Leaky Way: A Conflict-Based Cache Covert Channel Bypassing Set Associativity

Yanan Guo
University of Pittsburgh
USA
yag45@pitt.edu

Xin Xin
University of Pittsburgh
USA
xix59@pitt.edu

Youtao Zhang
University of Pittsburgh
USA
zhangyt@cs.pitt.edu

Jun Yang
University of Pittsburgh
USA
juy9@pitt.edu

***Abstract*—Modern x86 processors feature many prefetch instructions that developers can use to enhance performance. However, with some prefetch instructions, users can more directly manipulate cache states which may result in powerful cache covert channel and side channel attacks.**

In this work, we reverse-engineer the detailed cache behavior of `PREFETCHNTA` on various Intel processors. Based on the results, we first propose a new conflict-based cache covert channel named NTP+NTP. Prior conflict-based channels often require priming the cache set in order to cause cache conflicts. In contrast, in NTP+NTP, the data of the sender and receiver can compete for one specific way in the cache set, achieving cache conflicts without cache set priming for the first time. As a result, NTP+NTP has higher bandwidth than prior conflict-based channels such as Prime+Probe. The channel capacity of NTP+NTP is 302 KB/s. Second, we found that `PREFETCHNTA` can also be used to boost the performance of existing side channel attacks that utilize cache replacement states, making those attacks much more efficient than before.

***Keywords*-side channels, replacement policy, cache security**

I. Introduction

Modern processors often comprise many microarchitectural structures to enhance software performance. However, some of those structures are shared among applications, which may result in unintended information flows between applications. These information flows, when exploited by the adversaries, can enable powerful covert channel and side channel attacks. Cache timing covert channel and side channel attacks, or cache attacks for short, have been demonstrated to be extremely potent [4], [5], [10], [11], [13], [14], [16], [18], [19], [20], [24], [25], [26], [32], [37], [38], [40], [45], [53], [54], [64], [65], [67], [68], [70], [71], [72]. They are especially powerful primitives used in the more recently discovered transient execution attacks [8], [9], [28], [30], [49], [50], [55], [58], [59], [60], [61]. The execution of an application may cause various state changes to the cache that is shared with the attacker running on the same platform. The attacker can then learn these state changes through timing variations. Cache attacks can be used to surreptitiously transfer data (in the covert channel case), or infer secrets from a victim process (in the side channel case), bypassing sandboxes and software-level privilege boundaries. Cache attacks work on both the private cache and the last-level cache (LLC). However, LLC attacks are arguably more powerful than private cache attacks since the LLC is shared among physical cores.

Conflict-based cache attacks are an important class of cache attacks where the attacker (receiver) deliberately causes cache conflicts to learn the victim's (sender's) access pattern. For example, in Prime+Probe [26], [32], the attacker first primes a cache set by filling it with the attacker's cache lines, and then waits for the victim's execution: if the victim accesses her own cache line that is mapped to this set, one of the attacker's lines is evicted. Later the attacker probes the cache set and times the probing to learn whether the victim accessed her cache line. Conflict-based attacks are very practical since they usually do not assume any system features (such as page deduplication). However, when used as covert channels, their bandwidths are limited. This is because priming a cache set requires many accesses and takes very long to finish. For example, to build Prime+Probe on a 16-way associative LLC, priming the set needs about 16 cache accesses. In this paper, we seek to answer the following question:

Is it possible to cause cache conflicts without encountering the effort in priming the cache set?

x86 processors feature many prefetch instructions. Developers or compilers can use these instructions to inform the processor that a memory location will be accessed or modified soon. Then, the processor preloads the data (usually in cache line level) and places it closer to the CPU core, in order to accelerate future requests. `PREFETCHNTA` is one of the x86 prefetch instructions. When executing `PREFETCHNTA` on Intel processors with an inclusive LLC, the target data is brought into the requesting core's local L1 cache as well as the LLC [1], [2]. However, to avoid LLC pollution, the prefetched data is not placed into the most recently used position in the LLC set, and will be chosen for LLC replacement sooner than a regular cache fill. In this work, we reverse-engineer the detailed cache behavior of `PREFETCHNTA` on Intel processors and make three important observations. First, `PREFETCHNTA` installs the target cache line into the LLC set as the eviction candidate. Second, when `PREFETCHNTA` hits in the LLC, it does not update the age of this cache line in the LLC. Third, the execution time of `PREFETCHNTA` is related to the location of the target cache line in the memory hierarchy.

When filling a cache line into the LLC using PREFETCHNTA, it replaces the current eviction candidate in the set and then the prefetched line becomes the new eviction candidate. This means, when two processes both prefetch their own cache lines into the same LLC set, they will compete for the eviction candidate position (cache way), causing conflicts in one way of the LLC set. Based on this, we propose a new conflict-based cache covert channel, named NTP+NTP (Non-Temporal Prefetch). In this channel, the sender and receiver first agree on the LLC set for transmitting secrets. Then in each iteration of the transmission, the sender sends one bit by prefetching her cache line into the target LLC set (for "1") or not prefetching (for "0"). The receiver receives the bit by prefetching the receiver's cache line (which is also mapped into this target LLC set), and times the prefetch to determine if it is an LLC miss (for "1") or not (for "0"). If the sender prefetches her cache line into the LLC, it evicts the receiver's cache line that was prefetched into the same set; later the receiver's prefetch will miss in the LLC. We show that NTP+NTP has very high capacity as a conflict-based covert channel: on our Skylake processor, the capacity of NTP+NTP is 302 KB/s which is over 3× than the capacity of Prime+Probe. *To the best of our knowledge, NTP+NTP is the first conflict-based LLC covert channel that does not require priming the cache set.*

Although NTP+NTP is unlikely a side channel, we found that PREFETCHNTA can be used in many cache side channel attacks that are based on replacement state changes to make the attacks more efficient. This is because PREFETCHNTA makes it easier for users to manipulate cache replacement states. For example, Prime+Scope [42] is a cache attack proposed very recently. Prime+Scope achieves the highest-to-date temporal resolution for cache attacks, and is thus very powerful. However, this attack has strict requirements on the replacement state of the target LLC set. To satisfy the requirements, it uses a very long access sequence to prime the LLC set. On our Skylake processor, the priming comprises 192 cache references and takes about 1900 cycles to finish. This long priming step limits the attack from detecting frequent victim events. In contrast, when using PREFETCHNTA, the priming only needs 33 cache references and takes about 1000 cycles to finish, resulting in a much faster attack. In addition, using PREFETCHNTA makes cache conflicts occur more often and thus makes eviction set construction faster. In this work, we propose a new eviction set construction algorithm which significantly outperforms the state-of-the-art. The source code of our experiments can be found at https://github.com/PittECEArch/LeakyWay.

II. Background

A. Prefetch

Modern x86 CPUs use multiple levels of caches to store data that are frequently accessed. To further improve performance, data can be preloaded and placed closer to the CPU core (e.g., from the LLC to the L1 cache) before they are needed; this is usually referred to as *prefetch*. Prefetch can be performed in two ways. Hardware prefetch is usually implemented in cache hardware and is transparent to users. x86 processors support many hardware prefetchers such as the adjacent line prefetcher [23].

Different than hardware prefetch, software prefetch needs to be explicitly done by the programmer/compiler. Recent x86 CPUs offer many instructions for software prefetch, such as PREFETCHT0, PREFETCHT1, PREFETCHT2, and PREFETCHNTA [1], [3]. These instructions are used to hint the processor that a memory location is likely to be accessed or modified soon, then the processor preloads the corresponding data (usually in cache line level) into certain level(s) of cache, thereby accelerating future accesses to this data. For example, PREFETCHT0 preloads data into the requesting core's local L1 cache (and the LLC on some processors). Software prefetch is an important way to improve performance. Compilers sometimes automatically inject prefetch instructions to accelerate loops.

B. Cache Replacement Policy

When the CPU core loads a cache line that is not present in a cache level (i.e., cache miss), the cache line is usually filled to this cache level (into a certain set). If the set this cache line is mapped into is already full, one of the lines that are currently cached in this set will be evicted to make space for this new cache line. The cache replacement policy decides which line should be evicted, i.e., the *eviction candidate*.

LRU. LRU is one of the most widely used replacement policies as it provides high cache utilization and thus good performance. LRU always selects the least recently used cache line in a set as the eviction candidate. Thus, when using LRU, we need to track the age of each cache line in a set. For a w-way associative cache, $\log w$ bits are necessary to record the age of each way (cache line) in a set, for a total of $w \log w$ for each set. This makes tracking and updating the ages of cache lines very expensive in terms of storage and latency.

Pseudo LRU. Recent x86 CPUs use pseudo LRU algorithms to achieve high cache hit rate as well as maintain low age updating/tracking overhead. Typical Pseudo LRU algorithms include Tree-LRU [56] and Bit-LRU [33]. Prior work [10] has reverse engineered that recent Intel Core processors use *Quad-age LRU* for their LLCs. With this policy, each cache line in an LLC set is assigned with two bits to represent its age. Thus, the maximum (oldest) age for a cache line is 3, and the minimum (youngest) age is 0. The details of this policy are shown as follows:

- **Insertion policy.** When a cache line is filled into the LLC, its age is initialized as 2.[1]

[1] On early Intel processors (before Skylake), sometimes cache lines are inserted into the LLC with the age initialized as 3.

- **Replacement policy.** When replacement is necessary, Quad-age LRU searches all the ways in the target LLC set in order, and evicts the cache line that is stored in the first way with age 3; if such a way does not exist, it increases the age of every way by 1 and searches again.
- **Updating policy.** When an access request from the CPU core hits in the LLC, the age of the target cache line is reduced by 1 (if the age is 0 then it will not be changed).

Figure 1 shows an example of how the state of an LLC set changes with a sequence of CPU requests. In this figure, the replacement policy checks the cache lines in the set from the left to right, when looking for the eviction candidate.

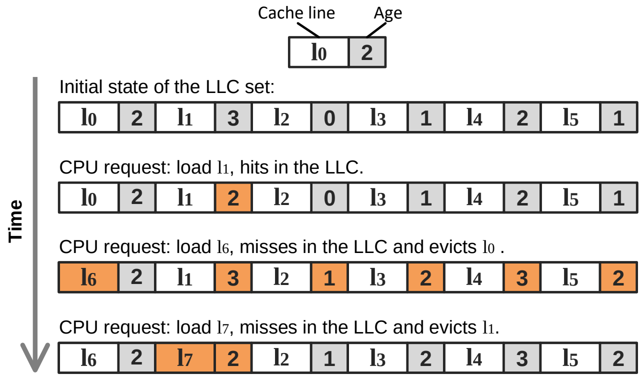

Cache line
Age
l_0 2
Initial state of the LLC set:
l_0 2 l_1 3 l_2 0 l_3 1 l_4 2 l_5 1
CPU request: load l_1, hits in the LLC.
l_0 2 l_1 2 l_2 0 l_3 1 l_4 2 l_5 1
CPU request: load l_6, misses in the LLC and evicts l_0 .
l_6 2 l_1 3 l_2 1 l_3 2 l_4 3 l_5 2
CPU request: load l_7, misses in the LLC and evicts l_1.
l_6 2 l_7 2 l_2 1 l_3 2 l_4 3 l_5 2
Time

Figure 1. The state change details of an LLC set upon CPU requests; changes after each request are highlighted.

C. Cache Attacks

There are typically two types of cache attacks. In the first type, the attacker *passively* monitors the contention on certain cache hardware (e.g., the ring interconnect [39] or L1 cache ports [36], [69]) to infer the victim's usage of it. Such attacks are usually referred to as contention-based attacks or stateless attacks. The other type is eviction-based attacks, which are also known as stateful attacks: the attacker *actively* brings a cache line/cache set to a certain state, and waits for the victim to execute (which potentially modifies the state); later the attacker checks the state of the line/set again to know whether the victim accessed it and thus changed the state. We further divide stateful attacks into two categories, based on whether they rely on the existence of shared data (between the attacker and victim). Note that in this overview, we only discuss cross-core attacks where the attacker and victim are located on different physical cores; most same-core attacks can be defended by disabling simultaneous multithreading (SMT), as done by many cloud providers [6], [12], [34].

Attacks with shared data. The required data sharing for these attacks is usually achieved with page deduplication or shared libraries. Flush+Reload [68] is a typical attack that relies on data sharing. In each attack iteration, the attacker flushes the victim's cache line (which is shared with the attacker) from all cache levels, and waits for the victim's execution. Later the attacker accesses this cache line and times the access to determine it is in cache or not: if it is cached (i.e., faster to access), it means the victim accessed this cache line and brought it back to cache, otherwise the victim did not access. Gruss et al. later proposed a variant of Flush+Reload, named Flush+Flush [18]. Instead of reloading the victim's cache line, it flushes this cache line again and times the flushing to learn the victim's behavior. This attack is stealthier than Flush+Reload because it does not generate any accesses (to the victim's cache line) and is then hard to detect using performance counters. Evict+Reload [19] is another variant of Flush+Reload where the attacker evicts the victim's cache line by building set conflicts instead of flushing it.

Instead of checking whether the victim brought the target cache line to cache, some attacks work by checking whether the victim changed the cache state of the target cache line. For example, in prior work [27], [67], the attacker monitors the changes to the coherence state of the target cache line. In Reload+Refresh [10], the attacker instead monitors the changes to the age of the target cache line (cf. Section II-B).

Attacks without shared data. Attacks that do not assume data sharing are arguably more practical: security-conscious operating systems/hypervisors may disable implicit data sharing across processes/virtual machines. Prime+Probe [26], [32] is one of such attacks. The attacker first primes the LLC set that the victim's cache line is mapped into, by filling the set with her own cache lines. This evicts the victim's cache line. Then after waiting for a period of time, the attack probes this set (by re-accessing the cache lines in the priming stage) and measures the probing latency: if the victim accessed her cache line and brought it back to the LLC, one of the attacker's cache lines was evicted and it now takes longer to probe; if the victim did not access then it is faster to probe. The prerequisite of Prime+Probe is having an inclusive LLC: when the victim's cache line is evicted from the LLC, it is also evicted from the private caches (if present). Yan et al. later proposed a directory Prime+Probe attack which builds set conflicts in the coherence directory [66], enabling Prime+Probe on platforms with non-inclusive LLCs. Very recently, Purnal et al. presented an optimization of Prime+Probe, named Prime+Scope. This attack has much higher resolution than Prime+Probe. Its details will be discussed in Section V.

Our goal. The above Prime+Probe type attacks are often referred to as *conflict-based* attacks. This is because in these attacks, the attacker evicts the victim's cache line by building set conflicts. As mentioned earlier, these attacks are practical and powerful. However, when used as covert channels, their bandwidths are usually very limited: in an attack iteration, to be able to build and observe set conflicts, the sender and receiver together need to access at least $w+1$ cache lines (in the same set), where w is the set associativity. In this paper, we aim at finding a way to build conflicts with less cache references, such that we can have faster cache attacks.

III. Characterizing The Non-Temporal Prefetch Instruction

A. *Non-Temporal Prefetch*

Among the prefetch instructions discussed in Section II-A, `PREFETCHNTA` works slightly different than the others: it minimizes the LLC pollution when fetching data into the cache hierarchy. To accelerate future accesses from the requesting core, `PREFETCHNTA` places the target cache line into the requesting core's private cache; with an inclusive LLC, this cache line has to be also brought into the LLC (if not already present). However, prefetching a cache line into the LLC may replace cache lines from other threads and degrade their performance. According to Intel [1], using `PREFETCHNTA` can reduce this disturbance to other data cached in the LLC: a cache line prefetched with this instruction will not be placed into the most recently used position (in the LLC set) and may be chosen for replacement faster than a regular LLC fill. Thus, when the target cache line is only accessed once in the entire execution path, the user should prefetch it using `PREFETCHNTA`. We reverse engineer the detailed cache behavior of `PREFETCHNTA` in this section, and will explain why this instruction raises severe security concerns in the next section. In the rest of this paper, we refer to "prefetch using `PREFETCHNTA`" as "prefetch".

Table I
The specifications of the tested processors.

Platform	Core i7-6700	Core i7-7700K
Microarchitecture	Skylake	Kaby Lake
Num of cores	4	4
Frequency	3.4 GHz	4.2 GHz
L1 associativity	8	8
L1 type	Private	Private
L2 associativity	4	4
L2 type	Private, non-inclusive	Private, non-inclusive
LLC associativity	16	16
LLC type	Shared, inclusive	Shared, inclusive

Experiment platform. The experiments in this paper are all performed on two Intel processors, Core i7-6700 and Core i7-7700K. The processor parameters are listed in Table I. In this section we only show the results on the Core i7-6700 processor due to limited space. Note that in this section we disable the hardware prefetcher to get accurate reverse engineering results. In the following sections, we enable the hardware prefetcher when evaluating the attacks for generality, and avoid triggering the hardware prefetcher using the techniques in prior work [18], [28].

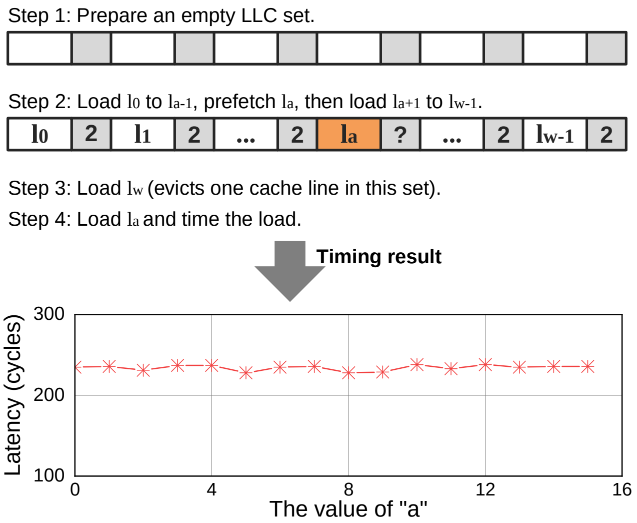


Figure 2. The experiment steps and results for verifying that prefetched data are evicted earlier than other data.

B. *Key Properties*

1) Insertion Policy: We first verify that a prefetched cache line is evicted faster/sooner than a loaded cache line in the same LLC set, using a four-step experiment. To prepare the experiment, we construct an eviction set, i.e., a group of cache lines that are all mapped to one specific set (the target set) in the LLC. This eviction set consists of $w+1$ cache lines where w is the set associativity of the LLC (16 for our processors). These cache lines are named l_0, l_1,..., l_w. As shown in Figure 2, this experiment consists of the following steps:

Step 1: We make the target LLC set empty. This can be achieved by loading the cache lines in the eviction set into the LLC and then flushing all of them with `CLFLUSH`.

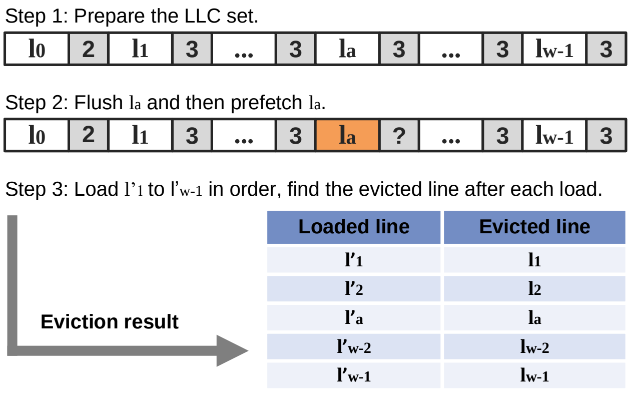
Step 2: We pick l_a from the eviction set, where $0 \leq a \leq w-1$. Then, we first load the cache lines before l_a in the eviction set into the LLC (l_0 to l_{a-1}, if $a \neq 0$), and then prefetch l_a. After the prefetch, we load the rest of the cache lines until the LLC set is full (l_{a+1} to l_{w-1}, if $a \neq w-1$). We add `LFENCE` after each load/prefetch operation to ensure that the cache lines are filled into the LLC in order.

Step 3: We load l_w into the LLC which evicts one of the existing cache lines in the target set.

Step 4: We load l_a and time the load to learn whether this prefetched line was evicted in Step 3. If l_a was evicted, it takes longer (typically more than 150 cycles) to load, otherwise it takes much shorter to load (less than 100 cycles).

We run the above experiment with a changing from 0 to $w-1$ and repeat the experiment 10000 times for each value of a. The average load latencies in Step 4 are shown in Figure 2: it always takes over 200 cycles to reload the prefetched line (l_a), meaning l_a was always evicted from the LLC in Step 3, regardless of its position in the set. This proves that a prefetched cache line is easier to be evicted than cache lines loaded into the LLC.

The above experiment indicates that a prefetched cache line is *distinctively inserted into the LLC*, so that the replacement policy will choose it to be evicted sooner than other cache lines. We hypothesize two possible hardware-level implementations to achieve this: 1) a prefetched cache line is inserted into the LLC set with the age initialized to be 3 instead of 2 (cf. Section II-B); 2) a prefetched cache line is inserted into the LLC set with age 2 as normal, but this line is flagged for "early eviction" in the LLC, i.e., a prefetched line and a line with age 3 are treated unequally by the replacement policy. As shown in Figure 3, we then perform the following experiment to know which option has more likely been chosen by Intel. In this experiment, we use two eviction sets that are mapped to the same LLC set (l_0 to l_w and l'_0 to l'_w).


Step 1: Prepare the LLC set.
l0 2 l1 3 ... 3 la 3 ... 3 lw-1 3
Step 2: Flush la and then prefetch la.
l0 2 l1 3 ... 3 la ? ... 3 lw-1 3
Step 3: Load l'1 to l'w-1 in order, find the evicted line after each load.
Eviction result

Loaded line	Evicted line
l'1	l1
l'2	l2
l'a	la
l'w-2	lw-2
l'w-1	lw-1

Figure 3. The experiment steps and results for learning the insertion policy of PREFETCHNTA.

Step 1: We prepare the target LLC set as shown in Step 1 of Figure 3. This can be achieved by first filling the set with l_w and l_1, l_2,..., l_{w-1} in that order, and then loading l_0 to evict l_w (cf. Section II-B).

Step 2: We flush l_a ($1 \leq a \leq w-1$) and then prefetch l_a. It is brought back to this flushed location.

Step 3: We load l'_1 to l'_{w-1} into the LLC in order and check which cache line in this set is evicted after loading each of them.[2]

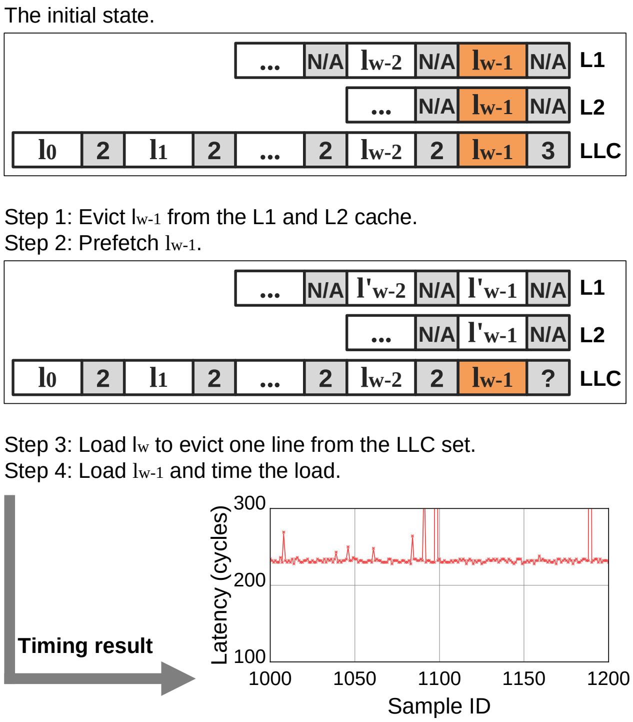

The initial state.
... N/A lw-2 N/A lw-1 N/A L1
... N/A lw-1 N/A L2
l0 2 l1 2 ... 2 lw-2 2 lw-1 3 LLC
Step 1: Evict lw-1 from the L1 and L2 cache.
Step 2: Prefetch lw-1.
... N/A l'w-2 N/A l'w-1 N/A L1
... N/A l'w-1 N/A L2
l0 2 l1 2 ... 2 lw-2 2 lw-1 ? LLC
Step 3: Load lw to evict one line from the LLC set.
Step 4: Load lw-1 and time the load.
Timing result
Latency (cycles)
300
200
100
1000
1050
1100
1150
1200
Sample ID

Figure 4. The experiment steps and results for learning the updating policy of PREFETCHNTA.

With each possible value of a, we run the experiment 10000 times. The eviction results in Step 3 are shown in Figure 3. We get the same results in each trial regardless of the value of a: when loading l'_1 to l'_{w-1} into this LLC set, l_1 to l_{w-1} are evicted in order (from the left to right in Figure 3). This indicates that the prefetched cache line (l_a) is treated equally as a cache line whose age is 3. Thus, we believe that a prefetched cache line is inserted into the LLC set with age 3 instead of being flagged for early eviction. This experiment also verified the replacement policy introduced in Section II-B.

Property #1: On an LLC miss, PREFETCHNTA inserts the target cache line into the LLC with the age initialized as 3.

2) *Updating Policy:* In this section, we study the cache behavior of PREFETCHNTA when the target cache line is already in the LLC (but not in the private cache). Specifically, we are interested in whether an LLC hit caused by PREFETCHNTA updates the age of the target cache line in the LLC like a load instruction (cf. Section II-B). In this experiment, we need two eviction sets: one for the LLC (l_0 to l_w) and one for the L1 and L2 cache (l'_0 to l'_w); l'_0 to l'_w are all mapped into the same L1/L2 set with l_0 to l_w, but different LLC sets. Then, we prepare the target LLC set as the initial state shown in Figure 4: the LLC set is filled with l_0 to l_{w-1}; the ages of l_0 to l_{w-2} are 2 but the age of l_{w-1} is 3. Thus, l_{w-1} is the eviction candidate in the LLC set. l_{w-1} may be also present in the L1/L2 cache. After the preparation, we run the following four steps (as shown in

[2] Checking if a cache line is evicted can be done by loading it and timing the load. Note that we should start over the experiment before checking the next cache line to avoid the noise caused by the measurement.

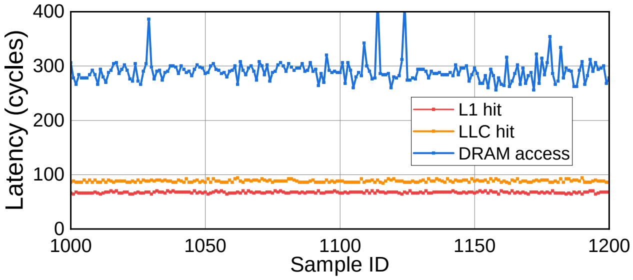


Figure 5. The execution times of PREFETCHNTA when the target data is the L1 cache, LLC, and DRAM.

Figure 4) to learn whether prefetching l_{w-1} updates its age from 3 to 2:

Step 1: We access l'_0 to l'_w multiple times to ensure that l_{w-1} is no longer present in the L1 and L2 cache.[3] This step is necessary because if l_{w-1} is present in the L1 or L2 cache, when prefetching l_{w-1}, the request will not reach the LLC and we cannot learn the updating policy in the LLC.

Step 2: We prefetch l_{w-1}. This results in an LLC hit and may update the age of l_{w-1} in the LLC.

Step 3: We load a new cache line l_w into the LLC which evicts one of the existing lines in the LLC set.

Step 4: We access l_{w-1} and time the access to learn whether it was evicted in Step 3: if l_{w-1} was not evicted, then PREFETCHNTA updated its age in Step 2 (from 3 to 2), otherwise PREFETCHNTA did not update the age in Step 2.

We repeat the above experiment 10000 times and Figure 4 shows a segment of the collected timing results (in Step 4). It always takes over 200 cycles to load l_{w-1}, meaning l_{w-1} was likely in DRAM before it's loaded in Step 4. Thus, we can safely conclude that PREFETCHNTA did not update its age (from 3 to 2) so it was chosen by the replacement policy and got evicted from the LLC in Step 3. Similarly, we have verified that PREFETCHNTA does not update the age of a cache line from 2 to 1, or 1 to 0 either, when hitting in the LLC.

Property #2: On an LLC hit, PREFETCHNTA does not update the age of the target cache line in the LLC.

3) Timing Variance: Prior work (e.g., [26], [32], [66], [68]) has shown that the execution time of a regular load instruction is related to the location of the target data in the memory hierarchy. Here we analyze whether PREFETCHNTA also has such timing variance. Specifically, we measure the execution time of PREFETCHNTA in three scenarios where the target cache line (l_t) is present in the L1 cache, not in the L1/L2 cache but present in the LLC, and not cached at all, respectively. The detailed operations for each scenario are as follows:

Algorithm 1: NTP+NTP Covert Channel

```
d_s: the sender's data (cache line) for transmitting signals
d_r: the receiver's data (cache line) for transmitting signals
message[n]: the n-bit long message to be transferred
Th0: the timing threshold for distinguishing prefetch hit and miss

Sender Algorithm

// Send 1 bit in each iteration.
for $i = 0$; $i < n$; $i++$ do
    synchronization();
    if message[i] == 1 then
        Prefetch d_s;
    else
        Do not prefetch;
    end
    wait_for_receiver();
end

Receiver Algorithm

// Detect 1 bit in each iteration.
for $i = 0$; $i < n$; $i++$ do
    synchronization();
    wait_for_sender();
    Prefetch d_r and time the prefetch;
    if prefetch_time > Th0 then
        Received a bit "1";
    else
        Received a bit "0";
    end
end
```

Scen. 1: We load l_t so that it is brought into the L1 cache; then we prefetch l_t and time the prefetch.

Scen. 2: We still load l_t first, as done in Scen. 1. However, before we prefetch l_t and measure the timing, we build set conflicts in the L1 and L2 cache to ensure that l_t is evicted from them.

Scen. 3: We first build set conflicts in the LLC to ensure that l_t is evicted from the entire cache hierarchy, and then we time the prefetch on it.

We test each scenario 10000 times and a segment of the collected timing results are shown in Figure 5. When the target cache line is present in the L1 cache, it takes around 70 cycles to prefetch it; it takes 90 to 100 cycles to prefetch when the cache line is only in the LLC, and over 200 cycles when the cache line is not cached at all.

Property #3: The execution time of PREFETCHNTA is related to the cache level of the target cache line.

IV. Prefetch-based Covert Channel

Based on the properties of PREFETCHNTA that are reverse-engineered in Section III, we build a new conflict-based cache

[3] $w+1$ cache lines are enough to evict l_{w-1} from both the L1 and L2 cache because $L1\_Associativity + L2\_Associativity < LLC\_Associativity$ on our processors.

covert channel. In this section, we first introduce the threat model, then discuss the details of this channel.

A. Threat Model

We use a similar threat model with previous conflict-based cache covert channels (e.g., [32]). We assume that the two essential parties for the channel, the sender and receiver are two unprivileged processes running on the same processor (but potentially different cores) with an inclusive LLC. We also assume that the sender and receiver are able to construct eviction sets for the LLC (e.g., using methods proposed in prior work [32], [42], [43], [62]). In addition, the sender and receiver should agree on the pre-defined channel protocols, including the synchronization, data encoding, target LLC set(s), and error correction protocols. Note that we do not assume any shared data between the sender and receiver, resulting in a more practical channel than channels relying on data sharing (e.g., [10], [18], [22], [27], [67], [68]).

B. NTP+NTP

1) Channel Protocol: When prefetching a cache line into the LLC, it replaces the current eviction candidate of the LLC set. According to the replacement policy explained in Section II-B, this eviction candidate is the first cache line in the set whose age is 3. Since the prefetched cache line's age is also set as 3 (cf. Property #1), it now becomes the first cache line in the set with age 3. This means that prefetching a cache line into the LLC evicts the current eviction candidate in the set, and then the prefetched cache line becomes the *new eviction candidate*. With this knowledge, we can build a covert channel where the sender and receiver communicate by competing (or not) for one way in an LLC set (i.e., the eviction candidate way). The sender and receiver can simply achieve this by prefetching their own cache lines which are mapped into the same LLC set. We name this covert channel NTP+NTP (Non-Temporal Prefetch+Non-Temporal Prefetch).

Basic channel protocol. In NTP+NTP, the sender and receiver first need to ensure that the sender's cache line d_s and the receiver's cache line d_r are mapped to the same LLC set, as done in prior work [26], [32], [42]. Then, the receiver prepares the channel by prefetching d_r into the LLC.[4] After this, the sender and receiver can communicate following Algorithm 1. One bit is transmitted in each iteration: the sender sends "1" by prefetching d_s into this target LLC set, or sends "0" by not prefetching. After this, the receiver receives the bit by prefetching d_r and times the prefetch. If the sender sends "1", then d_r should have been evicted from the LLC (by d_s); it takes longer for the receiver to prefetch. In contrast, if the sender sends "0", d_r is still in the LLC so it is faster for the receiver to prefetch. The sender and receiver can synchronize using the time stamp counters (TSCs).

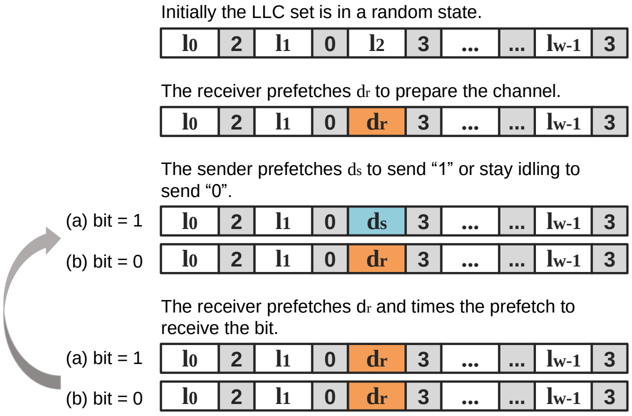

Initially the LLC set is in a random state.
l0 2 l1 0 l2 3 ... ... lw-1 3
The receiver prefetches dr to prepare the channel.
l0 2 l1 0 dr 3 ... ... lw-1 3
The sender prefetches ds to send "1" or stay idling to send "0".
(a) bit = 1
l0 2 l1 0 ds 3 ... ... lw-1 3
(b) bit = 0
l0 2 l1 0 dr 3 ... ... lw-1 3
The receiver prefetches dr and times the prefetch to receive the bit.
(a) bit = 1
l0 2 l1 0 dr 3 ... ... lw-1 3
(b) bit = 0
l0 2 l1 0 dr 3 ... ... lw-1 3

Figure 6. How the state of the target LLC set changes during the NTP+NTP covert channel.

The state change details in the target LLC set during the covert channel are shown in Figure 6. Before the sender and receiver start the covert channel, the target LLC set is in a random state, i.e., it is filled with random cache lines in random ages. When the receiver prefetches d_r for channel preparation, d_r becomes the first (left-most) cache line in the set with age 3, i.e., the eviction candidate. Thus, if now the sender prefetches d_s (to send "1"), it evicts d_r and then d_s becomes the new eviction candidate since it is now the first cache line with age 3. Therefore, when the receiver later prefetches d_r (for receiving the bit), it takes over 200 cycles to finish the prefetch (cf. Property #3). This prefetch also evicts d_s and then d_r is the eviction candidate again, i.e., this LLC set is ready for transmitting the next bit. In contrast, if the sender does not prefetch d_s in this iteration (to send "0"), then d_r is not evicted. Later when the receiver prefetches d_r, it will get an LLC hit (or a private cache hit) which takes less than 100 cycles. In addition, this prefetch does not update the age of d_r (cf. Property #2). Thus, d_r is still the eviction candidate and this LLC set is ready for the next iteration. In summary, the receiver's operation, prefetching d_r and timing the prefetch, is able to measure the bit from the sender in the current iteration, as well as reset the state of the target LLC set so that it is ready for transmitting the next bit.

Compared to Prime+Probe. Prior conflict-based covert channels such as Prime+Probe and its variants [32], [42] require the sender and receiver together access at least $w+1$ cache lines in each iteration to cause cache conflicts; w is the set associativity of the LLC. For example, if the sender sends a bit by loading (or not) a single cache line d_s, in each iteration the receiver needs to prime the target LLC set (by accessing at least w cache lines) to evict d_s and get ready for the next iteration. This is because after the sender loads d_s, it may become the youngest cache line in the target LLC set.

[4] We assume that the target set does not have empty ways which is true for most cases. The receiver can also prepare an eviction set and load it before the channel starts to ensure there is no empty way.

To evict d_s, the receiver needs to first access all other $w-1$ cache lines in the set to "refresh" their ages and make d_s the oldest cache line in the set. Then, the receiver accesses a cache line that is not present in the LLC, causing set conflict and thus evicting d_s.

In NTP+NTP, the sender inserts d_s into the target LLC set as the oldest cache line. Thus, the receiver is able to evict it using only *one operation*. Essentially, the sender and receiver can use PREFETCHNTA to bypass the w-way associativity of the LLC and use it as a one-way associative LLC. This results in much more efficient LLC conflicts and thus a faster covert channel.

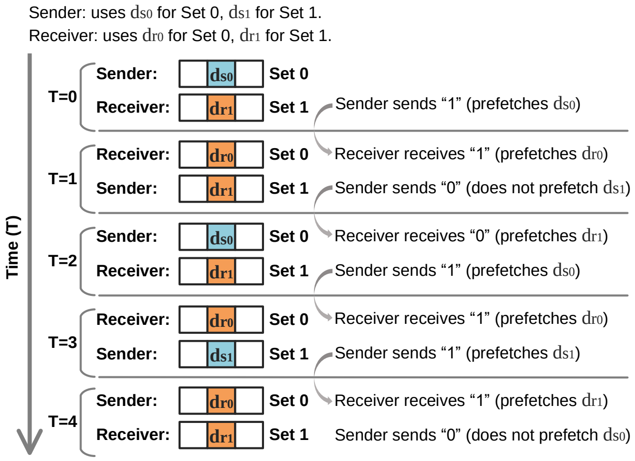

Sender: uses ds0 for Set 0, ds1 for Set 1.
Receiver: uses dr0 for Set 0, dr1 for Set 1.
Time (T)
T=0
Sender: ds0 Set 0
Receiver: dr1 Set 1
Sender sends "1" (prefetches ds0)
T=1
Receiver: dr0 Set 0
Receiver receives "1" (prefetches dr0)
Sender: dr1 Set 1
Sender sends "0" (does not prefetch ds1)
T=2
Sender: ds0 Set 0
Receiver receives "0" (prefetches dr1)
Receiver: dr1 Set 1
Sender sends "1" (prefetches ds0)
T=3
Receiver: dr0 Set 0
Receiver receives "1" (prefetches dr0)
Sender: ds1 Set 1
Sender sends "1" (prefetches ds1)
T=4
Sender: dr0 Set 0
Receiver receives "1" (prefetches dr1)
Receiver: dr1 Set 1
Sender sends "0" (does not prefetch ds0)

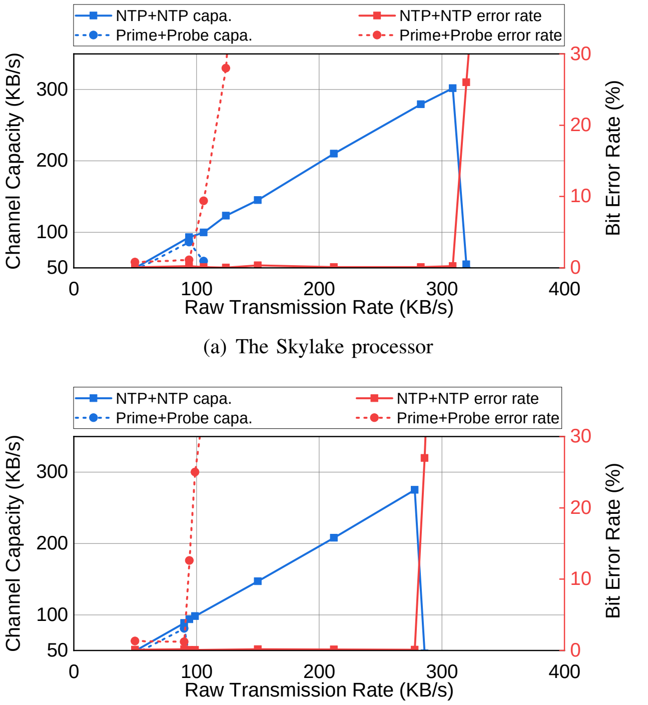
Figure 7. The operations of the sender and receiver in each iteration of NTP+NTP, when using two LLC sets; the receiver always detects the bit sent in the last iteration instead of the current iteration.

2) Channel Capacity: We implement NTP+NTP and Prime+Probe on two Intel processors (as listed in Table I) to test their bandwidths. For Prime+Probe, we use the example implementation discussed above: the sender accesses (or not) one cache line, and the receiver primes with w cache lines.

Table II
THE MAXIMUM CHANNEL CAPACITIES OF NTP+NTP AND PRIME+PROBE.

Platform	Skylake	Kaby Lake
NTP+NTP	302 KB/s	275 KB/s
Prime+Probe	86 KB/s	81 KB/s

The bandwidth of NTP+NTP is limited when using one target LLC set: if the cache line in an LLC way is in-flight (e.g., waiting for the memory response), this cache line cannot be evicted regardless of its age. This means d_r cannot evict d_s if d_s is still in-flight when the prefetch request of d_r reaches the LLC. Thus, we need to space out the prefetches from the sender and receiver (in each iteration). To avoid the slowdown caused by the spacing, we use two LLC sets in NTP+NTP and let the sender and receiver access different sets in each iteration. As shown in Figure 7, the receiver is always detecting the bit that was sent one iteration earlier. For fair comparison, we also use two sets in Prime+Probe. However, we do not use the sets as in Figure 7 since it does not benefit Prime+Probe much. Instead, we just use the two sets to transfer two bits in each iteration.



(a) The Skylake processor

(b) The Kaby Lake processor

Figure 8. The capacities and bit-error-rates of NTP+NTP and Prime+Probe.

We measure the channel capacities and bit error rates of both channels, under different transmission intervals. Although the raw transmission rate increases when decreasing the transmission interval, the bit error rate may also increase, especially when the interval is too short. To find the best transmission rate, we use the channel capacity metric (as in [39], [41]). This metric is computed by multiplying the raw transmission rate with $1-H(e)$, where e is the bit error rate and H is the binary entropy function. The results are shown in Figure 8. The bit error rates of both channels stay low (lower than 0.5% for NTP+NTP, 1.5% for Prime+Probe) and are almost constant, when the raw transmission rate is under a threshold (e.g., 304 KB/s for NTP+NTP in Figure 8 (a)). Thus, the channel capacity increases proportionally to the raw transmission rate. It reaches the peak when the raw transmission rate is around this threshold. Beyond this threshold, the increasing error rate causes a decrease in the channel capacity. The peak capacities of the two channels are summarized in Table II.

3) Channel Reliability: Similar to prior conflict-based covert channels, NTP+NTP is also affected by noise from other processes accessing data mapped to the target LLC set. For example, in a transmission iteration, although the victim sends "0" by not prefetching d_s, the receiver may receive "1" if other processes access their data and evict d_r, i.e., a false positive occurs.

This problem can be solved by using a more reliable data encoding method [26], [32], [35], rather than the very simple method in Algorithm 1. For example, multiple LLC sets can be used to send one bit. Note that the error caused by other processes' accesses in one attack iteration will not affect the next iteration: once the receiver prefetches d_r, d_r is the eviction candidate again. If other processes flush their data in the target LLC set, it will create empty ways, which can also impact the performance of NTP+NTP. However, `CLFLUSH` is rarely used in daily applications [44], [63], [68], and this problem can also be avoided by using a more reliable channel encoding method.

V. Prefetch-based Side Channel Attacks

NTP+NTP introduced in the last section is unlikely a side channel because the sender is transmitting the signal by "prefetching (or not) a cache line". In other words, the attacker (receiver) can only detect the victim's (sender's) prefetch patterns on a cache line, resulting in very limited attack opportunities to normal applications. However, the properties of `PREFETCHNTA` reverse-engineered in Section III make it much easier for users to manipulate the replacement states (ages) of cache lines in the LLC than before. Thus, attackers can also use `PREFETCHNTA` to improve the existing cache attacks that are based on cache replacement states, making them more efficient and accurate. In this section, we use two cache attacks that were proposed very recently as examples to show how they can benefit from using `PREFETCHNTA`.

A. Prime+Scope with PREFETCHNTA

1) Prime+Scope: Prime+Scope [42] proposed in 2021 is an LLC attack based on set conflicts. Prime+Scope is similar to Prime+Probe, but it has much higher temporal resolution. In each iteration of Prime+Scope, the attacker first primes the target LLC set with a *special pattern* to ensure two things. First, the target LLC set is occupied by the attacker's cache lines. Second, the current eviction candidate (a.k.a. the scope line, l_s) in the LLC set is also present in the attacker's private cache. Then, the attacker repeatedly accesses the scope line and times the access to detect the victim's access to her own cache line (which is also mapped to this LLC set). When the victim has not yet accessed her cache line in the current iteration, the attacker's accesses to l_s always hit in the private cache; once the victim accesses her cache line and brings it to the LLC, l_s is evicted and the attacker reaches an LLC miss. Then, the current iteration ends; the attacker primes this set again and moves to the next iteration. Note that the attacker can repeatedly access l_s without disturbing its replacement state in the LLC and changing the eviction candidate. This is because private cache hits do not update the replacement state of the LLC copy.

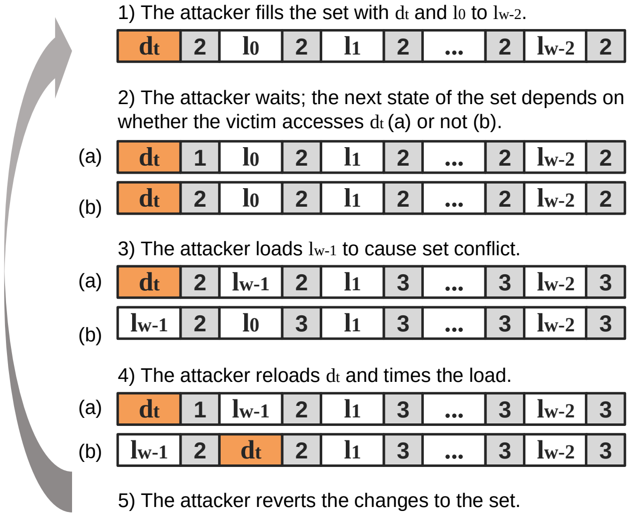
Prime+Scope is an important attack because it has very high temporal resolution. On our processors, loading a cache line that is in the private cache and timing the load together only take around 70 cycles. Thus, with Prime+Scope, the attacker can locate the victim's access in the time domain with a granularity of 70 cycles. For example, the attacker can know that the victim's access happened when $70 < current\_time < 140$ or when $140 < current\_time < 210$. In comparison, the resolution of Prime+Probe is over 2000 cycles [42].

```
/* evset is the eviction set used for priming
   */
/* the scope line addr is in evset[0] */
for(i = 0; i < 3; i++) {
    for(j = 0; j < 13; j+=4) {
        memaccess((void *) evset[j+0]);
        memaccess((void *) evset[j+1]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[j+2]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[j+3]);
        memaccess((void *) evset[j+0]);
        memaccess((void *) evset[j+1]);
        memaccess((void *) evset[j+2]);
        memaccess((void *) evset[j+3]);
        memaccess((void *) evset[j+0]);
        memaccess((void *) evset[j+1]);
        memaccess((void *) evset[j+2]);
        memaccess((void *) evset[j+3]);}}
```

Listing 1. The preparation step in Prime+Scope on our Skylake processor.

There are two necessary conditions for building this high-resolution attack. First, the attacker needs to know the eviction candidate (l_s) of the target LLC set after priming. Second, l_s needs to be present in the private cache after priming, otherwise once the attacker accesses l_s, it is no longer the eviction candidate in the LLC. These two requirements make the attack very challenging because they are intuitively contradictory: being the eviction candidate means l_s is accessed *less frequently* than other cache lines; being present in the private cache means l_s is accessed *more frequently* than other cache lines. To satisfy the requirements, the original Prime+Scope uses very long and complicated access sequences to manipulate the replacement states of both the private cache and the LLC. The access sequence[5] for our Skylake processor is shown in Listing 1. It contains 192 cache references in total. This long access sequence results in a slow

[5] This pattern is not optimal. For example, it could be more efficient with knowing the details of the L1 replacement policy. Prime+Scope does not assume that knowledge for generality.


1) The attacker fills the set with d_t and l_0 to l_{w-2}.

2) The attacker waits; the next state of the set depends on whether the victim accesses d_t (a) or not (b).

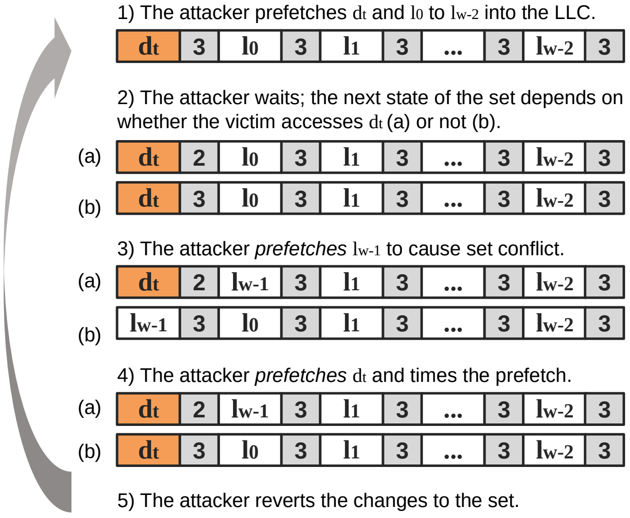
3) The attacker loads l_{w-1} to cause set conflict.

4) The attacker reloads d_t and times the load.

5) The attacker reverts the changes to the set.

dt 2 l0 2 l1 2 ... 2 lw-2 2
(a) dt 1 l0 2 l1 2 ... 2 lw-2 2
(b) dt 2 l0 2 l1 2 ... 2 lw-2 2
(a) dt 2 lw-1 2 l1 3 ... 3 lw-2 3
(b) lw-1 2 l0 3 l1 3 ... 3 lw-2 3
(a) dt 1 lw-1 2 l1 3 ... 3 lw-2 3
(b) lw-1 2 dt 2 l1 3 ... 3 lw-2 3

Figure 9. Sequence of the LLC set states during Reload+Refresh.


1) The attacker prefetches d_t and l_0 to l_{w-2} into the LLC.

2) The attacker waits; the next state of the set depends on whether the victim accesses d_t (a) or not (b).

3) The attacker *prefetches* l_{w-1} to cause set conflict.

4) The attacker *prefetches* d_t and times the prefetch.

5) The attacker reverts the changes to the set.

dt 3 l0 3 l1 3 ... 3 lw-2 3
(a) dt 2 l0 3 l1 3 ... 3 lw-2 3
(b) dt 3 l0 3 l1 3 ... 3 lw-2 3
(a) dt 2 lw-1 3 l1 3 ... 3 lw-2 3
(b) lw-1 3 l0 3 l1 3 ... 3 lw-2 3
(a) dt 2 lw-1 3 l1 3 ... 3 lw-2 3
(b) dt 3 l0 3 l1 3 ... 3 lw-2 3

Figure 10. Sequence of the LLC set states during Prefetch+Refresh.

preparation (priming) step. Thus, although Prime+Scope has high temporal resolution in each attack iteration, it requires a long preparation step between two consecutive iterations. Therefore, the attacker may miss the victim's accesses when the victim is repeatedly accessing her cache line with a high frequency, resulting in a high attack error rate.

2) Prime+Prefetch+Scope: The two key requirements in Prime+Scope can be satisfied in a much easier way when using PREFETCHNTA. As explained in Section IV-B, when prefetching a cache line, it is installed in the LLC set as the eviction candidate, and at the same time it is brought into the L1 cache. Thus, the preparation step in Prime+Scope can be done using the operations shown in Listing 2. We first prime the LLC set by accessing the eviction set (consisting of w cache lines without l_s) several times, so that the victim's data in this set gets old and can be reliably evicted. Then we prefetch l_s to install it into the L1 cache, as well as the LLC as the eviction candidate. Note that on tested processors, priming the eviction set twice is enough for reliably evicting the victim's data (with over 99.99% probability). Thus, on our Skylake processor, we only need 33 cache references (compared to 192 in the original Prime+Scope), resulting in a more efficient attack.

3) Faster Preparation Step: We test the total latency of the preparation step in each attack iteration. For the original Prime+Scope, to prepare the attack iteration, the attacker primes the target LLC set with a long pattern that takes a long period of time to finish. As shown in Figure 11, the preparation takes on average 1906 cycles on our Skylake processor (and 1762 on Kaby Lake). In contrast, with PREFETCHNTA, although the attacker needs to first prime the LLC set and then prefetch the scope line, the priming pattern is much shorter. The entire preparation step only takes 1043 cycles on the Skylake processor (and 1138 on Kaby Lake).

```
/* prime the eviction set n times */
for(i = 0; i < n; i++)
    for(j = 1; j <= 16; j++)
        memaccess((void *) evset[j]);

/* prefetch the scope line after priming */
prefetch_nta((void*) evset[0]);
```

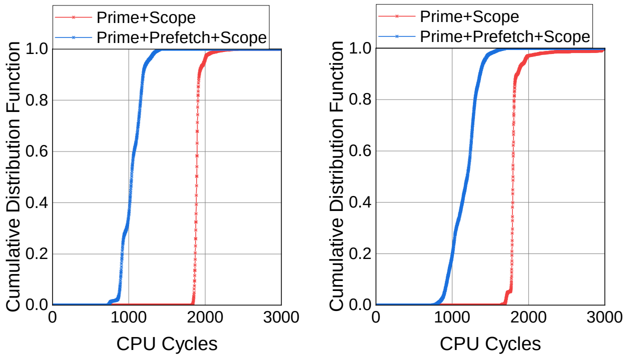
Listing 2. The preparation step in Prime+Prefetch+Scope on our Skylake processor.

The faster preparation step can make Prime+Prefetch+Scope more reliable and accurate than Prime+Scope. To prove this, we use three threads (T1 and T2) pinned on two different cores. T1 accesses a predetermined address every 1.5K cycles, as ground truth. T2 continuously monitors the LLC set for events using one of the attacks. We consider it a false negative if an event (from T1) is not detected. From the experiments on our Skylake processor, the false negative rate is about 50% for Prime+Scope. However, when using Prime+Prefetch+Scope, this rate is reduced to less than 2%.

B. Reload+Refresh with PREFETCHNTA

1) Reload+Refresh: Reload+Refresh [10] is one of the first attacks that leak the victim's information by monitoring *the replacement state changes* to the victim's cache line. Reload+Refresh is an LLC attack and it assumes shared data between the attacker and victim. To learn the victim's access pattern on the shared cache line (d_t), the attacker needs to prepare an eviction set (l_0 to l_{w-1}) that is mapped to same LLC set with d_t. Each attack iteration in Reload+Refresh consists of five steps, as shown in Figure 9. In Step 1, the attacker fills the target LLC set with d_t and l_0 to l_{w-2} in order. After this, all the cache lines in this set are in age 2.

Since d_t is the first line in the set, it is the eviction candidate. Then in Step 2, the attacker waits for the victim; if the victim accesses d_t, its age is updated to 1, and l_0 becomes the eviction candidate. Then in Step 3, the attacker forces replacement in this set by loading l_{w-1}. Either d_t or l_0 is evicted depending on whether the victim accessed d_t in Step 2. Then in Step 4, the attacker reloads d_t and times the load to learn whether it was evicted in the last step and infer whether the victim accessed it in Step 2. Finally in Step 5, the attacker reverts the changes in this set to prepare for the next attack iteration. The attacker first flushes d_t and l_{w-1} and then loads d_t and l_0 so that the states of d_t and l_0 are reset. After this, the attacker accesses l_1 to l_{w-2} in order, to refresh their ages from 3 back to 2.

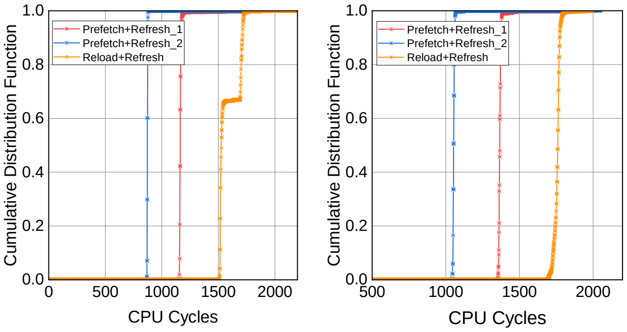


(a) The Skylake processor (b) The Kaby Lake processor

Figure 11. The total latency of the preparation step, for Prime+Scope and Prime+Prefetch+Scope.

Reload+Refresh is a powerful attack since it is much stealthier (on the victim's side) compared to prior LLC attacks such as Flush+Reload [68]. However, similar to Prime+Scope, many operations are needed in Reload+Refresh to reset the LLC state (in Step 5). In Flush+Reload, after measuring the victim's behavior (by reloading the shared data), the attacker only needs to flush this data to reset the state. In contrast, in Reload+Refresh the attacker needs to perform two flushes, two memory accesses, and $w-2$ serialized LLC accesses. Due to these operations, the state reset step and the entire attack iteration take very long to finish.

2) Prefetch+Refresh: We propose a new attack named Prefetch+Refresh which works similar to Reload+Refresh but with much less operations for resetting the state in each iteration. As shown in Figure 10, this attack also consists of five steps. In Step 1, the attacker prepares the target LLC set similar to the one in Reload+Refresh; however, the attacker initializes the age of each cache line to 3 instead of 2. Then in Step 2, the attacker waits for the victim; if the victim accesses d_t, its age is changed from 3 to 2. Later in Step 3, the attacker *prefetches* l_{w-1} (instead of loading it) to cause conflict in this set. Then in Step 4, the attacker *prefetches* d_t, as well as measures the prefetch latency to learn the victim's behavior in Step 2. Eventually in Step 5, the attacker reverts the changes to this LLC set. If we compare the state of this LLC set after Step 4 and the state in Step 1, only the two left most lines are potentially changed: if the victim accessed d_t, now its age is 2 instead of 3, and the second cache line from the left is l_{w-1} instead of l_0. Thus, the attacker does not need to access l_1 to l_{w-2} to change their ages, as done in Reload+Refresh. This results in a faster state reverting step in Prefetch+Refresh.



(a) The Skylake processor (b) The Kaby Lake processor

Figure 12. The total latency of the attacker's operations in each attack iteration, for Reload+Refresh and the two versions of Prefetch+Refresh.

We propose two options for the attacker to revert the states of these two cache lines in Step 5. First, the attacker can simply flush d_t and l_{w-1} and then reload d_t and l_0 to undo the state changes. In the second option, the attacker still flushes and reloads d_t. But she does not flush l_{w-1} and reload l_0, she instead uses l_0 to cause set conflict (in Step 3) in the next attack iteration, if the victim accessed d_t. In other words, the attacker exchanges the roles of l_0 and l_{w-1}. The second option makes Step 5 even faster compared to the first option; however, it slightly increases the complexity of the attack. The attacker needs to dynamically determine the cache line to use in Step 3 in each iteration. Table III shows the operations needed in Step 5 in Reload+Refresh and the two versions of Prefetch+Refresh.

Table III
# OF OPERATIONS FOR REVERTING THE CACHE STATE WITH A 16-WAY ASSOCIATIVE LLC.

Attack Method	# of flushes	# of DRAM accesses	# of LLC accesses
Reload+Refresh	2	2	14
Prefetch+Refresh v1	2	2	0
Prefetch+Refresh v2	1	1	0

3) Faster Attacks: We test the total latency of performing all the attacker operations in each iteration. For Reload+Refresh, the operations include loading l_{w-1} (to cause conflict), reloading d_t, flushing d_t and l_{w-1}, reloading d_t and l_0, and accessing l_1 to l_{w-2} (with pointer chasing). As shown in Figure 12, the average latency of a Reload+Refresh iteration (without the waiting window) is 1601 cycles on our Skylake processor (and 1767 cycles on the Kaby Lake processor). In contrast, when using Prefetch+Refresh (v1), the attacker does not need to access l_1 to l_{w-1}, and thus the average latency of an iteration is reduced to 1165 (and 1369) cycles. In Prefetch+Refresh (v2), flushing l_{w-1} and reloading l_0 are eliminated and the average latency is only 873 (and 1054) cycles.

VI. Discussion

A. *Fast Eviction Set Construction*

Conflict-based cache attacks such as Prime+Probe require the attacker to build eviction sets: given a target address, the attacker needs to find groups of addresses that are mapped into the same set with it (i.e., congruent with it) in the target cache such as the LLC. As mentioned in Section IV-B, the properties of `PREFETCHNTA` allow us to achieve one-way competition in an LLC set. As a result, with `PREFETCHNTA`, set conflicts occur more frequently than before when searching for congruent addresses. This leads to a more efficient algorithm for constructing eviction sets. Algorithm 2 shows our eviction set construction method. It repeatedly measures the prefetch latency of the target cache line l_t, and before each measurement, it prefetches a new candidate line l_c (which is potentially congruent with l_t). If the prefetched l_c is congruent with l_t, l_t is evicted and later it takes longer to prefetch it; then this l_c is added to the congruent address list. If the prefetched l_c is not congruent with l_t, it takes shorter to prefetch and l_t remains being the eviction candidate in the set after the prefetch; the algorithm then moves on to test the next candidate l_c. The algorithm keeps looking for congruent addresses until enough are found.

Algorithm 2: Eviction Set Construction

```
   Input: l_t, the target cache line for which an eviction set is desired
   Output: EV, the eviction set
1  EV ⟵ an empty list
2  ev_count ⟵ 0
3  while ev_count < ev_desired_size do
4      prefetch l_t
5      do
6          l_c ⟵ a candidate line
7          prefetch l_c
8      while prefetch l_t is fast;
9      EV[ev_count] ⟵ l_c
10     ev_count ++
11 end
```

The state-of-the-art eviction set construction method [42] uses a similar algorithm with ours. However, it accesses l_t and l_c in each searching iteration instead of prefetching them (in line 4, line 7, and line 8 of Algorithm 2). With this approach, a congruent cache line can only be observed (l_t can be evicted) if about w congruent lines have been tested/accessed since the last time l_t was brought into the LLC (in line 4), where w is the LLC associativity. This is because when accessing l_t (in line 4), it becomes the youngest cache line in the LLC set which will not be evicted until about w congruent cache lines are accessed (accessing EV between line 4 and line 5 can slightly reduce this number). When l_t is finally evicted, we only know that the last accessed l_c is congruent with it. In contrast, when using `PREFETCHNTA`, prefetching each congruent cache line can evict l_t since l_t is installed as the eviction candidate, making the algorithm much more efficient compared to the state-of-the-art. We test the execution time of these two approaches, and the results are shown in Figure 13.

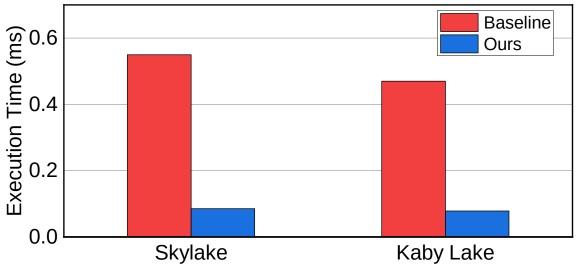


Figure 13. The execution time of the two algorithms.

B. `PREFETCHNTA` *with Non-Inclusive LLCs*

Most Intel server processors use non-inclusive LLCs. On such processors, `PREFETCHNTA` brings data only to the L1 cache and the coherence directory, but not the LLC [1]. Thus, the covert channel and side channel attacks discussed in this paper cannot directly work on those processors. However, if prefetched data are easier to be evicted from a set-associative coherence directory than loaded data, it may be possible for us to build fast set conflicts in the directory, resulting in a directory version of NTP+NTP. Verifying this vulnerability requires comprehensively understanding the replacement policy of the directory. Unfortunately, the directory policy has not yet been fully reverse engineered [42], [43], [66]. We leave it as future work.

Note that according to [3], on some AMD processors prefetched data are placed into a software-invisible buffer (instead of cache/directory). Therefore, it may be possible to build conflicts using PREFETCHNTA in this buffer and create a new covert channel.

C. *Related Work*

We have already introduced existing cache attacks in Section II-C. In this section we discuss prior prefetch-based side channel attacks.

Attacks based on software prefetch. Gruss et al. [17] found that on Intel processors, the execution time of a prefetch instruction, such as PREFETCHT0, leaks the translation levels of inaccessible kernel addresses. Using this observation, they built an attack to break Kernel Address Space Layout Randomization (KASLR). They also observed that prefetch instructions can bring inaccessible kernel data from DRAM to cache, but recent work [51] has proved this incorrect; their observation is actually the result of transient execution caused by a Spectre gadget in Linux kernel, not the prefetch instruction. Lipp et al. [29] later observed that on AMD processors, the timing (and power consumption) of a prefetch instruction on an inaccessible kernel address can leak the translation level and TLB state of this address. They used this to break KASLR as well as leak kernel memory data on AMD processors.

Very recently, Guo et al. [22] found that on Intel processors, PREFETCHW can be used to obtain the exclusive ownership even on read-only data. Based on this, they proposed a new cache eviction method and two new cross-core cache covert channels, Prefetch+Reload and Prefetch+Prefetch. However, both of their channels rely on the existence of shared data (between the attacker and victim). Our NTP+NTP channel does not have this requirement.

Attacks based on hardware prefetch. In 2018, Shin et al. [52] attacked OpenSSL, leaking the private key by leveraging the Intel stride prefetcher. Rohan et al. [46] later reverse-engineered the stream prefetcher on Intel processors, using it to build a covert channel.

D. *Countermeasures*

NTP+NTP uses a similar threat model with prior conflict-based cache covert channels such as Prime+Probe. Thus, countermeasures to mitigate conflict-based channels may also defend NTP+NTP. This includes 1) isolation-based defenses (e.g., [7], [15], [21], [31], [47]) which partition the cache so that data from different security domains do not interfere with each other, and 2) randomization-based defenses (e.g., [44], [48], [57], [63]) which make it very hard (if not impossible) to build set conflicts by modifying set index mapping.

In addition, a countermeasure for NTP+NTP specifically is to change the LLC insertion policy for both prefetched and loaded cache lines. For example, cache lines can be loaded into the LLC with age **1** and prefetched into the LLC with age **2**. Then, a prefetched cache line is still evicted sooner than a loaded cache line, but the prefetched cache line is no longer guaranteed to be the eviction candidate in the set. Thus, NTP+NTP can no longer work reliably. In addition, with this modified policy, the speed of our eviction set construction method (Algorithm 2) is significantly reduced. We build Python models of both the original Intel LLC policy and this modified policy, and simulate both our eviction set construction method and the state-of-the-art [42] with these two policies. With Intel LLC policy, our method requires **7.25**× less memory references compared to the state-of-the-art. In contrast, with the modified policy, this improvement is reduced to **1.26**×.

However, this countermeasure also weakens the performance benefit of PREFETCHNTA. With the original Intel LLC policy, prefetched cache lines can occupy at most one way in an LLC set, ensuring that the upper bound of LLC pollution is $1/w$, where w is the associativity. This is no longer guaranteed with the modified policy.

VII. Conclusion

In this paper, we first reverse-engineered the detailed cache behaviors of PREFETCHNTA, the non-temporal data prefetch instruction, on Intel processors. From the results, we found that using PREFETCHNTA, two cache lines that are mapped into the same LLC set can compete for the eviction candidate way in the set, achieving cache conflicts without priming the cache set for the first time. Based on this, we proposed NTP+NTP, a conflict-based cache covert channel which has much higher bandwidth compared to existing conflict-based channels such as Prime+Probe. In addition, we showed how PREFETCHNTA can be used in cache side channel attacks to improve their performance. Finally, we demonstrated a new LLC eviction set construction algorithm which is significantly faster than the state-of-the-art.

Acknowledgment

We thank the anonymous MICRO 2022 reviewers for their insightful feedback. We would also like to thank Andrew Zigerelli for his valuable comments. This work is supported in part by US National Science Foundation #1422331, #1535755, #1617071, #1718080, #1725657, #1910413, and #2011146.

References

[1] "Intel® 64 and IA-32 architectures optimization reference manual," available at https://cdrdv2.intel.com/v1/dl/getContent/671488.

[2] "Intel® 64 and IA-32 architectures software developer's manual," available at https://cdrdv2.intel.com/v1/dl/getContent/671200.

[3] "Software optimization guide for the AMD family 15h processors," available at https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf.

[4] O. Acıiçmez, "Yet another microarchitectural attack: Exploiting I-cache," in *ACM workshop on Computer security architecture*, 2007.

[5] O. Acıiçmez and W. Schindler, "A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL," in *Cryptographers' Track at the RSA Conference*, 2008.

[6] L. Armasu, "OpenBSD will disable Intel Hyper-Threading to avoid Spectre-like exploits (updated)," 2018, available at https://www.tomshardware.com/news/openbsd-disables-intel-hyper-threading-spectre,37332.html.

[7] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "CURE: A security architecture with CUstomizable and Resilient Enclaves," in *USENIX Security Symposium*, 2021.

[8] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison *et al.*, "Speculative interference attacks: Breaking invisible speculation schemes," in *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[9] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: Exploiting speculative execution through port contention," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[10] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security Symposium*, 2020.

[11] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, Gauss, and Reload-a cache attack on the BLISS lattice-based signature scheme," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[12] T. Claburn, "RIP Hyper-Threading? ChromeOS axes key Intel CPU feature over data-leak flaws – Microsoft, Apple suggest snub," 2019, available at https://www.theregister.co.uk/2019/05/14/intel_hyper_threading_mitigations/.

[13] Y. Cui and X. Cheng, "Abusing cache line dirty states to leak information in commercial processors," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

[14] S. Deng, W. Xiong, and J. Szefer, "A benchmark suite for evaluating caches' vulnerability to timing attacks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[15] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid side-channel-resilient caches for trusted execution environments," in *USENIX Security*, 2020.

[16] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX," in *USENIX Security Symposium*, 2017.

[17] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[18] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[19] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, 2015.

[20] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on AES to practice," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[21] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Ivcache: Defending cache side channel attacks via invisible accesses," in *Proceedings of the 2021 on Great Lakes Symposium on VLSI (GLSVLSI)*, 2021.

[22] ——, "Adversarial prefetch: New cross-core cache side channel attacks," in *2022 IEEE Symposium on Security and Privacy (S&P)*, 2022.

[23] R. Hegde, "Optimizing application performance on Intel® Core™ microarchitecture using hardware implemented prefetchers," 2008.

[24] T. Hornby, "Side-channel attacks on everyday applications: Distinguishing inputs with Flush+Reload," *BlackHat USA*, 2016.

[25] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[26] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[27] ——, "Cross processor cache attacks," in *ACM on Asia conference on computer and communications security (Asia CCS)*, 2016.

[28] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[29] M. Lipp, D. Gruss, and M. Schwarz, "AMD prefetch attacks through power and time," in *USENIX Security Symposium*, 2022.

[30] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[31] 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 *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[32] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[33] A. Malamy, R. N. Patel, and N. M. Hayes, "Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature," 1994.

[34] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, and V. Bertocci, "Security best practices for developing Windows Azure applications," *Microsoft Corp*, 2010.

[35] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the other side: SSH over robust cache covert channels in the cloud," in *NDSS*, 2017.

[36] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations," *International Journal of Parallel Programming*, 2019.

[37] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in JavaScript and their implications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

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

[39] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the Ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *USENIX Security Symposium*, 2021.

[40] C. Percival, "Cache missing for fun and profit," 2005.

[41] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "DRAMA: Exploiting DRAM addressing for cross-CPU attacks," in *USENIX Security Symposium*, 2016.

[42] A. Purnal, F. Turan, and I. Verbauwhede, "Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[43] ——, "Double Trouble: Combined heterogeneous attacks on non-inclusive cache hierarchies," in *USENIX Security Symposium*, 2022.

[44] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[45] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2009.

[46] A. Rohan, B. Panda, and P. Agarwal, "Reverse engineering the stream prefetcher for profit," in *IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[47] G. Saileshwar, S. Kariyappa, and M. Qureshi, "Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning," in *International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[48] G. Saileshwar and M. Qureshi, "MIRAGE: Mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Security Symposium*, 2021.

[49] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[50] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss, "NetSpectre: Read arbitrary memory over network," in *European Symposium on Research in Computer Security*, 2019.

[51] M. Schwarzl, T. Schuster, M. Schwarz, and D. Gruss, "Speculative dereferencing: Reviving foreshadow," in *International Conference on Financial Cryptography and Data Security*, 2021.

[52] 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 *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[53] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+Probe 1, JavaScript 0: Overcoming browser-based side-channel defenses," in *USENIX Security Symposium*, 2021.

[54] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security Symposium*, 2019.

[55] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling microarchitectural replay attacks," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[56] K. So and R. N. Rechtschaffen, "Cache operations by MRU change," *IEEE Transactions on Computers*, vol. 37, no. 6, 1988.

[57] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating cache conflicts with localized randomization." in *NDSS*, 2020.

[58] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[59] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking transient execution through microarchitectural load value injection," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[60] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE Symposium on Security and Privacy (S&P)*, May 2019.

[61] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking data on Intel CPUs via cache evictions," in *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[62] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[63] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "ScatterCache: Thwarting cache attacks via cache set randomization," in *USENIX Security Symposium*, 2019.

[64] Z. Wu, Z. Xu, and H. Wang, “Whispers in the Hyper-Space: High-speed covert channel attacks in the cloud,” in *USENIX Security Symposium*, 2012.

[65] W. Xiong and J. Szefer, “Leaking information through cache LRU states,” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[66] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[67] F. Yao, M. Doroslovacki, and G. Venkataramani, “Are coherence protocol states vulnerable to information leakage?” in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[68] Y. Yarom and K. Falkner, “Flush+Reload: A high resolution, low noise, L3 cache side-channel attack,” in *USENIX Security Symposium*, 2014.

[69] Y. Yarom, D. Genkin, and N. Heninger, “CacheBleed: A timing attack on OpenSSL constant-time RSA,” *Journal of Cryptographic Engineering*, 2017.

[70] Y. A. Younis, K. Kifayat, Q. Shi, and B. Askwith, “A new Prime and Probe cache side-channel attack for cloud computing,” in *IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015.

[71] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, “Cross-VM side channels and their use to extract private keys,” in *ACM SIGSAC conference on Computer and communications security (CCS)*, 2012.

[72] ——, “Cross-tenant side-channel attacks in PaaS clouds,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.