IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 31, 2020, accepted January 20, 2021, date of publication February 1, 2021, date of current version February 8, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3056190

Defeating Hardware Prefetchers in Flush+Reload Side-Channel Attack

ZIHAO WANG [1,2], SHUANGHE PENG [1], WENBIN JIANG [1], AND XINYUE GUO [1]
[1]School of Computer and Information Technology, Beijing Jiaotong University, Beijing 100044, China
[2]Department of Computer Science, Indiana University at Bloomington, Bloomington, IN 47405, USA

Corresponding author: Shuanghe Peng (shhpeng@bjtu.edu.cn)

This work was supported by the National Training Programs of Innovation and Entrepreneurship for Undergraduates in China.

ABSTRACT Hardware prefetching can seriously interfere with Flush+Reload cache side channel attack. This interference is not taken into consideration in previous Flush+Reload attacks. In this paper, an improved Flush+Reload is provided which minimizes the impact of hardware prefetchers. Specifically, prefetching is analyzed based on reverse engineering and the result is used to make an evaluation model to evaluate the impact of hardware prefetching on Flush+Reload attacks. Then the model is applied to fine tune the placement of probes in Flush+Reload attack to mitigate the prefetching impact. The experiments show that the approach is effective on the Core i5 processor which is equipped with highly aggressive prefetchers.

INDEX TERMS Hardware prefetching, Flush+Reload, reverse engineering.

I. INTRODUCTION

Side channel attacks leak sensitive cryptographic information through physical channels such as power, timing, etc. and are typically specific to the actual implementation of the cryptographic algorithm [6]. An important class of these attacks is based on measurements from cache memory systems. In the attacks, the cache data between cores results in an unintentional side-channel through which sensitive information may be leaked to attackers from other cores. The proposed attacks were based on two major techniques, Flush+Reload [4], [5], [7], [10], [16], [9], [25] and Prime+Probe [17]–[24], according to the granularity of the attack. While Prime+Probe works on a cache set granularity, Flush+Reload works on a single cache line granularity.

One thing that has not been well studied is the impact of the hardware prefetching on side channel attacks. Hardware prefetching [8] is a feature of modern processors and enabled by default. Modern CPUs exploit prefetchers to improve cache hit rate which greatly complicates Flush+Reload attack since it is unable to distinguish a line fetched on demand from one prefetched without subsequently being used.

The direct output attacker can obtain from Flush+Reload is the sequence of probes. The placement of the probes should be located according to the aim of the attack. A good set of probes, i.e., an appropriate workspace of Flush+Reload, is the key to the success of the attack. A well-settled set of probes means the probe sequence can represent different execution traces clearly, which is one of the main purposes of Flush+Reload attack. However, hardware prefetching made the process complicated. Therefore, we explored a strategy to optimize the placement of probes to mitigate the impact of hardware prefetching on Flush+Reload.

Since most technical details of the prefetchers in Intel processors have not been documented yet, we analyze the prefetching behavior based on reverse engineering to make a description of the hardware prefetcher. Since the reverse analysis of CPU prefetching does not target to a specific CPU and does not require specific information about the CPU type, it is adaptive which potentially enables the attack to be adaptable with little effort to other CPUs. We measure the reliability of a Flush+Reload side channel attack by true and false positives as well as true and false negatives. Cache hits that coincide with the event are considered as true positive and cache hits that do not coincide with the event as false positive. Cache misses which coincide with the event are considered as true negative and cache misses which do not coincide with the event as false negative. We evaluate the negative impact of hardware prefetching by the sum of the mathematical expectations of false positives and false negatives, which can be inferred from the reverse engineering results. Based on the evaluation, we use gradient descent to fine tune the placement of probes in Flush+Reload attack to

The associate editor coordinating the review of this manuscript and approving it for publication was Luis Javier Garcia Villalba.

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

mitigate the impact of the prefetcher on the attack, which substantially improves the attack efficiency. We tested our placement strategy on the Intel Core i5 processor which has a very aggressive prefetching feature [8]. Experiments show that the strategy can circumvent the effect of cache prefetching and substantially improve the quality of Flush+Reload attack.

Our main contributions are:

(1) The impact of prefetching on Flush+Reload attack is analyzed and a mathematical model to evaluate the impact of prefetching on the attack is proposed.

(2) A novel strategy aiming to mitigate the effect of cache prefetching on Flush+Reload attack is proposed. The placement of probes in the Flush+Reload attack is adjusted automatically to let the attack more effective.

(3) The proposed approach is evaluated on real-world systems using a web application *Links*. The experiment is carried out on the Core i5 processor which shows that the approach is effective.

II. BACKGROUND

A. CACHE SIDE CHANNEL ATTACKS

It was first mentioned by Hu [1] that cache memory can be considered as a potential vulnerability in the context of covert channels to extract sensitive information. Cache side channel attacks exploit timing differences caused by the lower latency of CPU Caches compared to physical memory. The possibility of exploiting these timing differences was first discovered by Kocher [2] and Kelsey *et al.* [3]. Practical attacks have focused on side channels on cryptographic algorithms and covert channels. Access-driven cache attacks can be roughly categorized into either ''Prime+Probe'' or ''Flush+Reload'' case. Our work is constructed upon Flush+Reload case.

Flush+Reload attack exploits the availability of shared memory and especially shared libraries between the attacker and the victim process. Gruss *et al.* [9] have shown that a variant of Flush+Reload without the *clflush* instruction is possible without a significant loss in accuracy. Applications of Flush+Reload have been shown to be reliable and powerful, mainly to attack cryptographic algorithms [12]–[15]. Flush+Reload is also being used to compromise user privacy. For example, Cai and Chen [26] have shown that keystrokes can be reliably recovered from accelerometer data on smartphones. Chen *et al.* [27] showed that observing a web application's behavior reveals information about the user's input.

Flush+Reload technique proceeds in three phases, as shown in Figure 1. At *Flush* phase, the attacker flushes the selected memory addresses from the entire cache hierarchy using the *clflush* instruction. Owing to the inclusiveness property of the LLC in Intel processors, the *clflush* instruction will evict those cache lines from all cache levels. Then, in *Idle* phase, the attacker waits for the victim's operations. Finally, at *Reload* phase, the attacker reloads previously flushed memory addresses and measures the access time of each of them. For each memory address, if it is used by the victim during *Idle* phase, it will be reloaded to the cache, i.e., a cache hit occurs, which results in a lower reload time. If not, then this memory address resides in the memory, resulting in higher reload time due to a cache miss. In this way, the attacker can figure out which memory addresses are used by the victim, i.e., which Flush+Reload probes are triggered. By putting those three phases in loop, the attacker can obtain a probe sequence which is related to the execution trace of the victim and extract sensitive information from it.

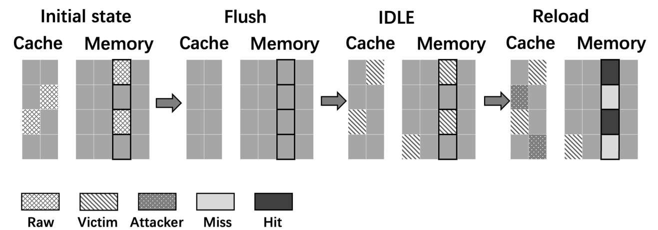

FIGURE 1. Phases of Flush+Reload Technique.

B. HARDWARE PREFETCHING

The high cost of memory accesses is one of the fundamental bottlenecks that limits processor performance. Cache misses result in accesses to lower-level caches or main memory, which is time-consuming. Data prefetching is a technique that predicts the usage and fetches data from the main memory or the lower-level cache to the higher-level cache prior to the actual access. Here we consider only hardware-based prefetching techniques.

Hardware prefetcher is introduced for the sake of performance. However, it also brings an impact on security on both good and bad side. On the good side, it complicates the traditional cache side channel attacks, since it makes them unable to distinguish a cache line fetched on demand from one prefetched without subsequently being used. On the bad side, it can be used to build new covert channels to extract sensitive information. Shin *et al.* [30] point out that CPU cache stride prefetching can be used as a reliable channel and be exploited against ECDH algorithm in OpenSSL. In this paper, we focus on its impact on traditional cache side channel attacks, specifically on Flush+Reload.

In Sandy Bridge and successive processors, each core is equipped with four types of hardware prefetchers: *Streaming prefetcher, Spatial prefetcher, Data Cache Unit (DCU) prefetcher, and Instruction pointer (IP)-based stride prefetcher* [28]. Unfortunately, details about the behavior of those prefetchers are not publicly known, except for a brief explanation of each prefetching mechanism in Intel documents [28], [29]. Those prefetchers can be quite aggressive. For example, according to the documents, the streaming prefetcher could prefetch up to 20 cache lines ahead.

III. DESIGN AND IMPLEMENTATION

A revised Flush+Reload attack that overcomes the negative impacts caused by hardware prefetching is proposed here. Figure 2 demonstrates a high-level workflow of our

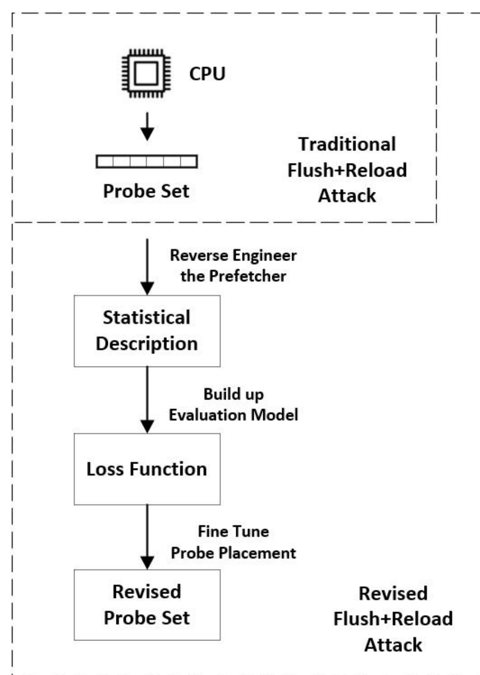

FIGURE 2. Workflow of Prefetcher Aware Flush+Reload Attack.

attack. We first conduct reverse engineering towards the prefetcher, and then use the obtained statistical description of the prefetcher to build up an evaluation model of the prefetching impact on Flush+Reload attack, which is regarded as the loss in the attack scenario. Using the obtained loss function, we then fine tune the placement of probes to minimize the loss value. We then take the new probe set into a Flush+Reload attack that circumvents the interference of the prefetcher. The whole process is done automatically, which potentially enables the attack to be migrated to other CPUs with little effort.

To simplify explanations, and without loss of generality, we assume that the attacker targets a single memory page. For the cases that target multiple memory pages, the attacker can do the same for each page. In the case of cache miss, CPUs always fetch a whole cache line. Therefore, we cannot distinguish between offsets of different accesses within a cache line. In addition, we can deduce the same information by probing only one memory address within each cache-line sized memory area. We design and implement our attack on Intel Core 2 Duo-P8700, Core i3-4000M, Core i5-6200U and Core i7-3520M and use Core i5-6200U for demonstration. All those processors have a cache line size of 64 bytes, which means it would require 64 cache lines to cover a 4KB memory page. Thus, on a memory page, there are only 64 positions that a probe can choose from. All results are from experiments on an Intel Core i5-6200U CPU.

A. REVERSE ENGINEERING THE PREFETCHER

Since most technical details of the prefetchers in Intel processors have not been disclosed yet, we tried to make a model that can describe the behavior of hardware prefetching, which enables the attacker to figure out whether their attack model is reasonable, i.e., whether their attack model will be heavily affected by hardware prefetching, even if they don't know how much and which prefetchers there are.

We analyzed the CPU memory prefetching based on reverse engineering and use the results later in the next section to build up the evaluation model of the prefetching impact. The behavior of prefetcher is complex and vary based on the access pattern and the availability of memory bandwidth [28]. To provide a basic characterization, we conduct prefetcher reverse-engineering with respect to accesses to a single cache line $\mathcal{A}$, by observing the cache state of the other cache line $\mathcal{B}$ after accessing cache line $\mathcal{A}$. For each cache line $\mathcal{A}$, we consider only the cache line $\mathcal{B}$ whose index is bigger than $\mathcal{A}$. Accesses to cache line $\mathcal{A}$ are (1-w%) sequential and w% non-sequential. In the experiments below, w is a fixed value 20. For instance, when the index of $\mathcal{A}$ is 50, we only consider the case that the index of $\mathcal{B}$ is between 51 and 63. Then for each $\mathcal{B}$ in the range, access $\mathcal{A}$, and observe the cache state of $\mathcal{B}$. Prefetcher reverse engineering aims to figure out the probability that cache line $\mathcal{B}$ is used by the CPU after accessing cache line $\mathcal{A}$.

The reverse-engineering result on Core i5-6200U processor is shown in Figure 3. The index of a line indicates its location in a memory page, i.e., index $\in$ [0,63]. Darker dot means higher probability that the cache line $\mathcal{B}$ is cached. Each cache line is tested 100 times. The index of line $\mathcal{A}$ and $\mathcal{B}$ is denoted as I_A and I_B. The probability that cache line $\mathcal{B}$ occupies the CPU cache after accessing cache line $\mathcal{A}$ is denoted as $P(I_A, I_B)$, which is a statistical description of the prefetcher.

In the Core i5-6200U processor, there are four types of prefetchers that we can deal with:

(1) *L2 hardware prefetcher*: Fetches additional lines of code or data into the L2 cache.

(2) *L2 adjacent cache line prefetcher*: Fetches the cache line that comprises a cache line pair (128 bytes).

(3) *DCU prefetcher*: Fetches the next cache line into L1-D cache.

(4) *DCU IP prefetcher*: Uses sequential load history (based on Instruction Pointer of previous loads) to determine whether to prefetch additional lines.

In Figure 3(a), all the four prefetchers are enabled. In Figure 3(f), all the four prefetchers are disabled. Figure 3(b) (c) (d) and (e) show the results of only enabling (1) to (4) respectively. Specifically, Figure 3(b) shows the result of enabling *L2 hardware prefetcher* only. Figure 3(c) shows the result of enabling *L2 adjacent cache line prefetcher* only, and so on.

In general, those four prefetchers are enabled as default and we do not have authority to enable or disable single one of them so only the result of Figure 3(a) is got. To get those individual settings results, root privilege is needed. In order to see the comparison, root account is login to get these individual results as blank control group.

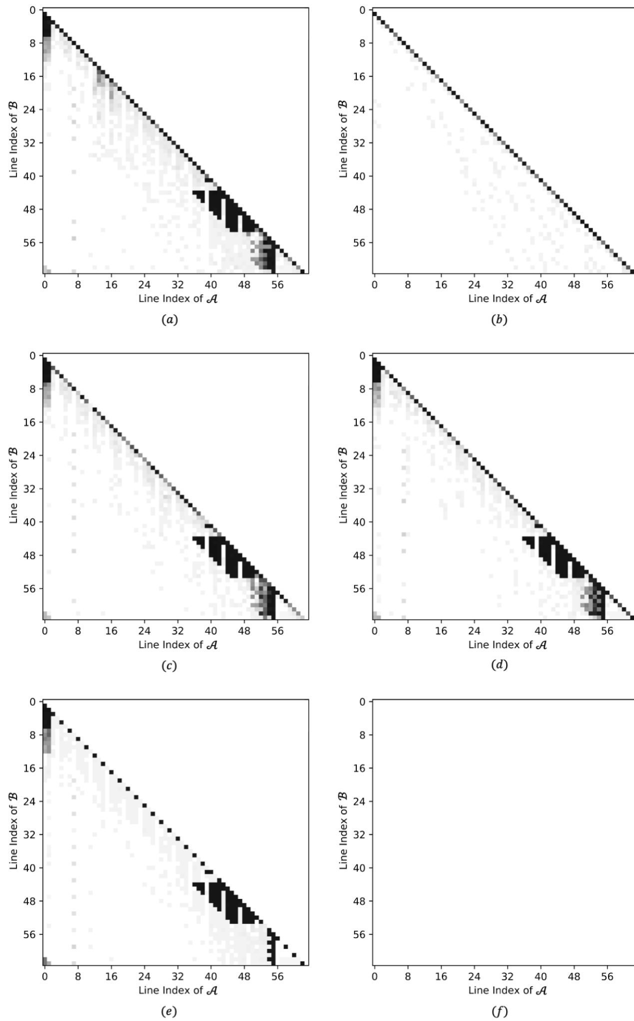

FIGURE 3. Probability Distribution $P(I_A, I_B)$ tested on Core i5-6200U.

The probability distribution function P got from Figure 3(a) is the final output of the reverse-engineering process.

B. EVALUATION MODEL

In this section, we are going to use the statistical description of the prefetcher obtained in section III-A to quantify the impact of hardware prefetching on Flush+Reload attack. Probes in Flush+Reload attack are placed on memory address which are accessed depending on secret information or specific events. We measure the reliability of a Flush+Reload attack by true and false positives as well as true and false negatives, which are defined as follows:

- True positive (*true* +): Cache hit that coincides with the event.
- False positive (*false* +): Cache hit that does not coincide with the event.
- True negative (*true* −): Cache miss which coincides with the event.
- False negative (*false* −): Cache miss which does not coincide with the event.

We evaluate the hardware prefetching impact by the sum of the mathematical expectations of false positives and false negatives. Thus, the loss function is defined as formula (1):

$$Loss = E\,(false\ +) + E\,(false\ -) \tag{1}$$

For a Flush+Reload attack which has N probes within one memory page, denote their corresponding cache line index as $[\theta_1, \theta_2, \ldots, \theta_N]$ (from low to high in address order). Then the mathematical expectation of false positives can be calculated by formula (2):

$$E(false\ +) = \sum_{1 \le i < j \le N} P\left(\theta_i, \theta_j\right) \tag{2}$$

Since hardware prefetching does not generate additional false negatives, the mathematical expectation of false positives is the only indicator to evaluate the impact of hardware prefetching on Flush+Reload. It can be used to determine the rationality of the probe placement, and be used as a reference on probe selection.

C. DEFEATING HARDWARE PREFETCHING

With the knowledge of hardware prefetching, we can make a more refined selection of Flush+Reload probes. By using static analysis, we can know the range of the code that highly corresponding to a specific event. However, there must be an extra range of code that also corresponds to the event to some degree because of the hardware prefetching. Therefore, we can take advantage of hardware prefetching in turn to widen the placement range of a probe. As shown in Figure 4, the box filled with grey is defined as the range of the code that is highly related to a specific event, which is usually shown as a function, a functionally independent piece of code, or a critical branch. The blue frame defines the scope of a probe placed. It is generally believed that a probe can only be placed in the box filled with grey [25]. The code outside the box filled with grey is always not considered, because it will bring extra false negatives. In many cases, the probe is placed directly on the first instruction of the critical code segment. However, based on the knowledge of hardware prefetching, we can make a more refined choice on a larger code scope, as shown in Figure 4. Our basic idea is to avoid a large number of false positives by introducing a small number of false negatives.

Widening the scope of a probe placed will bring extra false negatives, which should be counted as the fault of hardware prefetching. The initial position of all the Flush+Reload probes is set as the last instruction of their corresponding grey-color box, denoted as $[\theta_1^0, \theta_2^0, \ldots, \theta_N^0]$ (from low to high in address order). Their new position at time t is denoted as $[\theta_1^t, \theta_2^t, \ldots, \theta_N^t]$. Then the mathematical expectation of false negatives can be calculated by formula (3):

$$E(false\ -) = \sum_{i=1}^{N} P\left(\theta_i^0, \theta_i^t\right) \tag{3}$$

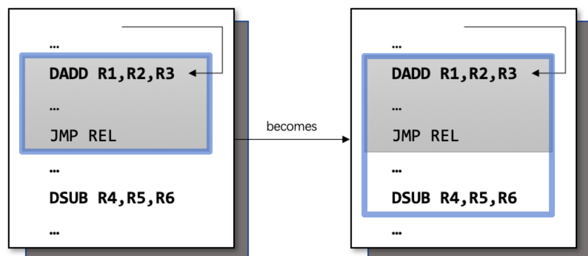

FIGURE 4. Placement Scope of Flush+Reload Probes.

Thus, the *Loss* at time *t* can be calculated by formula (4):

$$Loss = \sum_{1 \leq i < j \leq N} P\left(\theta_i^t, \theta_j^t\right) + \sum_{i=1}^{N} P\left(\theta_i^0, \theta_i^t\right) \tag{4}$$

By using the loss function, we can apply gradient descent algorithm to fine tune the probe placement, and thereby minimize the loss, i.e., the impact of prefetching.

Since the probe placement that minimizes the loss is determined uniquely by the loss function, the process of fine tuning the probe placement is done automatically according to the result of reverse-engineering.

IV. EVALUATION

Here, we take side-channel leaks in web applications as an example to test our strategy of defeating hardware prefetching.

A. SIDE-CHANNEL LEAKS IN WEB APPLICATIONS

Unlike a desktop application which is usually a single executable program, a web application is usually composed of two parts: browser-side and server-side. Here we build an attack against the command-line web browser *Links*. From the result of the Flush+Reload attack, we can know which web page the victim is visiting. The specific attack scenario is described as follows.

Suppose that the attacker knows the possible web pages (a candidate set) that the victim would visit. Then the attacker tries to use Flush+Reload against *Links* to figure out which web page in the set the victim was visiting.

As shown in Figure 5, the process of the attack is composed of three stages.

At the training stage. The attacker simply runs the Flush+Reload attack against its own *Links* browser (same as the victim) when they run the program on every web page in the set for multiple times. The number of times that each web page is sampled is denoted as T. If there are N different web pages, there will be $N * T$ training samples. Each sample is a sequence of probes in Flush+Reload attack which is represented by a string of single-character probe names, such as 'A', 'B', 'D' as shown in Figure 5, and is labeled with its corresponding web page. The probe sequence somehow reflects the browser's execution trace when some web page is visited. All probe sequences with their labels form a complete training set.

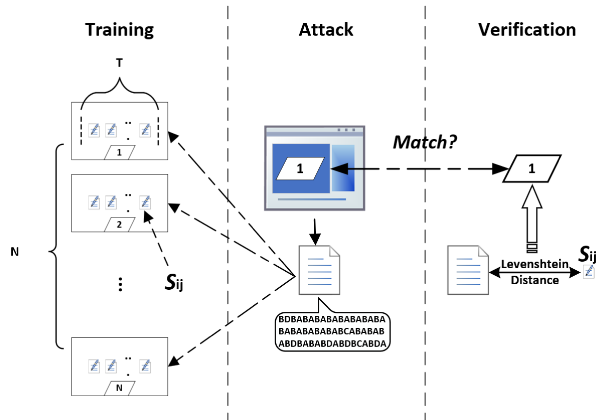

FIGURE 5. Stages in the Process of Flush+Reload Attack.

Next, the actual attack happens. The attacker runs Flush+Reload attack against the victim. When the victim visits one of the web pages using *Links*, the attacker records the probe sequence.

Finally, at the verification stage, the attacker finds the probe sequence in the training set that is closest to the one obtained from the attack stage. Levenshtein distance [11] is used to measure the closeness, which is defined as the smallest number of basic edit unit needed to bring one string to the other. The attacker computes the Levenshtein distance between the obtained probe sequence and sample S_{ij}, where $i = 1, 2, \ldots, N, j = 1, 2, \ldots, T$. The label of the one with the smallest value is assumed to be the web page the victim visited using *Links*.

In the test, T samples of each web page are taken at the training stage. Then, for every web page in the set of size$N$, the vulnerable process is run on the web page K times. The probe sequences from each of the $K * N$ runs are put independently at the verification stage, and we check how many of them is correct. The number of correct verifications is denoted as S. Then the quality of the Flush+Reload attack can be evaluated by formula (5):

$$Success\ Rate = \frac{S}{K * N} \tag{5}$$

B. COMPARISON TO TRADITIONAL FLUSH+RELOAD

We tested our strategy of defeating hardware prefetching on the Intel Core i5-6200U with 4GB memory and 3MB, 12-way, 64-byte lines LLC. All tests are based on OS Fedora 27.

We tested on a page set that contains 100 Wikipedia pages. Each Wikipedia page is sampled 100 times, 70 of which form the training set and 30 of which form the test set. In the experiment, we use 4 probes on the *Links* browser. The positions of the probes are chosen by using traditional and revised strategies respectively.

As discussed in section III-A, In the Core i5-6200U processor, there are four types of prefetchers that we can deal with:

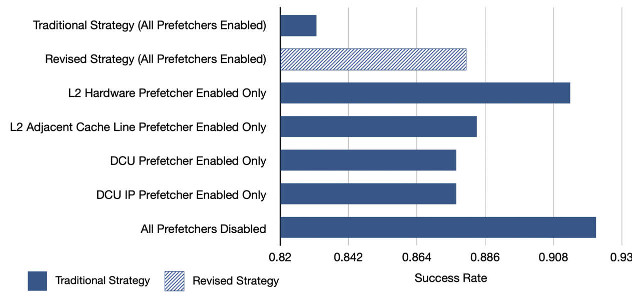

FIGURE 6. Success Rate Comparation between Traditional and Revised Flush+Reload Attack.

L2 hardware prefetcher, *L2 adjacent cache line prefetcher*, *DCU prefetcher* and *DCU IP prefetcher*. We tested both the traditional and the revised Flush+Reload strategy with all the four types of prefetchers enabled. In addition, we tested the traditional Flush+Reload strategy in the condition of enabling the four types of prefetchers separately and disabling all of them.

Figure 6 shows the success rate of the traditional Flush+Reload attack and our prefetcher-aware Flush+Reload attack. The result shows that the revised Flush+Reload attack strategy has a higher quality. In addition, as experiment shows, *L2 hardware prefetcher* has the least impact on the attack.

V. DISCUSSION

A. TIMELINESS

The reverse-engineering results can only represent the characteristics of prefetching behavior at that moment, since it corresponds to the hardware status of that moment. Therefore, it could be weird if we use the probes that decided by the reverse-engineering results during the whole attack process, since Flush+Reload usually works for a long time. However, the behavior patterns of prefetching only fluctuate within a certain range, even if the reverse-engineering results are no longer time effective, adjusting the placement of probes based on the reverse-engineering results can also play a positive role.

To solve the problem of timeliness, we can design a multi-threaded spy program to regularly reverse engineer the prefetcher and update the probe placement, which may perform better.

B. LIMITATION

Our defeating strategy usually perform well on relatively large application. However, it doesn't do well in small programs, because there is only a small scope to adjust the position of the probes.

In addition, our strategy has advantages only when the attacker needs to place multiple probes within a memory page.

VI. CONCLUSION

A framework and methodology to figure out the impact of hardware prefetching on Flush+Reload side channel attack is proposed in this paper. An evaluation model and a novel probe placement strategy that can defeat hardware prefetching are introduced. The proposed strategy is tested on Core i5 processor. Hardware prefetchers on modern machines complicate Flush+Reload attack as the prefetched cache lines are wrongly reported as being accessed by the victim. The proposed strategy decreases the number of false positives but increases the number of false negatives. Yet our strategy has a higher success rate in a web application attack scenario, which is a simple but practical attack scenario. Overall, this paper takes a step to successfully defeat hardware prefetching.

REFERENCES

[1] W.-M. Hu, "Lattice scheduling and covert channels," in *Proc. IEEE Symp. Secur. Privacy*. Washington, DC, USA: IEEE Computer Society, May 1992, pp. 52–61.
[2] P. C. Kocher, "Timing attacks on implementations of Diffe-Hellman, RSA, DSS, and other systems," in *Proc. 16th Annu. Int. Cryptol. Conf. (Crypto)*, 1996, pp. 104–113.
[3] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," *J. Comput. Secur.*, vol. 8, nos. 2–3, pp. 141–158, 2000.
[4] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *Proc. IEEE Symp. Secur. Privacy*, May 2011, pp. 490–505.
[5] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. USENIX Conf. Secur. Symp.*, 2014, pp. 719–732.
[6] Y. Zhou and D. Feng, "Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing," *IACR Cryptol. ePrint Arch.*, vol. 2005, p. 388, 2005.
[7] B. Gülmezoglu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *Proc. Int. Workshop Constructive Side-Channel Anal. Secure Des. (COSADE)*, 2015, pp. 111–126.
[8] *Intel R 64 and IA-32 Architectures Optimization Reference Manual. No. 248966-033*, Intel Corp., Santa Clara, CA, USA, 2016.
[9] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *Proc. 24th USENIX Secur. Symp.*, 2015, pp. 897–912.
[10] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 549–564.
[11] V. Levenshtein, "Binary codes capable of correcting deletions, insertions and reversals," *Sov. Phys. Doklady*, vol. 10, p. 707, Feb. 1966.
[12] B. Gülmezoglu, M. S. Inci, G. Irazoqui, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*. 2015.
[13] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Know thy neighbor: Crypto library detection in cloud," *Proc. Privacy Enhancing Technol.*, vol. 2015, no. 1, pp. 25–40, Apr. 2015.
[14] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Lucky 13 strikes back," in *Proc. 10th ACM Symp. Inf., Comput. Commun. Secur.*, Apr. 2015, pp. 85–96.
[15] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in PaaS clouds," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Nov. 2014, pp. 990–1003.
[16] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A fast and stealthy cache attack," in *Proc. 13th Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*, 2016, pp. 279–299.
[17] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Proc. Cryptographers Track RSA Conf. (CT-RSA)*, 2006, pp. 1–20.
[18] C. Percival, "Cache missing for fun and profit," in *Proc. BSDCan*, 2005, pp. 1–13.
[19] .E Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *J. Cryptol.*, vol. 23, no. 1, pp. 37–71, Jan. 2010.
[20] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proc. 11th USENIX Workshop Offensive Technol.*, 2017, pp. 1–12.

[21] B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache attacks enable bulk key recovery on the cloud," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.*, 2016, pp. 368–388.
[22] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing–and its application to AES," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 591–604.
[23] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 605–622.
[24] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *Proc. Int. Conf. Cryptograph. Hardw. Embedded Syst.*, 2017, pp. 69–90.
[25] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with flush+reload," in *Proc. BlackHat USA*, 2016, pp. 1–11.
[26] L. Cai and H. Chen, "On the practicality of motion based keystroke inference attack," in *Proc. 5th Int. Conf. Trust Trustworthy Comput.* Berlin, Germany: Springer-Verlag, 2012, pp. 273–290.
[27] S. Chen, R. Wang, X. Wang, and K. Zhang, "Side-channel leaks in Web applications: A reality today, a challenge tomorrow," in *Proc. IEEE Symp. Secur. Privacy*, Jun. 2010, pp. 191–206.
[28] *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel, Santa Clara, CA, USA, Apr. 2018.
[29] *Intel 64 and IA-32 Architectures Software Developer Manuals*, Intel, Santa Clara, CA, USA, Mar. 2018.
[30] 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.*, Oct. 2018, pp. 131–145.

ZIHAO WANG received the bachelor's degree in computer science from Beijing Jiaotong University, in 2020. He is currently pursuing the Ph.D. degree with the School of Information and Computing, Indiana University at Bloomington.

SHUANGHE PENG is currently an Associate Professor with the School of Computer and Information Technology, Beijing Jiaotong University. Her research interest includes system security.

WENBIN JIANG received the bachelor's degree in information security from Beijing Jiaotong University, in 2020, where she is currently pursuing the Ph.D. degree with the School of Computer and Information Technology.

XINYUE GUO received the bachelor's degree in security technology from Beijing Jiaotong University, in 2020, where she is currently pursuing the master's degree with the School of Computer and Information Technology.