IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 26 February 2023, accepted 12 March 2023, date of publication 22 March 2023, date of current version 22 May 2023.

Digital Object Identifier 10.1109/ACCESS.2023.3260179

RESEARCH ARTICLE

Vizard: Passing Over Profiling-Based Detection by Manipulating Performance Counters

MINKYU SONG, (Member, IEEE), TAEWEON SUH, (Member, IEEE), AND GUNJAE KOO, (Member, IEEE)

Department of Computer Science and Engineering, Korea University, Seoul 02841, Republic of Korea

Corresponding author: Gunjae Koo (gunjaekoo@korea.ac.kr)

This work was supported in part by the Institute of Information and Communications Technology Planning and Evaluation (IITP) funded by the Korean Government (MSIT) (Research on CPU Vulnerability Detection and Validation) under Grant 2019-0-00533, in part by the ICT Creative Consilience Program under Grant IITP-2022-2020-0-01819, and in part by the National Research Foundation of Korea (NRF) funded by the Korean Government (MSIT) under Grant NRF-2022R1A2C1011469 and Grant NRF-2021R1C1C1012172.

ABSTRACT Cache side-channel attacks have been serious security threats to server computer systems, thus researchers have proposed software-based defense approaches that can detect the security attacks. Profiling-based detectors are lightweight detection solutions that rely on hardware performance counters to identify unique cache performance behaviors by cache side-channel attacks. The detectors typically need to set appropriate criteria to differentiate between attack processes and normal applications. In this paper, we explore the limitations of profiling-based detectors that rely on hardware performance counters. We present an attack scenario, called Vizard, that can bypass the existing profiling-based detectors by manipulating cache performance behaviors of an attack process. Our analysis discloses that cache side-channel attacks include idle periods that can be exploited as attack windows for creating cache events. Vizard generates counterbalancing cache events within the attack windows to hide particular cache performance behaviors of cache side-channel attacks. Our evaluation exhibits that Vizard can effectively bypass profiling-based detectors while maintaining high attack success rates. Our research work represents that attackers can bypass the existing detection approaches by manipulating performance counters.

INDEX TERMS Security attacks, cache side-channel attacks, security attack detectors, hardware performance counters.

I. INTRODUCTION

Cache side-channel attacks have been significant security threats to server computer systems since attackers can leak secret or private data by exploiting the timing differences observed in the cache hierarchy. Note that cache hierarchy is an essential architectural component automatically managed by hardware, and many processes can share the resources in the cache hierarchy. Furthermore, most caches exhibit *deterministic* access behaviors to achieve efficient data reuse rates in limited cache spaces. As such, attackers monitor cache block allocations/evictions provoked by victims' cache accesses to estimate secret data referenced by victim processes. Researchers have disclosed that attackers can effectively decode victims' confidential data using various cache side-channel attack approaches [1], [2], [3], [4], [5], [6], [7], [8], [9]. The attacker usually elaborates the cache side-channel attacks by preparing shared cache space before a target victim process references secret data. Then the victim process can modify the prepared cache blocks while the victim accesses the secret data. The attacker finally decodes the changed cache blocks by referencing the prepared cache space. The attacker can identify the modified cache states since the changed cache blocks exhibit distinct access timing.

Researchers have proposed architectural defense approaches to defend against cache side-channel attacks [10], [11], [12], [13]. Such approaches focus on mitigating the architectural vulnerabilities exploited by the attacks. Moreover, the recent generations of commodity processors include hardware patches for defending against the

The associate editor coordinating the review of this manuscript and approving it for publication was Barbara Masucci.

cache side-channel attacks [14], [15]. However, many server systems still equip old generations of processors, which are vulnerable to the cache side-channel attacks. Hence software-based mitigations are the only possible approaches for such systems even though the software patches lead to significant performance drops [16], [17], [18], [19]. Another possible solution is to detect the attack processes that perform cache side-channel attacks. As mentioned, most cache side-channel attacks include the preparation and scanning of cache blocks; thus, the attacks exhibit particular performance behaviors. Researchers have presented profiling-based detectors that can detect the unique performance behaviors of the cache side-channel attacks [20], [21], [22], [23], [24], [25], [26], [27]. The profiling-based detectors monitor system-wide or per-process performance counters to identify the particular performance behaviors of the target attacks. Since the cache side-channel attacks exhibit extremely distinct behaviors compared to normal applications, the profiling-based detectors can effectively catch attack processes using the cache-related performance counters. Once an attack process is detected, an administrator can terminate the detected attack process to protect confidential data.

Although the profiling-based detectors are effective lightweight solutions for protecting systems from cache side-channel attacks, the detectors may incorrectly catch an innocent application as an attack process if the normal application exhibits unusual cache performance behaviors (e.g. extremely high cache miss rates). Inversely, the detectors may struggle to recognize an attack process if the attack exhibits the commonly observed cache performance levels. Note that the profiling-based detectors rely on the performance counters that monitor the cache behaviors of running processes. Therefore, attackers may evade the detectors by generating counterbalancing cache events that can perturb cache performance counters.

In this paper, we explore an attack scenario, called Vizard, which can hide the unique cache performance behaviors of cache side-channel attacks to bypass the profiling-based detection approaches that rely on cache performance counters. We observe that a typical cache side-channel attack consists of two steps - preparation and scanning. An attack process usually postpones the scanning phase until the target victim changes the prepared cache space. The Vizard uses the time gap between the preparation and the scanning phases for creating the cache events that can counterbalance the cache performance counters by the attack process. By analyzing the cache side-channel attack on a cryptography application, we can figure out an appropriate window size for compensating cache performance levels while maintaining an attack success rate. Our evaluation reveals that the Vizard can effectively hide the unique cache behaviors of the cache side-channel attack to bypass the profiling-based detectors.

The followings are the contributions of this paper.

- We explore the limitations of the existing profiling-based detection approaches that heavily rely on cache performance counters.
- Based on the analysis of cache side-channel attacks, we disclose the attacks include enough idle cycles for creating cache events.
- We present how to set the window size for counterbalancing cache performance while maintaining attack performance.
- We present an efficient way to generate compensating cache events within a limited attack window.
- We evaluate the Vizard attack scenario to exhibit that our proposed evasive attacks can bypass profiling-based detectors.

The rest of this paper is organized as follows. We provide background about cache organizations and cache side-channel attacks in Section II. We discuss the limitations of profiling-based detection approaches in Section III. In Section IV, we propose the Vizard attack scenario. In Section V, we explore how the Vizard attack scenario can be designed with an exemplar cache side-channel attack. We evaluate the effectiveness of the proposed Vizard attack in Section VI. We conclude in Section VIII.

II. BACKGROUND

A. CACHE MANAGEMENT

Cache hierarchy is one of the major attack surfaces in computer processors since attackers can exploit the timing discrepancies in accessing different cache hierarchy levels to leak secret data. Namely, attackers can estimate whether the target data blocks exist at a certain cache level by monitoring the data access time. Processors usually manage cache blocks in a deterministic way, thus attackers elaborately prepare security attacks by exploiting the deterministic cache behaviors. In this subsection, we introduce the deterministic cache organization and management policies that can be exploited by security attacks.

1) CACHE ORGANIZATION

Most modern processors employ cache hierarchy. A memory transaction from a processor core sequentially references multiple cache levels to check whether the requested data is stored in a target cache level. Usually, a cache closer to a processor core is smaller and faster than lower-level caches. When a data block is initially allocated in the cache hierarchy, the *inclusion policy* of the cache hierarchy manages where to place it.

Each cache in the cache hierarchy is configured as a set-associative cache; there are S sets in a cache and W ways in each set. When a memory transaction accesses a target cache, a set index of the cache is computed from the address of a demanded data block. Normally, a processor uses a physical address for calculating cache indexes. However, a virtual address can be used to compute the index of the L1 cache in order to reduce the access latency to the L1 cache by excluding accesses to a translation lookaside buffer (TLB). Once a set is computed, a cache checks whether it holds the demanded data block by comparing the tags of cache blocks

within the selected set. If the demanded data is not found in the cache (i.e. a cache miss), the cache forwards the memory transaction to the lower-level cache, then the demanded data block can be provided from the lower-level caches or main memory.

2) INCLUSIVE AND EXCLUSIVE CACHES

The *inclusion policy* of the cache hierarchy decides target cache levels that will include a data block when the block is allocated to caches or evicted from one of the cache levels. Different inclusion policies can be configured by processor generations or target systems (i.e. personal or server systems).

The inclusive cache policy is generally adopted by most processors. This policy guarantees that all lower-level caches contain copies of the data block if a current cache level includes the data block. In other words, the data blocks in a higher-level cache are a subset of the data blocks of lower-level caches. Even if a cache block is evicted from a higher-level cache due to cache replacement, lower-level caches still include copies of the data block. Thus, if a previously evicted block is re-referenced, the data block can be provided from lower-level caches with a high probability. On the other hand, if a data block is evicted from a lower-level cache, the same data blocks in all higher-level caches need to be evicted from the caches to meet the inclusive cache policy. Since the same data blocks are included in multiple cache levels, inclusive caches exhibit relatively lower access latency if data blocks are re-referenced frequently. However, cache spaces are wasted by multiple copies of data blocks. Intel's personal-use CPUs usually employ inclusive caches.

On the other hand, a data block is placed in only one cache level with the exclusive cache policy. When a data block is first referenced, the data block is only placed in the highest-level cache (usually L1 cache). The cache block is allocated to the lower-level cache if the block is evicted from a current cache level. If the data is re-referenced, the data block can be provided from one of the lower-level caches if found, then the data block is allocated to the highest-level cache only. Exclusive caches utilize cache spaces more efficiently compared to inclusive caches. However, exclusive caches provoke frequent data movements between cache levels when allocating or evicting cache blocks. Recent ARM processors typically adopt exclusive caches.

3) CACHE REPLACEMENT

The cache replacement policy determines which block is evicted from a cache set if the target set is already full and a new block needs to be allocated. With LRU policy, a cache chooses the least recently accessed block in a target set when cache block replacement is necessary. However, LRU policy requires complex hardware design. Thus, most processors employ a pseudo-LRU policy, which mimics the LRU policy using a simpler prioritizing mechanism. A recent article discloses that Intel's modern processors employ a modified pseudo-LRU policy called quad-age LRU [5]. Even though the LRU policy is not the optimal replacement policy, it is known that LRU-like policies exhibit higher cache hit rates compared to other non-deterministic policies, such as a random policy. However, the deterministic mechanisms of LRU-like policies can be exploited by cache side-channel attacks [5].

B. CACHE SIDE-CHANNEL ATTACKS

Cache side-channel attacks exploit differences in access latency observed in the cache hierarchy to leak secret information [1], [2], [3], [4], [5]. The attacks manipulate cache blocks and then monitor cache states modified by victim processes. The modified cache states can be identified by exploiting different access latency between cache hits and misses. Attackers that exploit cache side-channel attacks can effectively decode secret data if the secret information can be decoded by checking executions of specific instructions (e.g. RSA cryptography) [1] or static data accesses (e.g. AES cryptography) [6]. A typical cache side-channel attack is composed of two steps. In the first step (a *preparation step*), an attack process manipulates target cache blocks, which victim processes can change. Then in the second step (a *scanning step*), the attacker scans the previously manipulated cache blocks to monitor cache blocks changed by the target secret data accessed by the victim processes. The attacker can figure out whether a known cache block exists in the target cache level by using time-measuring instructions (e.g. *rdtsc* of x86 ISA).

Flush+Reload is a cache side-channel attack that can identify secret information by detecting loaded cache blocks indexed by secret data [1]. In the preparation step, the attack process removes target cache blocks by exploiting cache manipulation instructions such as *clflush* of x86 ISA. Then in the scanning step, the attack process measures the access latency to the cache blocks evicted during the preparation step. The attacker can identify the secret target data if a victim reallocates the evicted blocks to the target cache level. Since the Flush+Reload attacks rely on privileged cache manipulation instructions, the attacks can be available only for specific processors. However, several researchers disclosed that Flush+Reload attacks could be implemented by exploiting other types of instructions such as *dccisw* of ARM ISA [7], [8].

Prime+Probe utilizes an eviction set, which is a group of data blocks that can be mapped into the same cache set [2]. The Prime+Probe attack can be designed as follows. The attacker first fills the target sets of a target cache level with a prepared eviction set. Then a victim process can allocate new cache blocks in the target cache level by accessing secret data. While the victim allocates new cache blocks, some primed blocks (i.e. the data blocks in the eviction set) are evicted from the target cache. The attacker can identify which blocks are evicted from the cache by measuring access cycles to the eviction set.

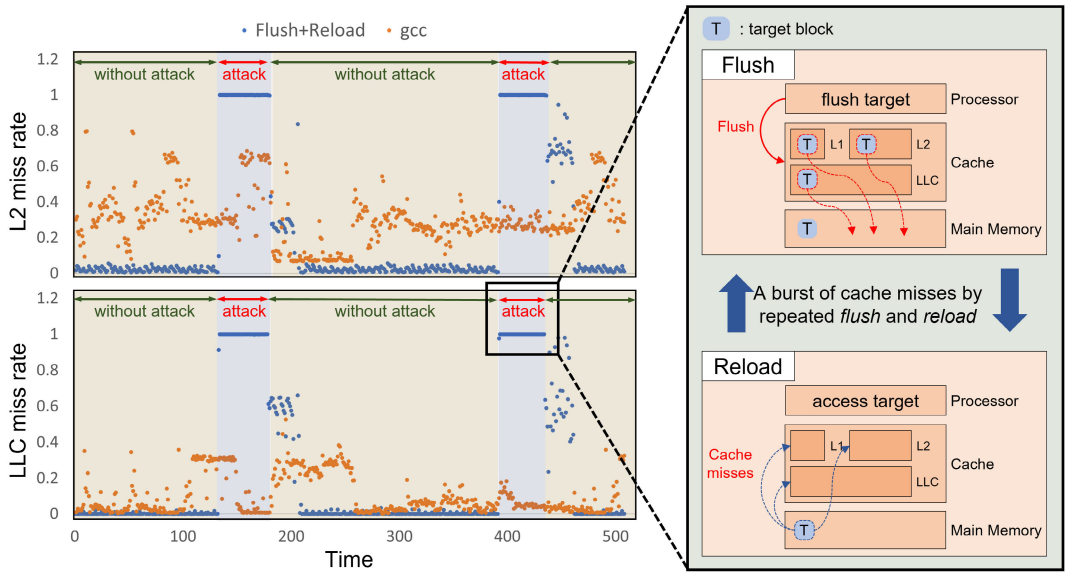


FIGURE 1. L2/LLC miss rates of a Flush+Reload attack and gcc.

III. LIMITATIONS OF PROFILING-BASED DETECTION

As introduced in the previous section, cache side-channel attacks manipulate target cache blocks to prime and/or monitor cache block sets. Hence, the attacks inevitably influence the cache performance counters while the attack processes are running. The existing profiling-based detection approaches monitor such unique performance changes to detect cache side-channel attacks. In this section, we analyze the existing attack detection approaches that rely on performance counters to reveal the limitations of such approaches.

A. BREAKDOWN OF DETECTION APPROACHES

The profile-based security attack detection approaches rely on unique performance footprints by attack processes. Figure 1 exhibits the changes in L2/LLC performance counts by a normal application (gcc from SPEC2017) and an attack process (Flush+Reload). The profiling-based detector collects performance counters (i.e. L2 miss rates and LLC miss rates) every 10 ms for each process. As shown in the figure, the hardware profiler exhibits relatively low miss rates on L2 cache (around 10% – 70%) and LLC (lower than 40%) while gcc is running. On the other hand, the Flush+Reload attack exhibits extremely high miss rates (nearly 100%) on both cache levels while an attack process is working. Note that Flush+Reload creates bursts of cache misses since the attack process accesses the cache blocks that were mainly flushed from the target cache. Hence, the detector can recognize the abnormally high cache miss rates by Flush+Reload to check whether the attack process is running.

As described in Figure 1, the profiling-based detectors exploit hardware performance counters (HPCs) in cache hierarchy to identify cache side-channel attacks. The detection approaches can efficiently collect the performance counters since system processes can read the values in HPC registers without heavy performance overhead. We can categorize the existing detection approaches as signature-based detection and anomaly-based detection.

In order to detect malicious processes, signature-based approaches detect unique footprints of cache-related performance counters created by cache side-channel attacks [20], [21], [22], [23]. Figure 1 describes an exemplar detection mechanism by signature-based detection. Namely, such detection approaches recognize the unique performance counts (e.g. extremely high L2/LLC miss rates) by attack processes. Researchers have proposed several signature-based detection approaches against security attacks. HexPADS monitors cache accesses and misses by each process to detect cache-based attacks that provoke high cache miss rates and significant cache misses [20]. RT-Sniper exploits several performance counters such as instruction counts, L2 misses, and LLC misses to detect cache side-channel attacks more accurately [21]. RT-Sniper also employs two-step monitoring phases to minimize performance overhead. Depoix et al. proposed a recurrent neural network (RNN) model using LLC access counts and miss rates of running processes [22]. CBA-detector employs multiple machine learning-based models and a self-feedback mechanism using PinTool in order to detect security attacks more precisely [23].

Anomaly-based detection approaches can recognize abnormal performance behaviors of the target applications (e.g. encryption/decryption libraries) that demand secure executions [24], [25], [26], [27]. Such detection methods monitor the performance counters of possible victim processes since the performance behaviors of the victim processes can be influenced while attack processes are working. For instance, an attack process evicts cache blocks that target victims may reference to monitor the victims' footprints. Then the victim processes can encounter unusual cache misses. In order

TABLE 1. Analysis of the existing profiling-based detection solutions.

Method	Detection type	Detectable attacks	Monitored performance counters	Detection algorithm
HexPADS [20]	Signature-based	FR	cache accesses, cache misses	threshold
RT-Sniper [21]	Signature-based	FR, PP	L2 hits, L2 misses, LLC hits, LLC misses	change point detection
Depoix et al. [22]	Signature-based	FR, PP	LLC accesses, LLC misses, instructions	threshold, neural networks
CBA-Detector [23]	Signature-based	FR, PP, FF	LLC accesses, LLC misses, instructions, cycles	MLP, decision tree, XGBoost
SpyDetector [24]	Anomaly-based	FR, PP, FF	cache hits/misses (victim), cache hits/misses (system)	logistic regression
CacheShield [26]	Anomaly-based	FR, PP	LLC misses, LLC miss rates	change point detection
WHISPER [25]	Anomaly-based	FR, PP, FF	L1d misses, LLC accesses, LLC misses, cycles	decision tree, SVM, random forest
CloudRadar [27]	Anomaly-based	FR, PP	cache hits, cache misses, instructions	threshold

to detect atypical performance behaviors of target applications, an anomaly-based detector has been trained using performance counters of the target applications. The detector can consider that the target applications are under attack if anomalous performance behaviors from the target processes are observed. In that case, the detector can halt the target applications to protect secret data.

To summarize, the existing profiling-based detection approaches count on hardware performance counters to pinpoint attack processes (*signature-based approaches*) or test whether victim processes are under attack (*anomaly-based approaches*). Table 1 lists the existing detection solutions that utilize hardware performance counters. The columns in the table exhibit detectable attacks, monitored performance counters, and detection algorithms by each detection method. In the *detectable attacks* column, FR, PP, and FF represent Flush+Reload, Prime+Probe, and Flush+Flush attacks respectively.

B. LIMITATIONS OF PROFILING-BASED DETECTION

Our analysis of the existing profiling-based detection approaches exhibits that the detection solutions rely on cache performance counters to detect popular cache side-channel attacks. It is because cache side-channel attacks exhibit unique performance behaviors in cache hierarchy and/or provoke remarkable cache performance changes of victim processes, as described in the previous section. For instance, a signature-based detection method can exploit the significant cache performance gap between attack processes (Flush+Reload) and normal applications (gcc), as shown in Figure 1. However, if the cache performance differences between normal applications and security attacks are insignificant, the detector may struggle to identify attack processes. In that case, the detector may fail to catch attack processes or incorrectly recognize normal processes as cache side-channel attacks (i.e. false positive detection).

Moreover, attackers can manipulate cache performance counts to conceal the unique cache behaviors of cache side-channel attacks. It is a possible attack scenario since attackers can create cache events unrelated to cache side-channel attacks within the same attack process. In the next section, we will present such attack scenarios that can perturb unique cache performance behaviors from cache side-channel attacks.

IV. PROPOSED VIZARD ATTACK SCENARIO

As motivated in the previous section, the existing profiling-based detection approaches may fail to detect security attacks if the target attacks exhibit eccentric performance behaviors. Especially, the detectors that target cache side-channel attacks will struggle to detect the attacks if malicious processes exhibit unexpected (e.g. low cache miss rates) cache behaviors. As such, attackers may intentionally include specific procedures that can counterbalance cache hit/miss rates in their attack processes to hide the unique cache behaviors from the cache side-channel attacks. In order to reveal the vulnerabilities of the existing profiling-based detection approaches to such attack scenarios, we study a cache side-channel attack method, called Vizard, that can bypass the existing detectors by counterbalancing cache hit/miss rates.

In this section, we describe how the proposed attack scenario can be employed in the existing cache side-channel attacks. As mentioned in Section II, a typical cache side-channel attack composes two separate steps, *preparation* and *scanning* steps, to check the cache state changed by victim processes. The attack process allows the target victim process to modify the *prepared* cache blocks before starting the *scanning* step. Namely, the attack process takes cycles to monitor cache state changes or wait until the target victim process accesses secret data before the scanning step. In this paper, we call the time gap between the preparation and the scanning steps a *attack window*. The proposed Vizard attack exploits this attack window to hide particular cache performance behaviors caused by the cache side-channel attack by creating opposite cache events. For instance, the Vizard attack can create cache hits for the target cache to neutralize high cache miss rates by Flush+Reload. Hence, the Vizard attack can evade the disclosure from the profiling-based detectors since it perturbs the cache performance counters utilized by the detection solutions within the attack window.

To summarize, the Vizard attack scenario can be designed as follows.

1) IDENTIFYING AN ATTACK WINDOW

In order to leak secret or private data, the attack process exploiting cache side-channel attacks allows the target victim process to access the primed cache blocks. Then the attack process monitors the changed cache states. The Vizard attack

utilizes the attack window between the *preparation* and the *scanning* steps of the attack process to hide the unique performance behaviors of the cache side-channel attack. Hence, the attackers should break down the interplay between the attack process (i.e. two required attack steps) and the victim process to identify the attack window.

2) EVALUATING THE ATTACK WINDOW SIZE

Once the attackers identify the attack window, they can evaluate the attack success rates by adjusting the attack window size. In order to leak secret data effectively, the attack process needs to set the appropriate attack window size between the preparation and the scanning steps. Note that the attacker cannot decode the target data if the scanning step is initiated before the victim accesses the secret data or after the victim's multiple accesses. The attack process can monitor the accesses by the victim process on the prepared cache blocks to wait until the victim accesses secret data. Moreover, the attack success rates usually go down as the attack window size increases. It is because the prepared cache blocks can be modified by other normal data transactions as well as access to the target secret data if the scanning step starts too late. Hence, in order to design an effective Vizard attack, attackers need to set allowable attack success rates for target victim applications, and then the attackers evaluate the attack window sizes that can guarantee the defined attack success rates. The attackers can set the largest attack window size that can achieve the allowable attack success rate as the attack window size of Vizard.

3) GENERATING PERFORMANCE EVENTS

In order to hide the unique performance behaviors of the exploited security attacks, the Vizard attack creates counterbalancing performance events during the attack window. Note that the attacker sets the largest attack window that can guarantee the allowable attack success rate for designing the Vizard attack. During this attack window, the attack process generates events that can perturb the target performance counters. For instance, to hide the extremely high cache miss rates by a cache side-channel attack, the Vizard attack can create many cache hits on the target cache level. The attacker should create the counterbalancing performance events effectively within the pre-defined attack window to hide the attack behaviors.

V. VIZARD ATTACK EXPLORATION

In this section, we explore the usage model of the Vizard attack using the Flush+Reload cache side-channel attack on RSA [1]. As described in the previous section, we design the Vizard attack scenario on the Flush+Reload attack to hide the unique cache access patterns by the Flush+Reload. We first analyze the mechanism of the Flush+Reload attack to identify the attack window. Then, we evaluate the key recovery rates of the attack by adjusting the size of the attack window in order to figure out the available attack window size. Finally, we explain how to design the cache hit generation module that can compensate for the high miss rates caused by the Flush+Reload attack.

A. ANALYSIS OF ATTACK MECHANISM

We use the Flush+Reload attack on RSA as an example of the Vizard attack scenario. Figure 2 briefly illustrates the cache side-channel attack mechanism on RSA cryptography [1]. In this attack, the attacker targets the square-and-multiply exponentiation functions that perform the exponential computation of $b^e (mod\ m)$ [28]. This approach is widely adopted in the popular RSA cryptography libraries since it exhibits less computational complexity of $O(log(e))$ compared to the original RSA algorithm that exhibits the complexity of $O(e)$. As shown in the code snippet in Figure 2 (see ①), the RSA encryption executes the diverged function sequences based on the bit value of each key digit. Namely, RSA executes *square–modulo–multiply–modulo* instructions if the bit value of the processed key digit is 1. Otherwise, the RSA function performs *square–modulo* instructions only if the bit value is 0.

Attackers can exploit such diverged instruction sequences of RSA to leak the encryption key using the Flush+Reload cache side-channel attack. The instructions that perform the RSA encryption can be found in a shared cache (usually LLC in most processors) if inclusive cache hierarchy is employed (② in Figure 2). In order to track the instruction sequences performed by the victim (i.e. RSA encryption) process, the attacker applies the Flush+Reload attack to the target instruction data in the shared cache. Namely, the attack process flushes the target instruction data from the shared cache to make the instructions evicted from the private caches (L1 instructions cache and L2 cache of most processors) in the core that is executing the RSA cryptography. This is the *preparation* step of the attack process. Then, the victim fetches the required instructions based on the bit value of the encryption key. The victim process sends the requests for the missing instructions to the private L1 instruction cache, then the requested instruction data are also allocated to the shared cache by the inclusive cache policy. By scanning the cache blocks flushed during the *preparation* step in the shared cache, the attacker can figure out which instructions are requested by the victim process (③). This is the *scanning* step of the Flush+Reload attack. Finally, the attacker can figure out the bit value of the key digit by decoding the instruction sequences referenced by the victim process (④).

In order to identify available attack windows in the target security attacks, we analyze the Flush+Reload attacks that prove the function sequences in the RSA encryption process. Figure 3 exhibits an example of the RSA function sequences (i.e. the sequences of *square*, *multiply*, and *modulo* functions) probed by the Flush+Reload attacks. The X-axis of the graphs represents the time (in cycle number) of the function sequences. The attacker checks the executions of the square, multiply, and modulo instructions by exploiting the Flush+Reload attacks as described in the previous paragraph. Each dot in the graphs represents the try of the Flush+Reload attack to check the execution of the target

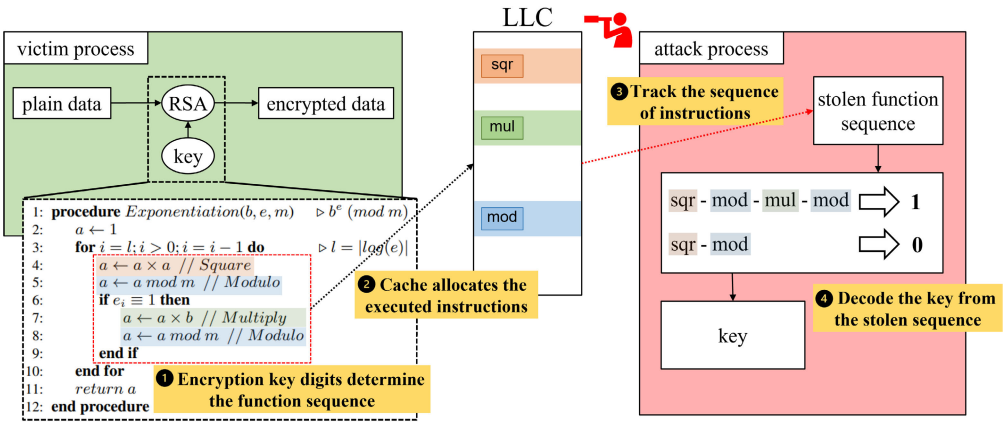


FIGURE 2. Cache side-channel attack on RSA encryption.

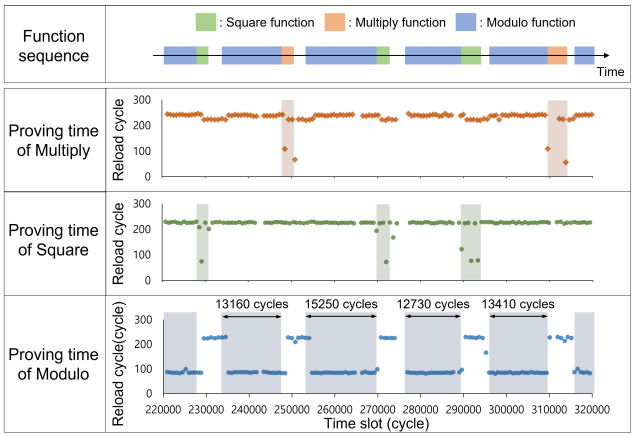


FIGURE 3. Proving RSA function sequences using Flush+Reload.

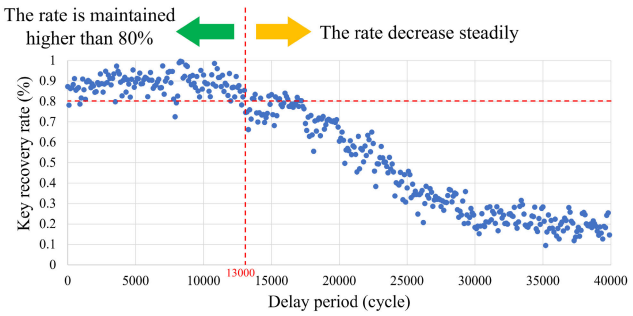


FIGURE 4. Key recovery rate by attack window size.

function. The Y-axis of each graph represents the reload time (in cycles) of the Flush+Reload attack. If the target function has been executed, the reload time of the Flush+Reload attack is low since the target instructions are found in LLC. As shown in the Figure, the execution times of *square* and *multiply* functions are relatively short (around 2000–3000 cycles) thus two or three attack tries exhibit low reload cycles. On the other hand, the *modulo* functions exhibit long execution time, thus 15–20 consecutive attack tries exhibit low reload cycles. Such frequent attack tries are redundant since only one or two probes are sufficient for detecting the executions of the modulo function. Consequently, the Vizard attack can exploit the Flush+Reload attacks towards the modulo function as attack windows for perturbing cache performance counters.

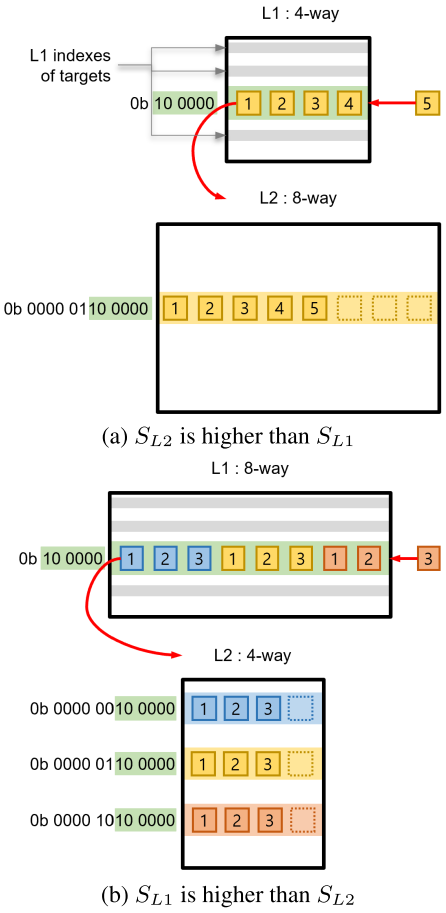
B. DECIDING ATTACK WINDOW SIZE

For the Vizard attack scenario, we define the time gap between the *preparation (flush)* and the *scanning (reload)* steps targeting RSA's square functions as an attack window. Now, we describe how to decide the appropriate attack window size for an effective Vizard attack. Figure 4 exhibits the key recovery rates of the Flush+Reload attacks on the RSA encryption by the attack window size. For this evaluation, we adjust the size of the attack window by inserting sleep instructions between *flush* and *reload* steps of the attacks, then we evaluate the key recovery rates by the sleep cycles.

Our evaluation results reveal the recovery rates by the Flush+Reload attack are high even if the attack window size is large. As shown in Figure 4 the high key recovery rates (over 80%) are observed until the attack window size is increased up to 13,000 cycles. Then the recovery rates go down steadily if the attack window size increases over 13,000 cycles. Based on our evaluation results, we set the attack window size of the Vizard attack as 13,000 cycles since the Flush+Reload attacks on the RSA encryption guarantee high recovery rates even if the *reload* phase is delayed. Note that the Vizard attack can utilize more cycles for perturbing performance counters if a larger attack window is configured. However, the attack may exhibit lower success rates if the Vizard attack applies too large attack windows.

C. PERTURBING PERFORMANCE COUNTERS BY GENERATING CACHE HITS

In this section, we describe how to perturb performance counters for hiding unique cache performance behaviors. Since we take the Flush+Reload attacks that exhibit extremely high



(a) S_{L2} is higher than S_{L1}

(b) S_{L1} is higher than S_{L2}

FIGURE 5. Cache block allocations by CHG with respect to set associativity.

miss rates in L2 and LLC as an example, Vizard's counterbalancing procedure creates many cache hits within the defined attack window. Note that the Vizard attack should create cache hit events as many as possible within the limited attack window period. As such, we will explain how to create a bunch of cache hits effectively in the target cache levels. We call this procedure that creates many cache hit events a *cache hit generator* (CHG).

Simply, we may generate cache hits by referencing the same data element repeatedly. However, such approaches cannot be used for increasing the cache hit counts in L2 and LLC since cache requests are filtered by L1 caches. Thus CHG needs to be elaborately designed to manipulate L2/LLC performance counters. For this purpose, CHG constructs a block set that exists in the target cache level, not in the higher-level caches. For instance, in order to increase cache hit counts in L2, CHG loads data elements in L2, but the loaded data needs to be evicted from L1. Namely, CHG allocates cache blocks exclusively in the target cache level to generate cache hit events.

CHG employs the following mechanism in order to allocate the necessary cache blocks to the target cache level exclusively. Let us assume CHG allocates the cache blocks to the cache level L exclusively, thus the allocated cache blocks are not found in the higher-level (i.e. level $L-1$) cache. First CHG prepares N data elements that can fill the cache blocks in the same cache set of the higher-level cache. Note that N should be larger than the associativity (S_{L-1}) of the higher-level cache. Then CHG allocates the S_{L-1} elements out of the prepared N elements, thus all blocks in the target set of the higher-level cache are occupied. CHG continues allocating the prepared data elements to make the one of occupied blocks evicted from the higher-level cache. Thus the evicted block can be found in the level L cache, not in the higher-level cache. Finally, CHG accesses the evicted block to increase the cache hit counter of the level L cache.

Figure 5 illustrates how CHG allocates cache blocks exclusively to generate hit events of the target cache level. We assume CHG targets the hit counter of the L2 cache, thus the cache blocks need to be evicted from the L1 cache. We explain the two different set associativity configurations between L1 and L2 caches. Figure 5a depicts the behavior of CHG when the set associativity of L2 is higher than L1. We assume the associativity of L1 and L2 is 4 and 8 respectively and caches are inclusive. In this example, CHG prepares 5 data blocks that can fill the same cache set of L1. When the *block 5* is allocated, one of the occupied cache blocks (*block 1*) is evicted from L1. Then this evicted block is found in L2 only when CHG accesses *block 1*. Note that CHG can prepare more blocks to allocate more evicted blocks only in L2 in order to create cache hits more efficiently. If the associativity of L1 is higher than L2, CHG prepares the multiple data block sets assigned to different sets of L2 as shown in Figure 5b. In this figure, the blocks colored differently (blue, yellow, and red colored) are allocated to different sets in L2. If CHG allocates *red-colored block 3* in L1, *blue-colored block 1* is evicted from L1 thus it is found in L2 only.

CHG's mechanism for allocating cache blocks and generating cache hits can be tweaked for various cache configurations. We describe the required approaches for CHG by cache configurations as follows.

1) CACHE INDEXING

Some processors compute the indexes of L1 caches using virtual addresses in order to reduce the access latency of L1. Since CHG exploits the cache blocks allocated to the same cache set, CHG needs to figure out the address mapping rule if L1 caches are virtually indexed. In this case, we can get the address mapping information from the page map file (*/proc/self/pagemap* for Linux OS). If an operating system utilizes a huge page, all caches are physically indexed thus CHG can compute the indexes of data blocks using physical addresses.

2) CACHE REPLACEMENT POLICY

CHG can make more blocks evicted from the high-level cache consecutively to allocate more blocks in the target cache. If the higher-level cache employs the deterministic cache replacement policies such as lease-recently used (LRU) or

TABLE 2. Cache configurations of Intel i7-10700.

cache	size	associativity	replacement	inclusion
L1I	32 KB	8-way	pseudo-LRU	-
L1D				
L2	256 KB	4-way		inclusive
L3 (shared)	16 MB	16-way		

first-in-first-out (FIFO), CHG can easily control the number of evicted cache blocks since cache blocks can be evicted in the order or allocation. On the other hand, CHG cannot figure out how many blocks will be evicted from the higher-level cache if a random replacement policy is employed. Since the evicted cache line is selected randomly, CHG cannot disclose which blocks are evicted and found in the target cache exclusively. In this case, CHG prepares a larger data set that will be allocated to the same cache set in the higher-level cache. By accessing more data elements allocated to the same cache line, CHG can make more blocks evicted from the higher-level cache, thus, it can increase cache hit rates of the target cache.

3) EXCLUSIVE CACHE

In Figure 5 that depicts the mechanism of CHG, we assume caches employ inclusive cache policy. However, if exclusive caches are employed, the evicted block from the higher-level cache is newly allocated to the target cache only. Since the main purpose of CHG is allocating the cache blocks in the target cache exclusively, CHG works in the same way even in the exclusive caches.

VI. EVALUATION

A. EXPERIMENT SETUP

In order to evaluate the Vizard attack scenario, we implement CHG in the Flush+Reload attack to make the CHG procedure work within the attack window. We evaluate the Vizard attack scenario on the testbed that equips Intel i7-10700 CPU (16-threads/8-cores running at 2.9 GHz) with 32GB DDR4 main memory. The testbed system runs Ubuntu 20.04.4 OS with Linux kernel 5.15.0-46. Table 2 summarizes the cache configurations of the Intel i7-10700 CPU.

As mentioned in Section III, most profiling-based detectors utilize high L2 and L3 cache miss rates to detect cache side-channel attacks. Thus in the Vizard attack, CHG is designed to create counterbalancing cache hit events in L2 and L3 caches. In order to measure the performance counters changed by the Vizard attack, we design the performance counter collector that reads the performance counters of the target process. The collector reads the performance counters of L2 hits/misses and LLC hits/misses every 10 ms.

The Vizard attack is designed as described in Section V. The CHG procedures of the Vizard attack generate cache hits in L2 and LLC. Note that the Vizard attack needs to generate more cache hits to make the L2 and LLC miss rates of the attack lower. In order to create more cache hit events in L2 and LLC to make L2/LLC miss rates lower, the Vizard attack requires a larger attack window, which will decrease the attack success rates as we investigated in Section V-B.

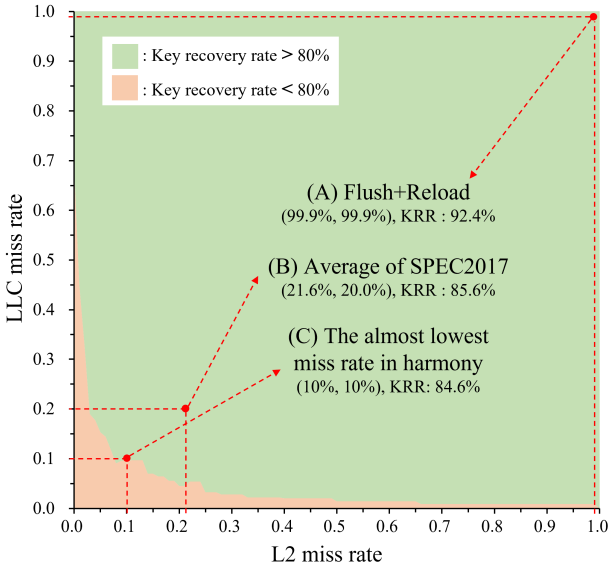


FIGURE 6. Key recovery rate of the Flush+Reload attack w.r.t. L2/LLC miss rate manipulated by the Vizard.

B. ATTACK SUCCESS RATE BY VIZARD's COUNTERBALANCE

As described in Section IV, the Vizard can manipulate the performance counters for cache hierarchy to camouflage high L2/LLC miss rates by cache side-channel attacks. Figure 6 represents the L2 and LLC miss rates manipulated by the Vizard attack and the target key recovery rates by the Flush+Reload attack on RSA encryption. We set the target key recovery rate for this Vizard attack design as 80%. Note that we can set the Vizard's attack window size as large as 13000 cycles to guarantee 80% of the key recovery rate as explained in Section V-B. In the figure, the x-axis and the y-axis represent the L2 miss rate and LLC miss rate, respectively; thus, the (x, y) coordinates of a point on the graph represent the L2/LLC miss rates manipulated by the Vizard. In the figure, the green-colored area represents the points that can achieve higher than 80% of the key recovery rate, and the orange-colored area means the recovery rate is less than 80%. Point A exhibits the L2/LLC miss rates (99.9%, 99.9%) by the original Flush+Reload attack. By applying the Vizard attack scenario, the L2/LLC miss rates can be decreased to the point equivalent to the average L2/LLC miss rates of SPEC2017 benchmarks (point B, 21.6% and 20.0%). Note that the Vizard can create more hit events to decrease L2/LLC miss rates under 10% (point C); however, it requires a larger attack window (larger than 13000 cycles) that the recovery rate becomes lower than the target success rate. Our evaluation reveals that the Vizard can exploit the available attack window for the exemplar attack scenario (i.e. the Flush+Reload attack on RSA) to create many counterbalancing cache events to hide the unique cache performance behaviors of the attack process.

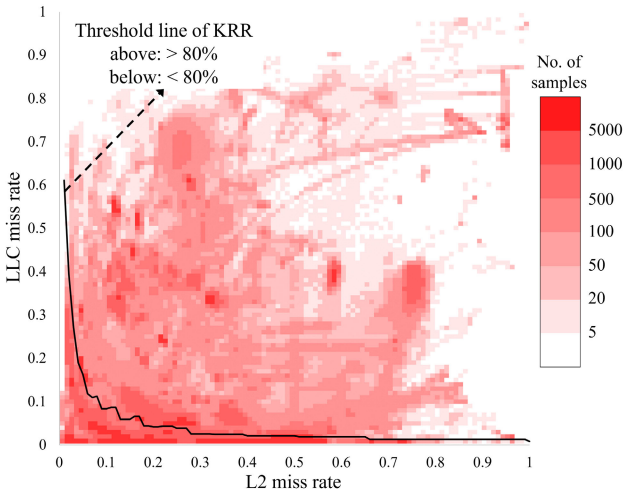


FIGURE 7. Heatmap for L2/LLC miss rates of SPEC2017 benchmarks (sampled every 10 ms).

The main purpose of the Vizard is to camouflage the unique cache performance behaviors of cache side-channel attacks by making the cache miss rates of the attack process similar to the cache performance levels of ordinary applications. In order to analyze the cache miss rates of normal applications, we measure the heatmap for L2/LLC miss rates of SPEC2017 applications as shown in Figure 7. We collect the benchmarks' L2/LLC miss rates every 10 ms on the testbed machine. Each rectangular region on the coordinate is partitioned by 1% of L2/LLC miss rates, respectively. The density of the red color on each region represents the number of samples. Thus, dense-colored regions represent that the corresponding L2/LLC miss rates are frequently observed in SPEC2017 applications. The black line on the coordinate represents the boundary of the 80% of key recovery rates by the Vizard attack (please see Figure 6).

As shown in Figure 7, we can observe that most samples of SPEC2017 benchmarks exhibit relatively low L2/LLC miss rates compared to cache side-channel attacks. More than 70% (2,314,444 out of total 3,242,094 samples) of the SPEC2017 samples exhibit L2 and LLC miss rates, both lower than 50%. The average L2 and LLC miss rates of SPEC2017 applications are 21.6% and 20.0%, respectively. 30.7% (994,154 samples) of the total SPEC2017 samples are included in both under-50% of L2/LLC miss rates region and the green-colored region of Figure 6. Our observation exhibits that the Vizard attack can efficiently perform cache side-channel attacks without sacrificing the attack success rate (i.e. the key recovery rate higher than 80%) even if the Vizard manipulates the L2/LLC miss rates of an attack process to make the miss rates similar to SPEC2017 applications. Profiling-based detectors may lower the threshold levels of L2/LLC misses to snipe at the Vizard attack, and then the detectors will incorrectly catch normal processes since the Vizard attack also exhibits low L2/LLC miss rates like ordinary applications.

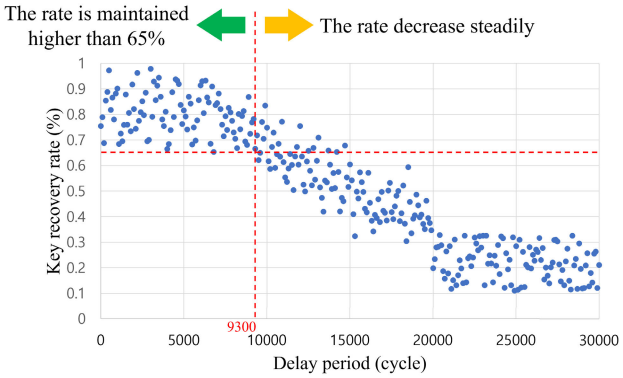


FIGURE 8. Key recovery rate by attack window size for a Prime+Probe attack.

C. EXPLORING PRIME+PROBE ATTACKS

Even though we evaluate the effectiveness of a Vizard attack scenario by using Flush+Reload attacks, the Vizard attack can be applied to other types of cache side-channel attacks. In this subsection, we evaluate the Vizard attack scenario combined with Prime+Probe attacks. Note that Prime+Probe-type attacks usually exhibit lower attack success rates compared to the Flush+Reload attacks since the Prime+Probe attacks can be designed without exploiting the privileged cache manipulation instructions such as *clflush* of x86 ISA. While we intensively describe the Vizard attack scenario applied to the Flush+Reload attacks in Section V, we briefly explain how to implement the Vizard attack combined with the Prime+Probe attacks on the RSA encryption using the design policy described in Section IV. We define the Vizard attack combined with Flush+Reload and Prime+Probe attacks as Vizard+FR and Vizard+PP, respectively.

1) ATTACK WINDOW SIZE

Vizard+PP can utilize smaller attack windows compared to Vizard+FR due to the native attack mechanism of the Prime+Probe attack. Note that the Prime+Probe attack takes more cycles priming target cache blocks, unlike the Flush+Reload attack, which requires one instruction per cache block to make cache blocks evicted from the target cache level. Namely, the Prime+Probe attack needs to load entire blocks in a target cache set to prepare cache blocks effectively. Considering modern processors configure LLC's set associativity as 12 to 16, the Prime+Probe attack requires much more instructions for priming one cache set, equivalent to removing one cache block for the Flush+Reload attack. Hence, Vizard+PP can exploit the smaller attack windows for generating counterbalancing cache events since the Prime+Probe attack takes much more cycles during the preparation step of the attack process.

Figure 8 exhibits the key recovery rates of the Prime+Probe attacks on the RSA encryption with respect to attack window sizes. Since the target victim application is the same, we define the Vizard attack window between the preparation and the scanning steps of the Prime+Probe attack as

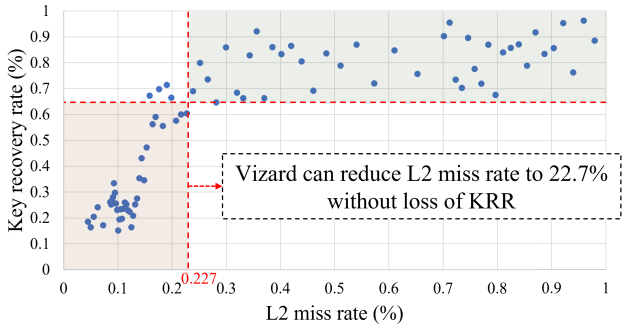


FIGURE 9. Key recovery rate of the Prime+Probe w.r.t. L2 miss rate manipulated by the Vizard.

described in Section V-A. Note that the Prime+Probe attack exhibits a wide range of key recovery rates (65%–100%) even if no additional delay cycles are added. It is because the Prime+Probe attack can be intervened by other processes more easily compared to the Flush+Reload attack. We can observe meaningful changes in the attack success rates when the attack window size is set as around 9300 cycles, which means Vizard+PP can create cache events within 9300 cycles of the attack window.

2) GENERATING CACHE EVENTS

Unlike the Flush+Reload attack, the scanning step of the Prime+Probe attack exhibits low cache miss rates in the target cache level and high cache miss rates in the higher-level caches. For instance, if the Prime+Probe attack primes the cache blocks in LLC (i.e. typically L3 cache), the attack process needs to probe the primed cache blocks in LLC during the scanning step. Thus, the cache requests generated by the Prime+Probe result in high miss rates in L1 and L2 caches to access LLC. Since normal applications also exhibit low miss rates in LLC, profiling-based detectors monitor the high miss rates in L1 or L2 to detect Prime+Probe attacks. Hence Vizard+PP evaluated in this paper counterbalances the L2 miss rates to hide the high cache miss rates caused by the Prime+Probe attacks.

Figure 9 shows the key recovery rates by the Prime+Probe attacks on the RSA encryption when Vizard+PP manipulates the L2 cache miss rates. In order to lower the L2 cache miss rates provoked by the Prime+Probe attacks, Vizard+PP generates cache hit events in L2. We regard 65% of the recovery rate as the marginal success rate of Vizard+PP since the Prime+Probe attacks exhibit 65%–100% of the key recovery rate without any delays between preparation and scanning steps. Our evaluation reveals that Vizard+PP can reduce the L2 miss rate of the Prime+Probe attack to 22.7% without sacrificing the key recovery rate. Considering the average L2 miss rate of SPEC2017 applications is 21.6%, our study exhibits that the Vizard attack scenario can also be applied to the Prime+Probe-type cache side-channel attacks to conceal the attack processes effectively.

D. BYPASSING PROFILING-BASED DETECTORS

We evaluate whether the Vizard attack can evade the detection mechanisms of profiling-based detectors. We study the performance of HexPADS and RT-Sniper against Vizard attacks and normal cache side-channel attacks (i.e. Flush+Reload and Prime+Probe attacks) as exhibited in Table 3. We configure the target cache miss rates of Vizard+FR as 10% for L2 cache and LLC respectively. We configure the target L2 miss rate of Vizard+PP as 30%. As described in Section III, HexPADS and RT-Sniper rely on the cache performance counters to detect cache side-channel attacks. The original version of HexPADS is configured to use 70% of cache miss rates as the threshold level for detecting suspicious cache behaviors [29]. HexPADS detects a certain process as a cache side-channel attack if the average cache miss rates of the process exceed the configured threshold level during one second of the sampling period. We change the HexPADS threshold lower than the original level to make HexPADS detect the security attacks that exhibit lower cache miss rates. RT-Sniper employs a cumulative sum algorithm for L2 and LLC miss rates to detect suspicious processes [30]. We configure RT-Sniper to collect cache performance counters at every 10 ms. We measure the correct detection performance (i.e. true-positive rate) of the profiling-based detectors by testing the Vizard attacks and the normal cache side-channel attack. We repeatedly run the attacks and the detectors on separate cores 1000 times to measure the number of detected cases. We also test the false-positive rate of the detectors using SPEC2017 benchmarks. In this evaluation, we test 46 SPEC2017 applications and count the number of applications detected as security attacks by the profiling-based detectors.

As shown in the table, Vizard attacks can effectively bypass the profiling-based detectors by manipulating the L2/LLC miss rates of the attack process. Note that RT-Sniper and HexPADS cannot detect Vizard+FR at all. Only RT-sniper detects Vizard+FR as a security attack once out of 1000 tries. Vizard+PP can also hide the Prime+Probe attacks effectively, thus the original version of HexPADS cannot detect Vizard+PP at all. RT-Sniper catches Vizard+PP 116 times out of 1000 tries, however 11.6% of the detection rate is extremely low. On the other hand, the detectors can always catch Flush+Reload and Prime+Probe attacks successfully. We also measure the false-positive rates by the detectors using SPEC2017 benchmarks. We set the threshold levels of HexPADS as 70%, 50%, and 30%. Note that HexPADS can detect the cache side-channel attacks with low threshold levels more effectively, however the false-positive rate can also be increased. Our evaluation exhibits that 36 out of 46 SPEC2017 applications are detected as security attacks by HexPADS when the threshold level is set as 30% of the LLC miss rate. However, Vizard+PP is not detected by HexPADS even though the threshold level is extremely low. Our evaluation exhibits HexPADS can catch Vizard+PP if HexPADS sets the 30% of cache miss rates as a threshold level for detecting security attacks. However, this is not a viable option

TABLE 3. Detection performance by modified HexPADS and RT-Sniper.

Applications		Detectors			
		HexPADS (70%, origin)	HexPADS (50%)	HexPADS (30%)	RT-Sniper
Attack application	Flush+Reload	100% [1)]	100%	100%	100%
	Vizard+FR	0% [1)]	0%	0%	0.1%
	Prime+Probe	96.8% [1)]	100%	100%	99.7%
	Vizard+PP	0% [1)]	47.3%	99.8%	11.6%
Normal application	SPEC2017	3 / 46 [2)]	9 / 46	36 / 46	3 / 46

1) Number of detected attacks / 1000
2) Number of applications detected as an attack / total number of applications

since HexPADS exhibits extremely high false-positive rates if the threshold level is configured as lower than 50%.

Even though we study only HexPADS and RT-Sniper to evaluate the effectiveness of the Vizard attack in this work, we can say Vizard can effectively evade other detection approaches that rely on performance counters. As shown in Figure 6, Vizard attacks can make L2/LLC miss rates lower than the average miss rates of SPEC2017 benchmarks. That means the profiling-based detectors cannot identify the Vizard attacks among normal processes by monitoring cache miss rates. Machine learning-based detectors may track the specific performance counter values to detect security attacks. However, the attackers that employ the Vizard attack scenario can randomly select the target miss rates in run-time within the allowable attack window ranges. Thus, the Vizard attacks cannot be identified by tracking the specific cache performance behaviors.

VII. RELATED WORK AND DISCUSSION

A. EVASIVE ATTACKS

Researchers have explored several attack mechanisms that can bypass the existing security attack detectors. Li et al. presented a modified Spectre attack, called Evasive Spectre, which can bypass the signature-based detectors that rely on specific performance footprints (e.g. LLC references/misses and retired branches/misses) of Spectre attacks [32], [36]. Evasive Spectre executes additional branch and cache access instructions to hide the specific performance features of Spectre-like attacks. The authors also proposed a delaying mechanism that can dilute the performance counters created by Spectre attacks. Pashrashid et al. introduced modified Spectre attacks to test the performance of their proposed detector [34]. In order to reveal the limitations of ML-based detectors, the authors presented the modified Spectre attack that contains a part of normal application codes and the expanded Spectre attack that delays the branch mistraining step by inserting NOP instructions. Whereas those researches present evasive attack mechanisms that can be applied to transient execution attacks, Vizard represents the attack approaches based on the general cache side-channel attacks to bypass profiling-based detectors.

Khasawneh et al. presented attackers can reverse-engineer existing hardware malware detectors to evade the catches from the detectors [33]. The authors mentioned that attackers could randomly insert specific instructions into existing malware codes in order to perturb instruction counts and architectural event data exploited by detectors. They stated the general approaches that can bypass hardware malware detectors. Jiang et al. presented an attack approach that can bypass anomaly-based detectors [35]. The authors proposed an attack approach that partitions original attack processes into many smaller parts and slows down the attack executions in order to minimize the performance influences from security attacks. In this paper, we present an evasive attack that creates counterbalancing events within available attack windows to evade profiling-based detectors.

B. UTILIZING OTHER PERFORMANCE EVENTS

In this work, we propose the Vizard attack scenario that can bypass profiling-based security attack detectors. We apply the Vizard attack mechanism to popular cache side-channel attacks (i.e. Flush+Reload and Prime+Probe attacks) on RSA encryption applications. Since the profiling-based detectors exploit cache performance counters (i.e. cache misses) to detect cache side-channel attacks, Vizard generates cache hits to compensate for many cache misses generated by the security attacks. Even though we explore the Vizard attack by taking the cache-side channel attacks, the proposed Vizard attack scenario can also be applied to other types of security attacks. For instance, researchers presented several attack detection solutions that exploit branch-related or TLB-related events [36], [37], [38], [39], [40], [41]. Note that such detectors target security attacks that exhibit specific performance behaviors regarding branch predictors or TLBs. Based on the attack scenario described in Section IV, the Vizard attack scenario can be applied to different types of security attacks if attainable attack windows are available in the security attacks. For example, Vizard can be designed to generate many branch hits if a target detector monitors branch prediction misses to catch security attacks.

VIII. CONCLUSION

In this paper, we explore the limitations of profiling-based detectors by presenting the Vizard attack scenario that can hide cache side-channel attacks' particular cache performance behaviors. We reveal that most profiling-based detectors targeting popular cache side-channel attacks rely on performance counters that attack processes can also manipulate. In order to design the Vizard attack scenario, we first analyze cache side-channel attacks to identify available attack

windows, which are usually found between the preparation and the scanning steps of the cache side-channel attacks. Then we evaluate the allowable sizes of the attack windows that can guarantee high attack success rates. Finally, Vizard runs a cache events generator to manipulate cache performance counters within the pre-defined attack windows. In order to evaluate the effectiveness of the Vizard attack scenario, we implement the Vizard attack on the Flush+Reload and Prime+Probe, targeting an RSA encryption. Our evaluation reveals that the Vizard can effectively manipulate the cache performance counters of the attack process while maintaining attack success rates. Our experiments also exhibit that Vizard can bypass the profiling-based detectors. Our work represents that more than profiling-based detectors will be required for security attacks that can manipulate performance counters.

REFERENCES

[1] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack," in *Proc. 23rd USENIX Secur. Symp. (USENIX Secur.)*, 2014, pp. 719–732.

[2] 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.

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

[4] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache attacks on mobile devices," in *Proc. 25th USENIX Secur. Symp. (USENIX Secur.)*, 2016, pp. 549–564.

[5] S. Briongos, P. Malogón, J. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *Proc. 29th USENIX Secur. Symp. (USENIX Secur.)*, 2020, pp. 1967–1984.

[6] 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.

[7] H. Cho, P. Zhang, D. Kim, J. Park, C. Lee, Z. Zhao, A. Doupé, and G. Ahn, "Prime+count: Novel cross-world covert channels on arm trustzone," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, 2018, pp. 441–452.

[8] Z. Kou, W. He, S. Sinha, and W. Zhang, "Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+evict," in *Proc. 58th ACM/IEEE Design Automat. Conf. (DAC)*, Dec. 2021, pp. 979–984.

[9] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 888–904.

[10] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Trans. Archit. Code Optim. (TACO)*, vol. 8, no. 4, pp. 1–21, Jan. 2012.

[11] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proc. 34th Annu. Int. Symp. Comput. Archit.*, Jun. 2007, pp. 494–505.

[12] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *Proc. 28th USENIX Secur. Symp. (USENIX Secur.)*, 2019, pp. 675–692.

[13] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 775–787.

[14] (2022). *Affected Processors: Guidance for Security Issues on Intel Processors*. Intel. Accessed: Dec. 19, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/processors-affected-consolidated-product-cpu-model.html

[15] (2016). *Introduction to Cache Allocation Technology in the Intel Xeon Processor E5 V4 Family*. Intel. Accessed: Dec. 19, 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/introduction-to-cache-allocation-technology.html

[16] T. Kim, M. Peinado, and G. Mainar-Ruiz, "STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud," in *Proc. 21st USENIX Secur. Symp. (USENIX Secur.)*, 2012, pp. 189–204.

[17] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "CATalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Mar. 2016, pp. 406–418.

[18] J. Shi, X. Song, H. Chen, and B. Zang, "Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw. Workshops (DSN-W)*, Jun. 2011, pp. 194–199.

[19] Y. Ye, R. West, Z. Cheng, and Y. Li, "COLORIS: A dynamic cache partitioning system using page coloring," in *Proc. 23rd Int. Conf. Parallel Archit. Compilation (PACT)*, 2014, pp. 381–392.

[20] M. Payer, "HexPADS: A platform to detect 'stealth' attacks," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2016, pp. 138–154.

[21] M. Song, J. Lee, T. Suh, and G. Koo, "RT-sniper: A low-overhead defense mechanism pinpointing cache side-channel attacks," *Electronics*, vol. 10, no. 22, p. 2748, Nov. 2021.

[22] J. Depoix and P. Altmeyer, "Detecting spectre attacks by identifying cache side-channel attacks using machine learning," in *Proc. 4th Wiesbaden Workshop Adv. Microkernel Oper. Syst. (WAMOS)*. Wiesbaden, Germany: RheinMain Univ. of Applies Sciences (HSRM), Jul. 2018, pp. 75–85.

[23] B. Zheng, J. Gu, J. Wang, and C. Weng, "CBA-detector: A self-feedback detector against cache-based attacks," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 5, pp. 3231–3243, Sep. 2022.

[24] Y. Kulah, B. Dincer, C. Yilmaz, and E. Savas, "SpyDetector: An approach for detecting side-channel attacks at runtime," *Int. J. Inf. Secur.*, vol. 18, no. 4, pp. 393–422, Aug. 2019.

[25] M. Mushtaq, J. Bricq, M. K. Bhatti, A. Akram, V. Lapotre, G. Gogniat, and P. Benoit, "WHISPER: A tool for run-time detection of side-channel attacks," *IEEE Access*, vol. 8, pp. 83871–83900, 2020.

[26] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth, "CacheShield: Detecting cache attacks through self-observation," in *Proc. 8th ACM Conf. Data Appl. Secur. Privacy*, 2018, pp. 224–235.

[27] T. Zhang, Y. Zhang, and R. B. Lee, "CloudRadar: A real-time side-channel attack detection system in clouds," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*, 2016, pp. 118–140.

[28] D. M. Gordon, "A survey of fast exponentiation methods," *J. Algorithms*, vol. 27, no. 1, pp. 129–146, Apr. 1998.

[29] *HexPADS, a Host-Based, Performance-Counter-Based Attack Detection System*. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/HexHive/HexPADS

[30] *RT-Sniper PoC Program*. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/takeangle/RT-Sniper

[31] *Mastik: A Micro-Architectural Side-Channel Toolkit*. Accessed: Nov. 16, 2022. [Online]. Available: https://github.com/0xADE1A1DE/Mastik

[32] C. Li and J.-L. Gaudiot, "Challenges in detecting an 'evasive spectre,'" *IEEE Comput. Archit. Lett.*, vol. 19, no. 1, pp. 18–21, Jan./Jun. 2020.

[33] K. N. Khasawneh, N. Abu-Ghazaleh, D. Ponomarev, and L. Yu, "RHMD: Evasion-resilient hardware malware detectors," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2017, pp. 315–327.

[34] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, "Fast, robust and accurate detection of cache-based spectre attack phases," in *Proc. 41st IEEE/ACM Int. Conf. Comput.-Aided Design*, Oct. 2022, pp. 1–9.

[35] J. Jiang, C. Soriente, and G. Karame, "On the challenges of detecting side-channel attacks in SGX," in *Proc. 25th Int. Symp. Res. Attacks, Intrusions Defenses*, Oct. 2022, pp. 86–98.

[36] C. Li and J. Gaudiot, "Detecting spectre attacks using hardware performance counters," *IEEE Trans. Comput.*, vol. 71, no. 6, pp. 1320–1331, Jun. 2022.

[37] B. Ali Ahmad, "Real time detection of spectre and meltdown attacks using machine learning," 2020, *arXiv:2006.01442*.

[38] M. Alam, S. Bhattacharya, and D. Mukhopadhyay, "Victims can be saviors: A machine learning–based detection for micro-architectural side-channel attacks," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 17, no. 2, pp. 1–31, 2021.

[39] M.-M. Bazm, T. Sautereau, M. Lacoste, M. Sudholt, and J.-M. Menaud, "Cache-based side-channel attacks detection through Intel cache monitoring technology and hardware performance counters," in *Proc. 3rd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, Apr. 2018, pp. 7–12.
[40] Z. Tong, Z. Zhu, Z. Wang, L. Wang, Y. Zhang, and Y. Liu, "Cache side-channel attacks detection based on machine learning," in *Proc. IEEE 19th Int. Conf. Trust, Secur. Privacy Comput. Commun. (TrustCom)*, Dec. 2020, pp. 919–926.
[41] A. Tang, S. Sethumadhavan, and S. J. Stolfo, "Unsupervised anomaly-based malware detection using hardware features," in *Proc. Int. Workshop Recent Adv. Intrusion Detection*, 2014, pp. 109–129.
[42] S. M. P. Dinakarrao, S. Amberkar, S. Bhat, A. Dhavlle, H. Sayadi, A. Sasan, H. Homayoun, and S. Rafatirad, "Adversarial attack on microarchitectural events based malware detectors," in *Proc. 56th Annu. Design Autom. Conf.*, Jun. 2019, pp. 1–6.

MINKYU SONG (Member, IEEE) received the B.S. degree in computer science and engineering from the Ulsan National Institute of Science and Technology (UNIST), Ulsan, South Korea, in 2017. He is currently pursuing the M.S. degree with the Department of Computer Science and Engineering, Korea University. His research interests include computer system architecture, computer security, and secure computing.

TAEWEON SUH (Member, IEEE) received the B.S. degree in electrical engineering from Korea University, Seoul, South Korea, in 1993, the M.S. degree in electronics engineering from Seoul National University, in 1995, and the Ph.D. degree in electrical and computer engineering from the Georgia Institute of Technology, Atlanta, GA, USA, in 2006. He is currently a Professor with the Department of Computer Science and Engineering, Korea University.

GUNJAE KOO (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, in 2001 and 2003, respectively, and the Ph.D. degree in electrical engineering from the University of Southern California, in 2018. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Korea University. Prior to joining Korea University, he was an Assistant Professor with Hongik University. He is a Senior Research Engineer with LG Electronics and a Research Intern with Intel. His research interests include computer system architecture and span parallel processor architecture, storage and memory systems, accelerators, and secure processor architecture.