, the reference device is a white box under the control of the attacker with a known private key. For AES encryption, by monitoring one cache line of one T-table, the attacker can build a teacher model for one related key byte, which is applicable to the other three related bytes as well. For example, if we monitor the first cache line of Te0, the related four key bytes are {2, 6, 10, 14}. We choose to target the 2nd key byte, with the involved operation as $c^2 = Te_0[s^{10}] \oplus k^2$. If s^{10} falls into [0,15], this operation accesses the target cache line. We denote the cache timing traces from the reference device as $\boldsymbol{X}_t$. For a trace, we denote the label $\boldsymbol{y}_t \in \{0, 1\}$, which indicates if the last-round operation of $c^2 = Te_0[s^{10}] \oplus k^2$ has indeed looked up the monitored cache line. A teacher CNN model can be trained to map $\boldsymbol{X}_t$ onto $\boldsymbol{y}_t$, denoted as $f_t(\boldsymbol{X}_t, \boldsymbol{y}_t)$. Although the teacher model is built for the 2nd key byte, it also apply to the other three key bytes {6, 10, 14} in attack with the corresponding ciphertext bytes plugged in.

Attack Phase: In the attack phase, we are facing a victim device with a different architecture and an unknown encryption key. We would like to explore knowledge gained in the teacher model to help build student models for the victim device.

With the dataset of victim cache timing traces, $\boldsymbol{X}_s$, instead of performing *DDLA* directly, as shown in Algorithm 2, we incorporate pre-trained teacher models. When training each DNN model $\mathcal{M}_{k_g}$ (Line 3 of Algorithm 2), the model is initialized with the same structure and parameters as the teacher model $\mathcal{M}_t$ to get a student model $\mathcal{M}_{s,k_g}$. During training, we freeze middle layers and the end layer and only retrain beginning layers based on the victim dataset $\{\boldsymbol{X}_s, \boldsymbol{y}_{s,k_g}\}$. The algorithm for the attack phase only needs to update Line 3 of Algorithm 2 to be: Transfer the teacher model $\mathcal{M}_t$ to a student DNN model $\mathcal{M}_{s,k_g}$ with $\{\boldsymbol{X}_s, \boldsymbol{H}^m_{s,k_g}\}$. This is anticipated to require fewer traces than the direct *DDLA*. How to divide the layers of teacher model for freezing and retraining is to balance direct transferring due to similarity between the datasets and devices and learning due to specific features of victim dataset.

IV. Experiments

In this section, we present experimental results of our cross-platform cache timing attacks on three families of devices, Intel, AMD, and ARM CPUs.

A. Trace Collection

Considering the similarity of cache structures on different processors (e.g., common cache line size is 64 bytes), we believe cache side-channel information gathered by a cache monitor on different CPU platforms will be similar to each other. We pick three platforms: a 4-core ARM Cortex-A72 on a Rasberry Pi-4 board, a 12-core AMD Ryzen 9 3900X on a desktop, and a 4-core Intel i7-7700 processor on a laptop. The monitors all run 200 flush instructions continuously for each victim execution, which is an OpenSSL AES ECB encryption. The spy and the victim are pinned to different cores. Fig. 1 presents three cache timing traces collected from the three devices, where the victim is running with the same plaintext and key and the spy is monitoring the same cache line. The selected cache line is accessed by the AES execution three times, in $3^{rd}, 4^{th}$, and 10^{th} round. For each timing traces, we see a number of high points (corresponding to the number of accesses) and many low points (meaning no-access) interleaving with the high points. The ARM trace and Intel trace have the correct number of high points, while the AMD trace is less accurate - only 2 high points can be distinguished. Our hypothesis is the first two accesses are too close and the monitor only captures one. The timing side-channel also varies on the three architectures. For ARM, the high and low point are 300 cycles versus 100 cycles - for Intel, 300 versus 70 - and for AMD, 550 versus 300.

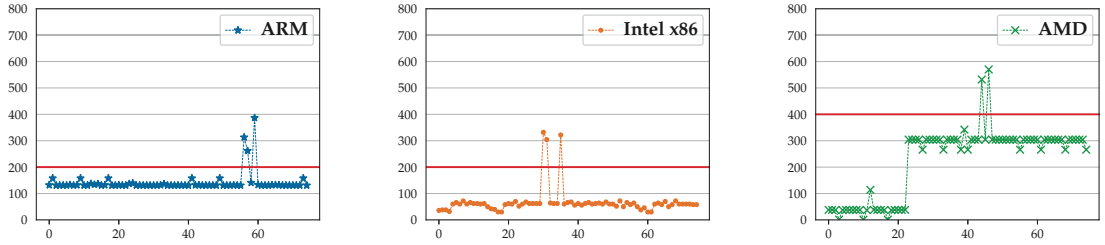
To compare the different resolutions of the monitors on the three devices, we run $10,000$ memory accesses continuously and vary the number of cycles (NOPs) between two loads. The result is shown in Fig. 2, where the X axis is the time interval between two loads, and the Y axis shows how many high points (load accesses) have been captured by the monitor. We can see that the monitor on AMD runs the slowest, causing the low monitoring accuracy as shown in Fig. 1 (c).

B. Building DNN Models

For unaligned cache timing traces, convolutional neural network is suitable. We train a CNN to map the traces $\boldsymbol{X}_t$ to the true last-round access (with the key known) by the target operation $\boldsymbol{y}_t$. With grid search optimization, we choose the structure of CNN: two convolutional layers, and one fully connected layer. The training dataset is balanced and has 5000 traces with a length of 200 points. We use 80% samples for training and 20% for testing. The model is compiled with optimizer ADAM and uses cross-entropy as the loss function. The training losses and accuracies for the three reference models are shown in Fig. 2. Among the three processors, Intel X86 has the best performance with a test accuracy of 89.2%. ARM has an accuracy of 83.5% and AMD has the lowest accuracy, 73.4%, due to the low resolution shown in Fig. 2(c). Similar observations are made for the training losses.

C. Non-profiled Direct DDLA Attacks

In the realistic scenario of attacking a target device, it has to be non-profiled as the key values are unknown and yet to be retrieved. We apply DDLA directly in the attack phase for each platform, i.e., train multiple models each with pseudo-labels according to a key candidate. We use the model loss values as the distinguisher to rank the key candidates (models). Taking Intel X86 processor as an example, we target the 2nd key byte. Fig. 4 shows the key distinguishing result for it. We use the same dataset as IV-B while assuming the key is unknown. The plot depicts the training losses of the 256 models each for



(a) ARM trace (b) Intel trace (c) AMD trace

Fig. 1: Timing traces from different platforms. The red lines are thresholds to classify high (victim access) and low (no-access)

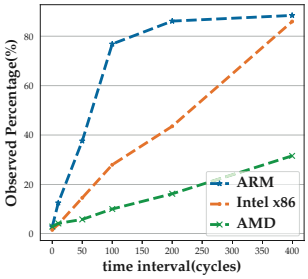


Fig. 2: Cache monitor resolution for three platforms

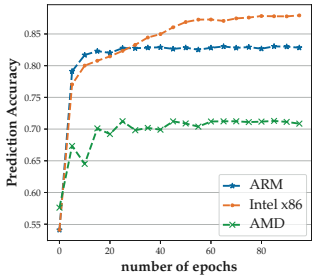


Fig. 3: Model training and accuracties for different platforms

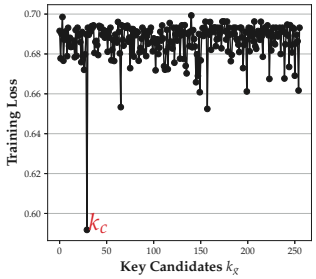


Fig. 4: Direct DDLA for Intel Traces (distinguisher for 2^{nd} byte)

a hypothesized key byte value, where the marked correct key value corresponds to the lowest model loss. The plot shows that only the deep learning model for the correct key byte value is fitting to leakage signals, while other models for incorrect key byte values are fitting to random noises. Using the same traces but different pesudo labels, the same process can be applied to other three key bytes, $6^{\text{th}}, 10^{\text{th}}, 14^{\text{th}}$.

D. Cross-Platform Attacks

In general, distinctively different systems and environments may make cross-platform cache attacks hard to realize. In our attacks, the model outputs, access or no-access of a target last-round operation on a selected cache line, are the same for different platforms due to one important fact: the cache line size is the same. The pre-trained parameters of the reference model for one device increases the model convergency and reduces the number of training traces for other victim devices.

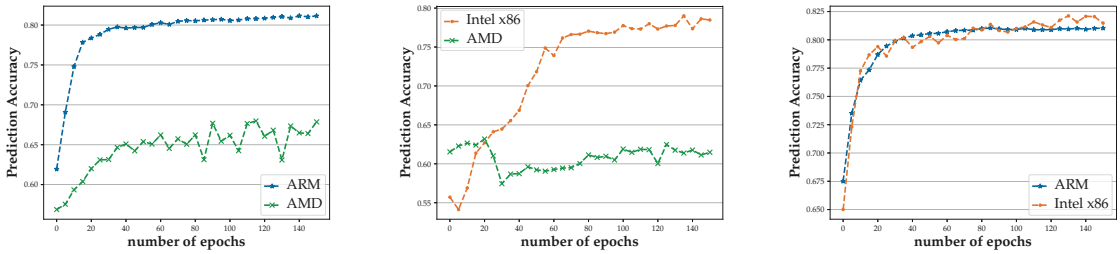
To analyze the performance of cross-device learning, we collect 200 unlabelled timing traces for each target device. Note that this number (for the victim device) is much smaller than what we use in IV-B for the reference device. Fig. 5 shows the testing accuracies of the best transferred student model (with only the first convolutional layer retrained), corresponding to the correct key value in DDLA, where one reference model is transferred for the other two platforms. The transferring between ARM and Intel has the best accuracy, over 80%. From any reference device to AMD CPU, the student models only achieve accuracies around 65%. This is because the teacher model is trained on more clean devices (with lower noise), and cannot handle the high system noise of AMD timing traces well. Interestingly, when the teacher model for the AMD CPU is transferred for the other two devices, the student models achieve high accuracies over 80%.

For a victim device, we consider and compare three attacks: the traditional statistical Flush+Flush attack, a direct DDLA attack, and a cross-platform attack that fine tunes a pre-trained model from a reference device. Our victim device is Intel CPU, and the reference device is ARM for cross-platform attack. In Fig 6, we plot change of the correct key rank (including both the average and the variance) along the number of traces for the three attacks. For the tradition statistical attack method with a synchronous "Flush+Flush" monitor, it requires more than $50,000$ traces to achieve good attack performance (the rank goes down to 0). With our persistent cache monitor and DDLA, the number of traces decreases to $1,600$. Furthermore, by cross-platform pre-training and parameter tuning, the attacker only needs 500 victim traces to retrieve the correct key.

V. Discussions and Conclusions

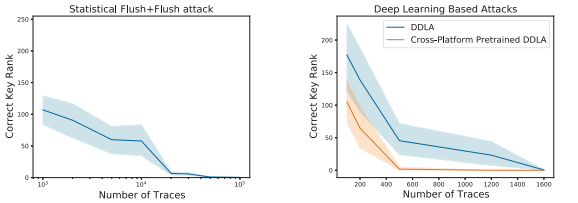
This work demonstrates that enabled by the persistent cache monitor, deep learning can be used for cache timing attacks, and cross-platform attacks also work. To the best of our knowledge, this is the first application of deep neural networks and cross-platform pre-training in cache timing attacks.

To protect systems against the deep learning cache timing attacks, both hardware and software techniques can be considered to fail the cache monitor or obfuscate cache access patterns. As one prerequisite for Flush+Flush attack is



(a) From Intel to Others (b) From ARM to others (c) From AMD to others

Fig. 5: Testing accuracies of two transferred student models from a teacher model, with a different reference device



(a) Statistical method (b) DDLA and transfer attack

Fig. 6: Average rank of correct keys for different attacks

shared memory, the OS can disable shared memory, with some performance loss. Alternative cache monitors, such as those based on Prime+Probe, can still be employed to monitor victim execution. However, Prime+Probe is coarser-grained, targeting cache set instead of the cache line, and it would be running much slower and the monitoring resolution may not be high enough to capture useful memory access details.

As our cache monitor tracks the entire execution, obfuscation (randomization) of the memory or cache access pattern of victim application makes it hard to monitor useful information leakage. The prior work that randomizes the locations of sensitive data in memory [20] will make the monitor lose the target. Other secure cache architecture designs, like RCache and PCache [21], map memory blocks to random cache sets. However, our flush-based monitor may still work as it only requires virtual addresses, and the hardware memory controller automatically maps the memory address to the cache index.

As the deep learning framework is sensitive to outliers and monitor resolution, we can have more specific countermeasures. Random memory accesses can be added into the victim (AES), and the DDLA model accuracy can reduce significantly (for AES, from 89.2% to 58.6%). Also, a platform with low-resolution timers and slow flush instructions (like AMD processors) can have better resistance to the deep learning attack.

The future work includes investigating other cache monitors such as Prime+Probe based ones, drastically different cache structures (e.g., different cache line size), other non-cryptographic victim applications, and also evaluation of various countermeasures.

REFERENCES

[1] O. Aciiçmez, "Yet another microarchitectural attack: exploiting i-cache," in *Proc. ACM Workshop on Computer Security Architecture*, Nov. 2017.

[2] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks," in *USENIX Security*, Aug. 2018.

[3] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proc. ACM Symp. on Information, Computer & Communications Security*, 2007.

[4] C. Percival, "Cache missing for fun and profit," in *In Proc. of BSDCan*, 2005.

[5] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symp.*, Aug. 2014.

[6] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Topics in Cryptology*, D. Pointcheval, Ed., 2006, pp. 1–20.

[7] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+flush: A fast and stealthy cache attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2016, pp. 279–299.

[8] G. Zaid, L. Bossuet, A. Habrard, and A. Venelli, "Methodology for efficient CNN architectures in profiling attacks," *IACR Trans. on Cryptographic Hardware & Embedded Systems*, vol. 2020, no. 1, 2020.

[9] S. Briongos, P. Malagón, J.-M. de Goyeneche, and J. M. Moya, "Cache misses and the recovery of the full aes 256 key," *Applied Sciences*, vol. 9, no. 5, 2019.

[10] "OpenSSL - Cryptography and SSL/TLS Toolkit," https://www.openssl.org/.

[11] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Security Symp.*, 2016.

[12] D. Svozil, V. Kvasnicka, and J. Pospichal, "Introduction to multi-layer feed-forward neural networks," *Chemometrics and intelligent laboratory systems*, vol. 39, no. 1, 1997.

[13] S. Lawrence, C. L. Giles, A. C. Tsoi, and A. D. Back, "Face recognition: A convolutional neural-network approach," *IEEE Trans. on Neural Networks*, vol. 8, no. 1, 1997.

[14] D. Hendrycks, K. Lee, and M. Mazeika, "Using pre-training can improve model robustness and uncertainty," in *Int. Conf. on Machine Learning*, 2019.

[15] L. Masure, C. Dumas, and E. Prouff, "A comprehensive study of deep learning for side-channel analysis," *IACR Trans. on Cryptographic Hardware & Embedded Systems*, 2020.

[16] B. Timon, "Non-profiled deep learning-based side-channel attacks with sensitivity analysis," *IACR Trans. on Cryptographic Hardware & Embedded Systems*, 2019.

[17] D. Das, A. Golder, J. Danial, S. Ghosh, A. Raychowdhury, and S. Sen, "X-deepsca: Cross-device deep learning side channel attack," in *Proc. Annual Design Automation Conf.*, 2019.

[18] T. Zhang, Y. Zhang, and R. B. Lee, "Analyzing cache side channels using deep neural networks," in *Proc. Annual Computer Security Applications Conf.*, 2018.

[19] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006.

[20] Z. H. Jiang, Y. Fei, A. A. Ding, and T. Wahl, "Mempoline: Mitigating memory-based side-channel attacks through memory access obfuscation," *IACR Cryptol. ePrint Arch.*, vol. 2020, no. 653, 2020.

[21] Z. Wang and R. B. Lee, "A novel cache architecture with enhanced performance and security," in *IEEE/ACM International Symp. on Microarchitecture*, 2008, pp. 83–93.