Ran$Net: An Anti-Ransomware Methodology based on Cache Monitoring and Deep Learning

Xiang Zhang*, Ziyue Zhang*, Ruyi Ding, Cheng Gongye, Aidong Adam Ding and Yunsi Fei
Northeastern University
Boston, Massachusett, United States
{zhang.xiang1,zhang.ziyue,ding.ruy,gongye.c,a.ding,y.fei}@northeastern.edu

ABSTRACT

Ransomware has become a serious threat in the cyberspace. Existing software pattern-based malware detectors are specific for certain ransomware and may not capture new variants. Recognizing a common essential behavior of ransomware – employing local cryptographic software for malicious encryption and therefore leaving footprints on the victim machine's caches, this work proposes an anti-ransomware methodology, Ran$Net, based on hardware activities. It consists of a passive cache monitor to log suspicious cache activities, and a follow-on non-profiled deep learning analysis strategy to retrieve the secret cryptographic key from the timing traces generated by the monitor. We implement the first of its kind tool to combat an open-source ransomware and successfully recover the secret key.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**; *Software security engineering*.

KEYWORDS

Anti-Ransomware, Cache Timing Analysis, Deep Neural Network

ACM Reference Format:
Xiang Zhang*, Ziyue Zhang*, Ruyi Ding, Cheng Gongye, Aidong Adam Ding and Yunsi Fei. 2022. Ran$Net: An Anti-Ransomware Methodology based on Cache Monitoring and Deep Learning. In *Proceedings of the Great Lakes Symposium on VLSI 2022 (GLSVLSI '22), June 6–8, 2022, Irvine, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3526241.3530830

1 INTRODUCTION

Ransomware incidents have become increasingly prevalent targeting the Nation's government entities, healthcare systems, schools, and critical infrastructures [1, 2]. Victims have to pay ransom to hackers so as to obtain the key to unlock their valuable files, which has been "locked" by the ransomware distributed by attackers after the victim system is infected.

Pattern-based anti-virus malware detectors recognize instruction sequence [5] or function calls [3, 6] employed by malwares. However, such detection can be evaded by obfuscated malwares with no more recognizable patterns [7, 11, 13]. Different from prior work that tries to detect the early stage of ransomware infection so as to prevent its execution, our work focuses on detecting a later stage, the most essential step of ransomware execution - encryption via the local symmetric cryptographic software with a secret key, which is erased from the system before extortion. Our work targets recovering the key from system activities collected by a specially designed monitor. This is the key insight of our work.

In this paper, we propose Ran$Net - an anti-ransomware methodology built upon system cache monitoring and deep learning techniques. The cache monitor collects critical data access information of the encryption algorithm execution, leaking the key information. We then design a deep learning tool to analyze the timing traces and retrieve the secret key, turning the power of conventional side-channel cache timing attacks against ransomware.

The most common cache timing attacks include Prime+Probe [9], Flush+Reload [14], Evict+Time [8], and Flush+Flush [4] with different side-channel granularity. However, the prior cache timing attacks pose strict requirements on the attack set-up, where the monitor has to synchronize with the victim in terms of program execution, an unrealistic assumption for independent monitor and victim (ransomware). Typically a summary timing sample is generated for each run. Our cache monitor leverages the common multi-core or multi-threading environment in modern computing platforms, and runs concurrently to other applications as a system watchdog. When the victim application is executed, the monitor is triggered to monitor the cache state. Our monitor collects much finer-grained memory access information of the victim during its execution, a trace with many timing samples, which facilitates more powerful attacks based on deep learning techniques.

The contributions of the proposed Ran$Net include:

- We design a cache timing monitor for ransomware. The two-level monitor first tracks activation of the victim software, then runs concurrently in parallel to the ransomware execution and achieves good observation resolution.
- We propose a non-profiled differential deep learning analysis (DDLA) strategy for secret key retrieval from the cache timing traces, without any prior knowledge of the victim device. A convolutional autoencoder is first used to extract leaky features from the timing traces, followed by multi-layer perceptron (MLP) networks to predict cache line access.

The rest of the paper is organized as follows. Section 2 provides background on cache monitors, deep learning attacks, and non-profiled DDLA strategy. We illustrate our proposed cache monitor

* indicates equal contribution.

GLSVLSI '22, June 6–8, 2022, Irvine, CA, USA

ACM ISBN 978-1-4503-9322-5/22/06…$15.00
https://doi.org/10.1145/3526241.3530830

for ransomware in Section 3. We describe our experiments for running Ran$Net and present results in Section 4. Finally, we discuss future work in Section 5.

2 BACKGROUND

This section presents the background on cache monitoring, ransomware, and deep neural networks for side-channel attacks.

2.1 Cache Monitor

The Ran$Net monitor is built upon Flush+Flush cache timing attack [4], which claims that a special X86 instruction - `clflush` - experiences different timing depending on the state of the target cache line. Given a virtual memory address, this instruction flushes the memory block that contains the given address from all the on-chip caches. If it does exist in the cache hierarchy, all the cache lines at different levels will be invalidated (flushed), taking longer. If it is not in caches (has been flushed from the cache hierarchy before), it takes a shorter time. The time difference is about 9-12 cycles, depending on processor architectures. Although the timing side-channel is not very strong, the Flush+Flush monitor incurs less cache misses compared to Flush+Reload attacks and therefore can run faster. We adapt the Flush+Flush cache monitor and improve the distinguishability and resolution of the side-channel, with details given in Section 3.2.

Our new monitor observes the cache access information *during* the execution, rather than only the cache state at the end of the execution. Therefore, the monitor generates timing traces with much more detailed finer-grained information, i.e., a sequence of many accesses and nonaccesses, rather than a simple summary of whether the cache line has been accessed or not. Such traces facilitate deep learning models for attacks, which can be much more powerful than the conventional statistical cache timing attacks.

We pick RAASNet as an ransomware example [12], a demonstration written in Python. After the ransomeware infects the system, targeted files are encrypted by AES-128 in CBC mode. RAASNet depends on `Pycryptodome`, one popular cryptographic library in Python, which is shared on the victim system. We also use `CONDA` to manage our Python environment, dependency and packages.

2.2 Deep Neural Network for Side-channel Attacks

Key-retrieval side-channel attacks can be considered as a classification problem which neural networks are privileged to address. Given an input $X \in \mathcal{X}$, a neural network aims to parameterize a function $f_\theta : \mathcal{X} \rightarrow S^{|\mathcal{H}|}$ that computes the prediction score vector $\mathbf{S} = (S_1, ..., S_{|\mathcal{H}|})$ for all classes $H \in \mathcal{H}$, and the classification problem can be well solved when the neural network picks out the right class with the highest score. A neural network is composed of three types of layers: an *input layer* (the first layer); an *output layer* (the last layer, whose output is the prediction score vector $\mathbf{S}$); and hidden layers in between. Each layer processes a vector product between its input and a vector of *trainable parameters* contained in its computational units called *neurons*. A *loss function* that measures the classification error of f_θ over the training set is minimized through forward-backward propagation.

2.3 Non-profiled Differential Deep Learning Analysis on Cache Timing Leakage

DNNs have been used in profiling power side-channel attacks, where a decoy system with a known key is used for training. We turn the power of side-channel attacks against ransomware execution, where in realistic scenarios, there is no such knowledge and therefore non-profiled SCA for anti-ransomware is desired. We follow the strategy of Differential Deep Learning Analysis (DDLA) presented in CHES2019 for power side-channel attacks [10], and propose augmentations to make the attack practical for our cache timing traces obtained by the cache monitor.

Algorithm 1: Differential Deep Learning Analysis

Input : CAE latent feature set $L^A = \{l_i\}_{1 \le i \le N}$;
Corresponding ciphertext bytes $(c_i^m)_{1 \le i \le N}$;
An MLP structure $\mathcal{M}$;
Output : The correct key k_c.
Split L^A into a training set L^{A_1} and a validation set L^{A_2};
for $k \in \mathcal{K}$ **do**
  Initialize trainable parameters of $\mathcal{M}$;
  Compute the series of labels $(H_{i,k})_{1 \le i \le N}$;
  Perform an MLP training on the training set $S^{A_1} = (L^{A_1}, \{H_{i,k}\}_{1 \le i \le N_1})$: $\mathbf{MLP}(\mathcal{M}_k, S^{A_1})$ with cross-entropy loss function $\mathbf{CE}_k(S^{A_1})$
  Record the validated loss $\mathbf{CE}_k(S^{A_2})$ of model $\mathcal{M}_k$
end
Return : $\hat{k}_c = argmin(\mathbf{CE}_k(S^{A_2}))$ for $k \in \mathcal{K}$

The DDLA artificially splits the measurement dataset (timing traces) into a training subset and a validation subset. While there is no knowledge of correct labels on training subset, DDLA assumes one k_g value at a time and produces pseudo labels over the training subset and builds a DNN model under this specific key hypothesis. This process is repeated for all candidates in the key space $\mathcal{K}$. For one key byte, 256 DNN models have to be built in training phase, while the prior profiled attack only builds one model. A distinguisher is used to select the optimal DNN model over the validation dataset, which corresponds to the correct key value.

Since training 256 convolutional neural network (CNN) models is computation-intensive, we propose to reduce the complexity by first training a common convolutional part in an unsupervised way, using a convolutional autoencoder (CAE), followed by training 256 fully connected multi-layer perceptron (MLP) networks. The CAE is trained to reconstruct the signal traces, where the low-dimensional features in the bottleneck layer of the well-trained autoencoder are used as inputs for the follow-on MLPs, which are easier to train than CNNs. Such extracted features should contain most of the key-related leakage information and also be free from random shifting along the time.

Take an AES encryption for example, the select function that involves a key byte and a ciphertext byte in the last round to attack is: $V_{i,k} = [Sbox^{-1}(c_i^m \oplus k)/16]$, where m is the ciphertext byte index ($\in [0, 15]$) , i is the execution index. The label function for the DDLA is over both the execution index and the assumed key byte value, $H_{i,k}$. We summarize the new DDLA strategy for MLP training and key byte retrieval as follows:

Note that due to the data structure of AES T-tables, we can actually retrieve corresponding four key bytes which operate on a selected T-table in the last round of AES encryption from one DDLA process.

3 RAN$NET CACHE MONITOR

In this section, we introduce our novel Ran$Net monitor. We first present the threat model in Section 3.1. We then describe our cache monitor in Section 3.2. The monitor is triggered only when the encryption starts, and runs concurrently with the ransomware to generate measurement traces while probing the cache state. We also analyze the properties/capabilities of the cache monitor in Section 3.3 with comparison to prior monitors.

3.1 Threat Model

Our threat model is based on real ransomware incidents. Hackers have infected a victim computer with a ransomware through some distribution channel. The ransomeware employs local cryptographic libraries to encrypt selected target files, with a pre-generated secret key, which is erased from the victim system after the encryption is done and before the extortion alert. The ransomware runs with user-level privilege.

To protect against ransomware, our victim system is equipped with the Ran$Net monitor, which coordinates with the slightly augmented cryptographic libraries to capture useful cache activities. The monitor is only triggered when the symmetric encryption software of the library is activated. The cache monitor also runs in the user mode on the victim's computer. It does not access the ransomware code or data at any time. We assume the ransomware uses AES-CBC mode with T-Table implementation which is the most popular standard symmetric encryption scheme.

3.2 Ran$Net Cache Monitor

Our cache monitor is based on the special X86 instruction - `clflush`. For AES, there are four T tables and each table is stored in 16 memory blocks (with the typical cache line size of 64 bytes). If the monitor chooses the first memory block of Te0, it keeps running a large loop as shown in Listing 1. Each loop iteration runs `clflush Te0[0]` and reads the time stamp (`rdtscp`) and logs it in an array. The execution time each `clflush` takes can be calculated by the differential between the time stamps of the current iteration and the prior iteration.

Listing 1: Ran$Net Cache Monitor

```
1. for i:=1 to NUM_LOOPS do
2. begin
3.     clflush Te0[0];
4.     array[i] = rdtscp;
5. end
```

We next describe how to mount our monitor against certain cryptographic library (augmented). Looking into common cryptographic libraries such us `Openssl` and `Pycryptodome`, AES is composed by two functions: `set_up_AES_key()` and `AES_encrypt()`. The function `set_up_AES_key()` derives sub-keys for each round from a master key (secret), and the function `AES_encrypt()` performs encryption with sub-keys. In Ran$Net, we use function `set_up_AES_key()` as a trigger to activate the cache monitor, as shown in Figure 1. This requires slight instrumentation of the function by hooking. The hook function contains a monitor trigger before the original `set_up_AES_key()` function. In Linux, we can use function wrapper and some system tricks, such as `LD_PRELOAD` or command hijack, to perform the function hooking. Once the cache monitor starts tracking the states of the cacheline, considering the victim file is typically of multiple AES blocks (at size of 16 byte), we do need some recognizable "markers" for each block. We insert a few `clflush` instructions in `AES_encrypt()` function before each data block encryption. Experiments show that such mechanism generate markers on the cache traces to distinguish block-wise encryptions.

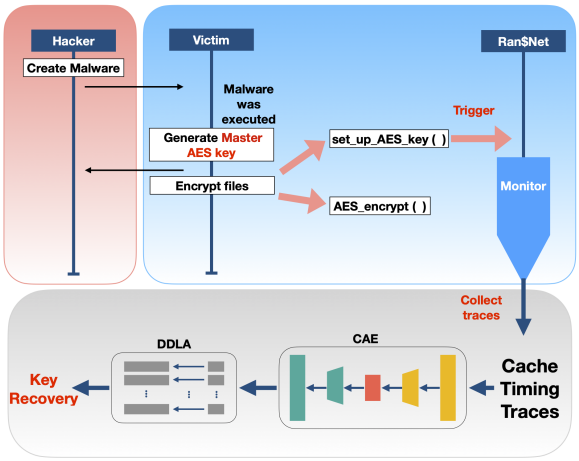


Figure 1: Ran$Net Methodology

Our cache monitor runs concurrently with the ransomware encryption after being triggered, shown in Figure 1. It generate a trace with timing samples, falling into three levels, as demonstrated by a sample trace shown in Figure 2. The timing samples at the middle level, ~90 cycles, correspond to the few `clflush` instructions inserted before each AES block encryption and are used as markers. The timing samples at the highest level, ~300 cycles, between two batches of marker samples indicate how many times the selected cache line has been accessed by the block encryption. The timing samples at the lowest level, ~ 70 cycles, also between two batches of markers, indicate non-access activities by the block encryption on the monitored cache line. As shown in Figure 2, subtraces for each block encryption can be easily partitioned, which consist of detailed access and non-access information during the cipher execution. What is more, the timing side-channel is very strong, the difference between the high timing points and the low timing points is ~230 cycles, much larger than the previously reported 9 – 12 cycles with the conventional cache monitors. The experimental setup is on an Intel 4-core i7-7700 processor (with Kaby Lake architecture) and Ubuntu 20.04.3 LTS.

3.3 Capability Analysis of the Cache Monitor

We next investigate two properties of Ran$Net cache monitor distinguishability of the timing side-channel and the monitor resolution.

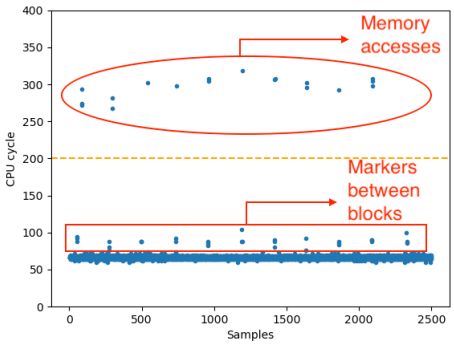


Figure 2: An example timing trace for running RAASNet when one cache line is monitored

Timing Side-channel - The cache monitor observes timing difference and infers memory accesses of the victim. The distance between the two timing distributions determines the distinguishability of the side-channel.

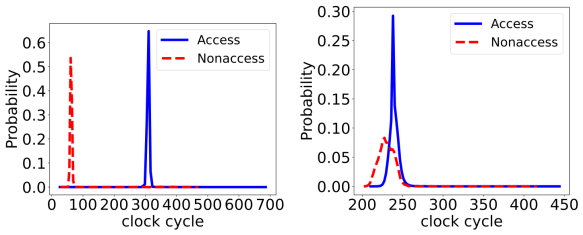
Monitor Resolution - As our cache monitor runs concurrently with the victim application and the meaningful information leakage lies in the high timing samples, the speed of the monitor relative to the victim memory accesses determines the resolution: the probability that two consecutive memory accesses can both be recognized.

3.3.1 Distinguishability of the Timing Side-channel. As shown in Figure 2, we observe two sets of timing samples in addition to the block markers: high indicates *access* to the monitored cache line by the victim and low indicates *nonaccess*. We run a microbenchmark to look into the two timing distributions to examine the timing side-channel.

We run our monitor concurrently to a sequence of memory access instructions (both `CLFLUSH` and memory load are on the same virtual address), and collect timing of each flush instruction. The two timing distributions are depicted in Figure 3a, which appear to be far apart with their means at 70 cycles and 300 cycles, respectively. This shows that on our experimental platform, the distinguishability of the timing side-channel is high and a threshold of 200 cycles can be used to infer *access* or *nonaccess* from the samples with an accuracy up to 96%. By comparison, for the prior Flush+Flush cache monitor, where two flush instructions sandwich a victim execution, and two timing distributions are obtained for the victim access versus no-access of the selected cache line, as shown in Figure 3b. The difference of the means is only 10 cycles, and the distributions are hard to distinguish.

3.3.2 Monitor Resolution. With the high distinguishability of the timing side-channel, we can accurately identify a victim memory access when it is captured by our cache monitor. However, if two or more consecutive victim accesses happen too closely in time, the concurrent cache monitor may not capture all of them as it takes time for each flushing activity of the monitor.

For AES-128, there are a total of 40 lookup operations to the same T table, distributed into 10 rounds. We vary the combination of the plaintext and key, and for each combination we consider the first two lookup operations that hit the selected cache line being monitored. We label these two lookup operations as $\{i_1\}$ and



(a) Our monitor **(b) Prior Flush+Flush**

Figure 3: Timing distributions of our monitor vs. the prior F+F monitor

$\{i_2\}$, where $i_1, i_2 \in [1, 40]$ are the lookup indexes, and the distance between them is defined as $\Delta = i_2 - i_1$.

We compute the probability for different number of accesses distinguished from the timing traces (where $N_{true} = 2$ is the correct number while others are either overcounting or undercounting), when the distance between the access indexes is varying. We summarize the probability matrix in Table 1, where HT stands for victim and monitor were pined to the same physical cores but different logic cores, and NonHT means the two processes are pinned onto different physical cores. When the lookup operation index distance is less than 3, as shown in the first two rows of Table 1, the probability that both the two accesses are distinguished is low, and the probability to observe at least one access is very high. When the distance is great than or equal 3, the probability to distinguish both them is above 80% and the probability to capture at least one access is about 90%.

Table 1: Distribution of $Access_{obs}$ under different distance between two consecutive accesses to the same cache line

Distance	$P(Access_{obs} = N\|HT)$				$P(Access_{obs} = N\|NonHT)$			
Δ	**N = 2**	$N = 1$	$N = 0$	$N > 2$	**N = 2**	$N = 1$	$N = 0$	$N > 2$
1	**53.1%**	40.7%	2.6%	3.6%	**0.9%**	91.9%	7.1%	0%
2	**68.6%**	23.0%	1.7%	6.7%	**4.1%**	88.8%	6.9%	0.3%
3	**81.5%**	6.4%	0.3%	11.9%	**83.4%**	11.3%	2.0%	3.3%
> 3	**85.5%**	7.7%	0.5%	6.3%	**88.6%**	7.5%	2.7%	1.2%

This result indicates that our concurrent cache monitor is able to capture **round – level** access with high accuracy, but is less reliable to differentiate accesses within each round.

4 EXPERIMENTS

In this section, we present experimental results for our proposed Ran$Net. We illustrate how we utilize the CAE to capture shifting signals in the cache timing traces and reduce the computational complexity of the signals using sensitivity analysis. We evaluate the DDLA on CAE features.

4.1 Trace Collection

The RAASNet ransomware experimental setup is Python 3.7 with `Pycryptodome` 3.11. A test file of size 160K bytes (10,000 data blocks) is encrypted by AES-CBC. Figure 4 depicts the subtrace for three blocks' encryption in a long trace collected by the monitor. We partition them into segments by the markers, shown in Figure 5. We use a timing threshold of 200 cycles to binarize the timing samples to 1 and 0, respectively, as shown Figure 6. Thousands of

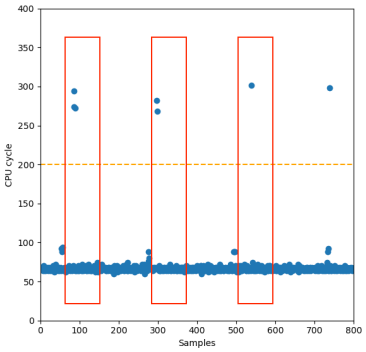


Figure 4: Partial cache trace (3 blocks)

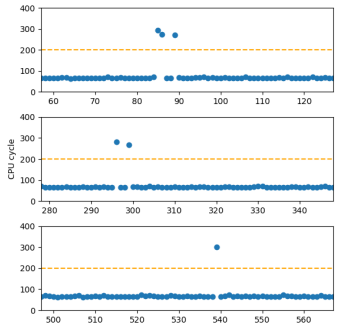


Figure 5: Partitioned to segments

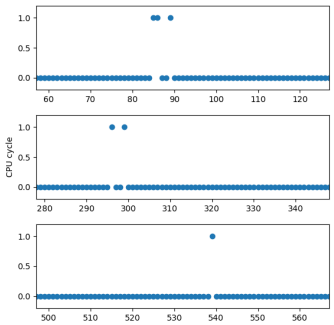


Figure 6: Binarized segments

segments can be generated from one trace and are fed to the follow-on deep learning tool for key retrieval. Note that each segment subtrace contains about 200 time points with small variation.

4.2 Feature Exaction Using CAE

Although our segment subtrace contains about 200 points, many of them do not carry useful information for AES T-table accesses. In addition, the real key-bearing signal (samples) may be a moving target in the subtrace and shifting in time. We propose to employ the convolutional autoencoder (CAE) to filter useless features and address the moving signal issue. Figure 7 shows the structure of a CAE, which consists of a convolutional encoder for feature compression and a deconvolutional decoder for reconstruction. The bottleneck layer outputs features at a reduced size.

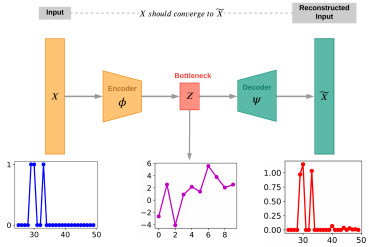


Figure 7: Convolutional autoencoder for feature extraction

To reach a minimal mean-square-error reconstruct loss, the Adam optimization is chosen with a batch size of 64 and the learning rate is set to 5×10^{-4}. ReLu is selected as activation function in both convolutional layers and deconvolutional layers. We use Random-Normal weight initialization. We find out that for the bottleneck layer, the smallest size (10 nodes) leads to the same performance as other larger sizes. Figure 7 shows the first 50 samples of one example input trace, which is recovered with almost no errors by the autoencoder, with the bottleneck layer outputting only 10 features.

4.3 Last-round Access Prediction using CAE features and Sensitivity Analysis

Based on the resolution analysis given in Section 3.3.2, the raw cache timing trace for each block contains round-wise access information on the monitored cache line. To assess if the same information persists in the CAE features, we train 10 MLPs all with the same input - the CAE extracted features, but with different output, one for the access of the monitored cache line in each round. If four related operations in a round access the cache line (no matter how many times), the label of the MLP is 1; otherwise 0. We then use the models to predict the access result of the monitored cache line by the corresponding round execution. Each MLP model is used to predict round-wised access pattern of the monitored cacheline. Figure 8 shows the prediction accuracy of all 10 MLP models, it verifies that MLP trained with CAE features are able to infer the round-wise access and, in particular, predict the first/last round access with > 80% accuracy (50% for random guess).

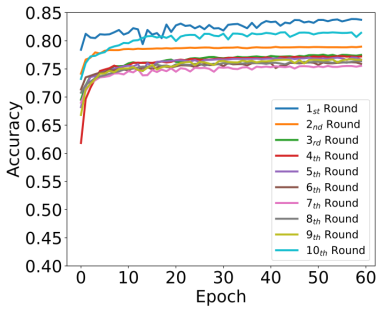


Figure 8: Validation accuracy of MLP round access predictions using CAE features as the input

Next we conduct a sensitivity analysis on the 10-th MLP (which predicts the last-round access) together with the CAE encoder part. This illustrates where are the points of interest on the raw traces, from which our DNN model extracts most leakage information for the target byte operation. Figure 9 shows the changes of access probability (lower part) output by the full neural network (CAE encoder + 10-th MLP) after we flip the value of input trace at each time point (upper part). In the example, monitored cacheline is accessed during the 4-th (29-th time point), 6th(30-th time point), and 10th(33-rd time point) round. It shows that the most sensitive time point is 33-rd point, where the predicted access probability decreases to 31.3% when the sample is flipped from '1' to '0', resulting in misclassification to "nonaccess". The sensitivity analysis shows that our trained CNN model is able to locate the time point on the input trace corresponding to the target last-round byte access,

despite its location varying in difference traces, and extracts the leakage information for inference.

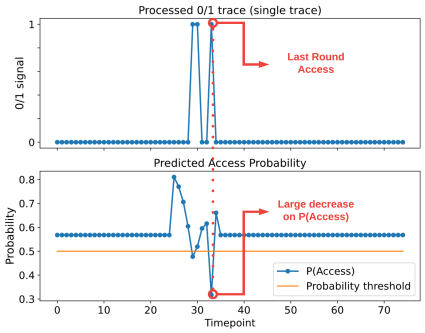


Figure 9: Sensitivity analysis of a processed timing trace

4.4 Improved Non-profiled DDLA Attacks

To recover the AES key used by a ransomware infection, we conduct DDLA on the CAE-features in the attack phase of Ran$Net, i.e., train 256 models each with the same CAE-features but a unique set of pseudo-labels under corresponding key hypothesis. In the attack/evaluation, validated cross-entropy loss is used to rank the key guess. In the following experiments, we first target the 14^{th} byte of last round key. We partition the attack dateset into a subset with 80% traces for training and another subset with 20% traces for validation. Figure 10 shows the values of validated cross-entropy loss for 256 neural networks in the non-profiled attack. Each neural network is trained under a distinct key hypothesis. The result shows that model trained under the correct key hypothesis corresponds to the lowest validation loss. It implies that only the correct model captures the leakage signal, while other wrong models are fitting the random noises. What is more, after selecting the correctly trained neural network, we can use it to recover the key-value of the rest 3 related bytes.

4.5 One Attack Model for Four Subkey Bytes Recovery

For the first key byte (14^{th}), the MLP model corresponding to the correct key (k_c) learns the mapping from the input traces to the memory access pattern of the target lookup operation in the last round precisely. This MLP model can be regarded as a profiled model for the other three subkey bytes related to T-table 0 (2^{nd},6^{th},10^{th} key byte), and can be used directly in the attack phase to recover them. Figure 11 plots success rates of recovering four bytes in the non-profiled attack. The success rate of recovering all the four key bytes by the MLP model reaches 100% as $|S^A| = 1350$, while some bytes can be recovered with a little less number of traces. Compared to the prior Flush+Flush attack [4], our non-profiled DDLA attack requires only 10% measurement traces, and one model is used for four key bytes, another 75% reduction in the attack complexity.

5 CONCLUSION

In this work, we propose Ran$Net, an anti-ransomware methodology based on cache monitoring and deep learning. The Ran$Net

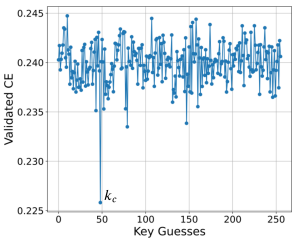


Figure 10: DDLA for Cache Timing Traces (Validated CE for 14^{th} byte)

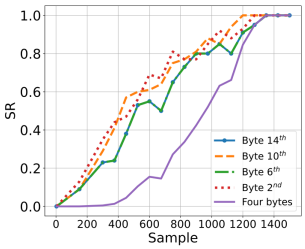


Figure 11: Success rates for 4 key bytes recovery of AES-CBC in Ransomware

monitor is able to capture high-resolution memory access patterns of a victim *during* its execution. With such high-dimensional detailed leakage, deep learning is exploited to learn the leakage and retrieve the secret. We adopt convolutional operations to address the signal shifting issue. The non-profiled attack is much more efficient than prior cache timing attacks. Similar cache monitors can also be built on other processors with counterpart instructions. Our future work will apply this framework to other systems (e.g., with ARM processors or AMD processors), other cryptographic algorithms (ECC etc.), to extend the use of our methodology.

Acknowledgment: This work was supported in part by National Science Foundation under grant # 1916762 and Center for Hardware and Embedded System Security and Trust (CHEST) industry funds.

REFERENCES

[1] Internet Crime Complaint Center. 2021. Internet Crime report 2020. https://www.ic3.gov/Media/PDF/AnnualReport/2020_IC3Report.pdf.
[2] Casey Crane. 2020. Recent ransomware attacks. https://www.thesslstore.com/blog/recent-ransomware-attacks-latest-ransomware-attack-news/.
[3] P. R. Lakshmi Eswari and N. Sarat Chandra Babu. 2012. A practical business security framework to combat malware threat. In *World Congress on Internet Security (WorldCIS-2012)*. 77–80.
[4] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. 279–299.
[5] Kai Huang, Yanfang Ye, and Qinshan Jiang. 2009. ISMCS: An intelligent instruction sequence based malware categorization system. In *2009 3rd International Conference on Anti-counterfeiting, Security, and Identification in Communication*. 509–512. https://doi.org/10.1109/ICASID.2009.5276989
[6] Jonghoon Kwon and Heejo Lee. 2012. BinGraph: Discovering mutant malware using hierarchical semantic signatures. In *2012 7th International Conference on Malicious and Unwanted Software*. 104–111. https://doi.org/10.1109/MALWARE.2012.6461015
[7] Philip O'Kane, Sakir Sezer, and Kieran McLaughlin. 2011. Obfuscation: The Hidden Malware. *IEEE Security Privacy* 9, 5 (2011), 41–47. https://doi.org/10.1109/MSP.2011.98
[8] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology*, David Pointcheval (Ed.). 1–20.
[9] Colin Percival. 2005. Cache Missing for Fun and Profit. In *In Proc. of BSDCan*.
[10] Benjamin Timon. 2019. Non-profiled deep learning-based side-channel attacks with sensitivity analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems* (2019), 107–131.
[11] Xabier Ugarte-Pedrero, Mariano Graziano, and Davide Balzarotti. 2019. A Close Look at a Daily Dataset of Malware Samples. *ACM Trans. Priv. Secur.* 22, 1, Article 6 (jan 2019), 30 pages. https://doi.org/10.1145/3291061
[12] Leon Voerman. 2021. Open-Source Ransomware As A Service for Linux, MacOS and Windows. https://github.com/leonv024/RAASNet.
[13] Wei Yan, Zheng Zhang, and Nirwan Ansari. 2008. Revealing Packed Malware. *IEEE Security Privacy* 6, 5 (2008), 65–69. https://doi.org/10.1109/MSP.2008.126
[14] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symp.* 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom