Evict+Spec+Time: Exploiting Out-of-Order Execution to Improve Cache-Timing Attacks

Shing Hing William Cheng[1], Chitchanok Chuengsatiansup[2], Daniel Genkin[3], Dallas McNeil[1], Toby Murray[2], Yuval Yarom[4] and Zhiyuan Zhang[2]

[1] University of Adelaide
[2] University of Melbourne
[3] Georgia Tech
[4] Ruhr University Bochum

Abstract. Speculative out-of-order execution is a strategy of masking execution latency by allowing younger instructions to execute before older instructions. While originally considered to be innocuous, speculative out-of-order execution was brought into the spotlight with the 2018 publication of the Spectre and Meltdown attacks. These attacks demonstrated that microarchitectural side channels can leak sensitive data accessed by speculatively executed instructions that are not part of the normal program execution. Since then, a significant effort has been vested in investigating how microarchitectural side channels can leak data from speculatively executed instructions and how to control this leakage. However, much less is known about how speculative out-of-order execution affects microarchitectural side-channel attacks.

In this paper, we investigate how speculative out-of-order execution affects the Evict+Time cache attack. Evict+Time is based on the observation that cache misses are slower than cache hits, hence by measuring the execution time of code, an attacker can determine if a cache miss occurred during the execution. We demonstrate that, due to limited resources for tracking out-of-order execution, under certain conditions an attacker can gain more fine-grained information and determine whether a cache miss occurred in *part* of the executed code.

Based on the observation, we design the Evict+Spec+Time attack, a variant of Evict+Time that can learn not only whether a cache miss occurred, but also in which part of the victim code it occurred. We demonstrate that Evict+Spec+Time is an order of magnitude more efficient than Evict+Time when attacking a T-table-based implementation of AES. We further show an Evict+Spec+Time attack on an S-box-based implementation of AES, recovering the key with as little as 14 389 decryptions. To the best of our knowledge, ours is the first successful Evict+Time attack on such a victim.

Keywords: Cache-timing attacks · out-of-order execution · AES

1 Introduction

Sharing computational resources poses a threat to information confidentiality. Microarchitectural side-channel attacks, which exploit contention on the internal components of the processor, have had an adverse impact on information confidentiality. Since their introduction in 2002 [TTMM02], a large number of attacks have been published, dismantling the security of symmetric cryptography [Ber05; GMWM16; GPS+20; OST06], public-key schemes [ABF+16; YF14], post-quantum cryptography [GBHLY16; GLG22; HSC+23], cryptographic protocols [RGG+19], and non-cryptographic software [GSM15;

OKSK15; SAO+21; YFT20; YPW22]. These attacks target most known components of the computer, including data caches [OST06; Per05; PTV21], instruction caches [ABG10; AS08], translation lookaside buffers [GRBG18; TTGB22; ZMFT22], branch prediction tables [EPA16; ZKE20; ZTO+23], prefetcher [GMF+16; VFP+22], and others [ABH+19; BSN+19; DHS22; MES18; PSHC21].

Traditionally, microarchitectural side-channel attacks focused on single components, treating activity of other components as noise [MIE17; YF14]. However, the recent disclosure of transient-execution attacks [KHF+19; LSG+18; VBMW+18] has demonstrated that interactions between microarchitectural components can have devastating impact on system security. In a nutshell, these attacks exploit the observation that an adversary can use microarchitectural side channels to leak secret information accessed during speculative and out-of-order execution, bypassing software and hardware security boundaries [BFM+22; KHF+19; KM21].

Significant effort has been dedicated to investigating microarchitectural side-channel attacks within the context of transient-execution attacks. Works in this area range from demonstrating the applicability of multiple channels [AGL19; BSN+19; CY22; TKK+22; ZBC+23] to proposing countermeasures that aim to block these channels [BBZ+19; KKS+19; LNM+21; ZBC+23]. However, much less attention has been invested in determining how out-of-order execution can affect microarchitectural side-channel attacks.

Early works in this area focused on documenting [YF14] and controlling [BP92; Gro00; TP08] the effects of out-of-order execution. Yet, recently several works have proposed exploiting speculation for overcoming countermeasures against microarchitectural side-channel attacks [BSP+21; ZBC+23; ZTO+23]. Moreover, several recent works show how speculative execution can be exploited for overcoming defenses based on limiting timer resolution [Kap23; KKC+23; PBPV23]. Finally, Rebeiro and Mukhopadhyay [RM15] investigate how the effects of pipelining and out-of-order execution can be used to mask cache-based timing leakage.

Our Contribution

In this work, we investigate another aspect of the complex interaction between out-of-order execution and microarchitectural side-channel attacks. Specifically, we demonstrate how to exploit out-of-order execution to improve the fidelity of Evict+Time [OST06], a cache-based side-channel attack technique.

In the Evict+Time attack, the attacker first evicts some memory from the cache and then measures the execution time of the victim code. If the victim code uses the evicted memory, access to that memory will be slow. Thus, observing an increase in the execution time of the code reveals that the victim has accessed the evicted memory. We present the Evict+Spec+Time attack, a variant of Evict+Time which exploits out-of-order execution to identify not only *whether* the access to the evicted memory occurs, but also *when* it occurs during the execution of the victim.

The main purpose of out-of-order execution is to hide the latency of some instructions by executing subsequent instructions before slow instructions complete. Our main observation is that this latency-hiding capability of out-of-order execution is restricted due to the limited resources available for tracking the execution. Consequently, controlling the resources available for out-of-order execution allows us to selectively hide the latency of accessing evicted memory and identify when, during the victim execution, the access occurs.

We demonstrate the effectiveness of Evict+Spec+Time in two attack scenarios. The first scenario targets the T-table implementation of AES, e.g. as provided in OpenSSL-3.0.9. We show that Evict+Spec+Time is an order of magnitude stronger than Evict+Time, allowing successful recovery of the four most significant bits of a key byte after observing only 250 decryptions, compared with the required 2 500 decryptions in the case of Evict+Time. This

improvement mirrors the results of Purnal et al. [PTV21], who show a similar efficiency with the Prime+Scope variant of the Prime+Probe attack.

Our second scenario targets S-box-based implementations of AES. Due to the small size of the S-box table and the number of accesses during the execution, such implementations are considered less vulnerable to side-channel attacks [ARVM20]. Nonetheless, past works have demonstrated that under strong attacker assumptions such attacks are possible. Specifically, the attacks assume the ability to interrupt the victim code frequently [ARVM20], the availability of special hardware features [BBM+21], or a combination of both [MIE17]. In contrast, our Evict+Spec+Time assumes no fine-grained control of victim execution and only uses generic features of out-of-order execution. However, it does rely on the context in which the victim's code executes. Thus, Evict+Spec+Time represents a new design point in the space of cache-based attacks against such implementations.

In summary, in this paper we make the following contributions.

- We investigate the interaction between out-of-order execution resources and its latency-hiding capabilities (Section 3).
- We design the Evict+Spec+Time attack, which exploits the limited latency-hiding capabilities to leak not only the fact that a cache miss occurs, but also when it occurs (Section 4).
- We show that Evict+Spec+Time allows an order of magnitude improvement in the efficiency of the T-table-based attack on AES compared to Evict+Time (Section 5).
- We present our Evict+Spec+Time attack, the first Evict+Time attack against an S-box implementation of AES (Section 6).

2 Background

This section briefly recalls fundamental concepts of out-of-order execution, advanced encryption standard, and cache-timing attack. We limit the explanation to details that are relevant to our work.

2.1 Out-of-Order Execution

To exploit instruction-level parallelism, processors do not necessarily execute instructions in the original program order. Instead, processors try to execute instructions as soon as all their inputs are ready and a suitable execution unit is available. To ensure that the results of the reordered instructions remain valid, the processors rely on a data structure called *reorder buffer* to keep track of the original program order and employ a variant of the Tomasulo algorithms [Tom67] to execute instructions in an arbitrary order. To track the dependencies between instructions, processors use a data structure called *reservation station*, which monitors the dependencies waiting for inputs to become available.

2.2 AES

The Advanced Encryption Standard (AES) is a symmetric block cipher that is based on a substitution-permutation network design. AES operates on a 128-bit block size, represented as a four-by-four state matrix in a column-major order (see Figure 1). During the encryption process, this matrix goes through multiple rounds of transformation where each round consists of the folowing four operations:

- SubBytes (*SB*) is a non-linear substitution where each byte of the state is replaced by another byte according to a predefined lookup table. Two common implementations of the lookup tables are S-box and T-table.[1]
- ShiftRows (*SR*) performs a circular rotate of row i to the left by i positions.

[1] T-table implementations combine SubBytes, ShiftRows, and MixColumns in a single lookup.

- MixColumns (MC) computes a linear function to combine the values along the column by multiplying the state matrix with a predefined four-by-four matrix.
- AddRoundKey (ARK) mixes round key k into the state matrix through an exclusive-or operation, denoted by $\oplus$.

We denote the inverse of these operations by InvSubBytes, InvShiftRows, InvMixColumns, and InvAddRoundKey respectively. Figure 2 shows the predefined matrix of InvMixColumns.

To decrypt, the inverse operations are performed in the reversed order of the encryption. However, many implementations of AES decryption, including both implementations we target, apply InvMixColumns to each of the round keys, allowing them to reorder the inverse operations so they follow the same order as in the encryption process. As we only focus on decryption, we use k^i to denote the round key used in the ith decryption round.

$$\begin{bmatrix} b_0 & b_4 & b_8 & b_{12} \\ b_1 & b_5 & b_9 & b_{13} \\ b_2 & b_6 & b_{10} & b_{14} \\ b_3 & b_7 & b_{11} & b_{15} \end{bmatrix}$$

Figure 1: Column-major order

$$\begin{bmatrix} 14 & 11 & 13 & 9 \\ 9 & 14 & 11 & 13 \\ 13 & 9 & 14 & 11 \\ 11 & 13 & 9 & 14 \end{bmatrix}$$

Figure 2: InvMixColumns Matrix

2.3 Cache-Timing Attacks

Cache. The cache is a small, fast bank of memory. It stores recently accessed data and exploits both the temporal and the spatial locality that a program exhibits to bridge the speed gap between the fast processor and the slow memory. Specifically, the memory space is divided into fixed-size *lines*, typically of size 64 bytes. When the processor accesses memory, it first checks if the requested memory line resides in the cache. In the case of a *cache hit*, when the line is found in the cache, the memory line can be served quickly. On the other hand, in case of a *cache miss*, when the requested memory line is not in the cache, the processor needs to bring it from the main memory, resulting in a longer retrieval time. In a cache miss, the processor typically stores the retrieved line in the cache for a potential future use. Since the cache is small and has a limited capacity, the processor may need to *evict* some lines out of the cache to make room for storing recently fetched memory lines.

Evict+Time. The Evict+Time attack [OST06] is a cache-based side-channel attack that observes timing behavior to determine cache states and infer victim's activity. The attack consists of two steps. In the first, *evict*, step, an attacker prepares the cache into a known state by evicting targeted memory addresses. In the second, *time*, step, the attacker measures the time it takes for the victim to execute an operation. A slow execution time indicates that the victim has accessed the evicted memory.

3 Out-of-Order Execution and Latency Hiding

The aim of out-of-order execution is to improve the performance and hide latency by exploiting parallelism. In this section, we investigate the interaction between out-of-order execution and cache-miss latency hiding. We first demonstrate that execution-time overlap can hide the latency of a cache miss. Then we explore the limitations of cache-miss latency hiding.

3.1 (In)distinguishable Cache Hit and Miss

The conventional wisdom is that a cache miss results in a much longer execution time compared to a cache hit. This is generally true if instructions are executed in-order, and subsequent instructions after a cache miss may not start until the cache miss is resolved.

However, in the case of out-of-order execution, processors try to reduce the latency by reordering the instructions while maintaining their dependency.

This observation implies that if (1) there are sufficient instructions that can be brought forward and executed while a cache miss occurs and (2) the total latency of those instructions is at least as large as the penalty of a cache miss, the overall execution time will be similar in both cases of cache hit and cache miss. In other words, cache misses may not necessarily lead to a longer execution time, which contrasts with the general belief.

In addition to the possibility of no distinction in timing between a cache hit and a cache miss, the length of cache-miss penalty, when it is observable, is not constant but depends on the level of possible execution overlap between the cache miss and subsequent instructions. This also implies that if a cache miss occurs early in the execution, there is a higher probability that the length of cache-miss penalty will be reduced. This is an important ingredient for our Evict+Spec+Time attack, which we introduce in Section 4.

Based on these observations, we design an experiment to demonstrate the impact of out-of-order execution on cache-miss latency hiding. In a nutshell, we perform two independent load operations, while controlling whether the data are cached, and show timing variations in different scenarios. Below we describe our experiment design and results.

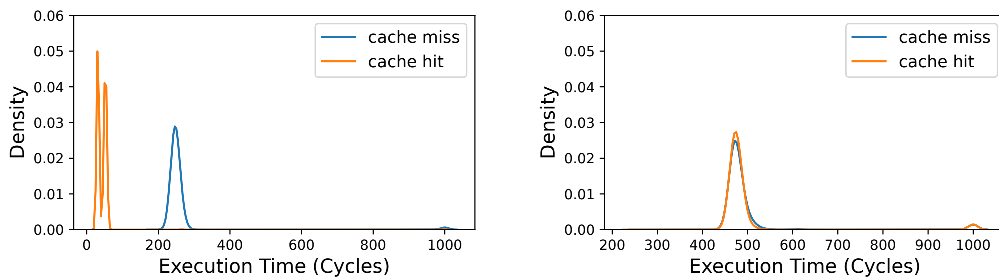
Experiment Design. To investigate the timing behavior due to the effect of out-of-order execution and cache states, we consider a program with two independent memory access operations as shown in Listing 1. Specifically, one operation follows a linked list (Line 1). This operation serves the purpose of controlling the execution-time overlap. The other operation is a pointer dereference (Line 2), whose latency we wish to measure in the cases that the memory it points to is cached or not.

Listing 1: Execution-time overlap between independent memory access

```
1  a = listhead->nextnode->nextnode;
2  b = *ptr;
```

Experiment Results. We conduct our experiments on an Intel Core i7-1165G7 processor, which supports out-of-order execution. To reduce the noise and effects of data prefetching [SKK+18; WQAK19], we randomize the memory locations in each experiment.

For each combination of cache states, we collect 100 000 samples. Figure 3 plots the distribution of execution time where the orange and blue lines indicate the execution time of when Line 2 in Listing 1 is a cache hit and a cache miss respectively.



(a) Linked list is cached: Distinguishable cache hit and cache miss

(b) Linked list is not cached: Indistinguishable cache hit and cache miss

Figure 3: Impact of out-of-order execution on cache-miss latency hiding

On the left-hand side, Figure 3a illustrates the situation where the linked-list memory locations are cached, i.e. this load operation can be done very quickly. In this situation, the execution-time overlap between traversing the linked list (Line 1) and pointer dereferencing

(Line 2) is very small. Consequently, out-of-order execution has a very limited opportunity to hide the cache-miss penalty. Therefore, we can clearly see from the figure that, as expected, a cache miss results in a longer execution time compared to a cache hit.

On the right-hand side, Figure 3b illustrates the situation where the memory locations that store the nodes of the linked list are not cached. In this situation, traversing the linked list takes a long time. Consequently, the execution-time overlap between traversing the linked list (Line 1) and dereferencing the pointer (Line 2) is large. Hence, out-of-order execution can hide the cache-miss penalty incurred in Line 2. Since Line 1 takes a long time to execute, Line 2 can finish its memory access within that time frame, In other words, the execution time of Line 2 fully overlaps with that of Line 1, and there is no additional cache-miss penalty caused by the memory access in Line 2. Therefore, and contrary to the naive expectations, we can no longer distinguish a cache hit from a cache miss.

3.2 Latency-Hiding Capability

We have seen that cache-miss latency can be hidden by execution-time overlap. This is because out-of-order execution allows reordering instructions and executing them according to the availability of the input data and execution units. While instructions can be executed out-of-order, the processor still needs to commit them in the order they appear in the program to ensure the correctness of the execution. For this reason, modern processors have integrated additional structures to enable such features.

We identify two structures that are relevant to out-of-order execution. The first is the reorder buffer, which keeps track of the original program order. The second is the reservation station, which keeps track of the operands of instructions and releases the instructions for execution when all operands are available. Since the sizes of the reorder buffer and reservation station are limited, once either is used up, out-of-order execution reaches its limit and younger instructions can no longer be reordered.

To analyze the threshold of latency hiding and out-of-order resource contention, we design experiments that exhaust the reorder buffer and the reservation station. In brief, we vary the number of instructions and create the dependency on input data in a program. We note that the similar technique has also been used to reverse engineer the size of backend buffers [TRVT22].

Experiment Design. Our new set of experiments follows a similar concept as the one in Section 3.1 where we study the (in)distinguishability in timing between cache hits and misses (see also Listing 1). That is, we consider a program with two independent memory access operations, which can be executed in parallel, and measure the overall execution time. However, in this experiment, we focus on the situation where the overall execution time is maximal, i.e. all linked-list memory locations are not cached. Hence, based on the experiments in Section 3.1, we expect that the latency of the pointer dereference will be masked.

Contrary to the prior experiments, in the current experiments, the program includes more operations between the two independent memory accesses. This enables the investigation of the latency-hiding capability in that long execution time.

Specifically, we use `nop` instruction to study the threshold of the reorder buffer. As illustrated in Listing 2, we insert `nop` instructions (Lines 3–5) between the two independent memory access instructions (linked list and pointer dereference). These `nop` instructions consume entries in the reorder buffer. We control the number of reorder buffer entries that the code uses by varying the number of `nop` instructions. Note that we could use different instructions other than `nop`. Our reason for choosing `nop` is that it consumes almost no computational resources but occupies the reorder buffer slot.

To study the threshold of the reservation station, we use `mov` and `cmp` instructions as shown in Listing 3. The purpose of the `mov` instruction (Line 2) is to create a dependency on the availability of the input data, which needs to wait for the linked list to complete

its traversal in order to obtain the loaded value. While the processor waits for the data, each of the `cmp` instructions (Lines 3–5) consumes a reservation station entry. Thus, the number of `cmp` instructions controls the number of reservation station entries used.

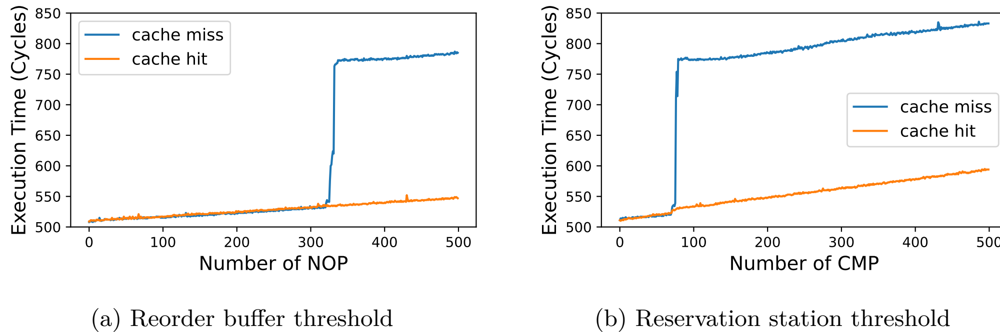
Listing 2: Exhausting reorder buffer

```
a = listhead->nextnode->nextnode;

NOP
...
NOP
b = *ptr

```

Listing 3: Exhausting reservation station

```
a = listhead->nextnode->nextnode;
mov r11, [a]
cmp r11, rdx
...
cmp r11, rdx
b = *ptr

```

Experiment Results. We perform our experiments on the same Intel Core i7-1165G7 processor, using the same setup as in Section 3.1. That is, we randomize the memory locations each time to reduce the noise. We repeat each measurement 100 000 times and report the average. The results are shown in Figure 4 where the orange and blue lines correspond to the execution time for a cache hit and a cache miss of the load instruction in Line 6 for both Listing 2 and Listing 3.



(a) Reorder buffer threshold

(b) Reservation station threshold

Figure 4: Impact of resources availability on cache-miss latency hiding

On the left-hand side, Figure 4a shows the results of exhausting the reorder buffer entries by varying the number of `nop` instructions. We observe that up to around 350 `nop` instructions, the execution time grows linearly with the number of `nop` instructions, and the cache hit and cache miss cases are indistinguishable. However, at around 350 `nop` instructions, we observe a sharp increase in the execution time for the case of a cache miss, whereas the execution time for cache hit continues the same linear growth. That is, we see that once we reach around 350 `nop` instructions, out-of-order execution no longer hides the cache-miss latency. This agrees with the explanation that the `nop` instructions consume all of the entries in the reorder buffer, preventing out-of-order execution of subsequent instructions. We do not have authoritative information on the size of the reorder buffer in our processor. However, the reported number of entries in the Sunny Cove microarchitecture, the predecessor of the Willow Cove that we use in our experiments, is 352.[2] This number agrees with our observation.

On the right-hand side, Figure 4b shows the results of exhausting the reservation station by varying the number of `cmp` instructions. Similarly, the execution time increases linearly as the number of `cmp` instructions increases. In this experiment, however, the increase in the execution time of the cache miss case occurs much earlier, at around 100 `cmp` instructions. At this point, the reservation station is fully exhausted, and out-of-order execution is at the limit of hiding the cache-miss penalty. Thus, we can clearly distinguish the cases of cache hit and cache miss.

[2] https://en.wikipedia.org/wiki/Sunny_Cove_(microarchitecture)

4 The Evict+Spec+Time Attack

Typical cache-timing attacks rely on the difference in the execution time between cache hit and cache miss to determine *whether* a victim has accessed a monitored cache line. In this section, we introduce a new attack called Evict+Spec+Time, which, in addition to determining *whether*, can also determine *when* the victim has accessed the monitored cache line.

4.1 Attack Overview

The Evict+Spec+Time attack combines eviction- and contention-based attacks. On the eviction-based side, Evict+Spec+Time follows a similar structure as the Evict+Time attack [OST06]. That is, it prepares the cache by evicting a certain line from it, and then infer the victim's behavior, i.e. *whether* the cache line has been accessed, by measuring the victim execution time.

On the contention-based side, Evict+Spec+Time considers the impact of speculation due to out-of-order execution. We create resource contention in, for example, the reorder buffer or the reservation station, and monitor the access to the targeted resource by measuring the latency of cache-miss penalty hiding to infer victim's behavior, i.e. *when*, during its execution, the victim accesses the cache line. Specifically, during the period of the victim's execution, the speculation naturally occurs. The variation in the execution time reveals how far the speculation proceeds, implying when the cache miss has been taken place.

The Evict+Spec+Time attack consists of two steps. In the first or the *evict* step, we evict the content from the monitored cache lines. This can be done by, for example, accessing a block of memory that maps to the targeted cache sets. In the second or the *spec+time* step, we let the victim execute then we measure the execution time, which is influenced by the speculation. Slow execution time indicates a cache miss while fast execution time indicates either no cache miss or a cache miss occurs early in the execution.

Thus, like Evict+Time, each instance of the Evict+Spec+Time attack is binary—it distinguishes between two conditions. However, there is a fundamental difference between the attacks. Whereas Evict+Time distinguishes whether the victim has accessed a cache line or not, Evict+Spec+Time identifies whether the victim has first accessed the cache line within a certain part of its execution. As we show in Section 6, this difference allows attacking implementations that were hitherto considered resilient to the Evict+Time attack. Moreover, with Evict+Spec+Time, the attacker can have a finer control over which part of the victim execution is targeted. Hence, while each instance of the attack only provides a binary result, multiple executions may provide more nuanced results.

4.2 Controlling Execution Timing Overlap

The heart of our attack is the Spec+Time part. Determining *when* a cache miss happens during the program execution requires an ability to manipulate the underlying structure that is relevant to latency hiding in out-of-order execution. Two approaches for controlling the execution-time overlap, namely, exhausting the reorder buffer and the reservation station, are discussed in Section 3.2.

The aim of controlling the execution-time overlap is to influence the latency of cache-miss penalty. If the execution-time overlap is short, the cache-miss penalty is large regardless of whether the cache miss happens sooner or later. In this situation, we can distinguish between a cache hit and a cache miss, hence detecting *whether* a cache miss occurs. On the other hand, if the execution-time overlap is long, a cache miss that occurs sooner will result in a smaller penalty overall, whereas a later cache miss will result in a larger penalty. Therefore, we can detect *when* the cache miss occurs.

4.3 Threat Model

As in most microarchitectural attacks, we assume that the attacker can execute the spy code on the same machine as the victim. We do not assume that the attacker has any special system privileges. In particular, the attacker cannot interleave its execution with that of the victim. However, as in other Evict+Time attacks, we assume that the attacker can execute victim code and measure its execution time.

Hyperthreading. For the attack, we assume that simultaneous multithreading (SMT) is either disabled or not used on the core that carries the attack out. In all of the experiments described in this work, SMT is enabled, but when running the attack code, we keep the sibling hyperthread of the same physical core idle. When disabling SMT, we achieve similar results.

Running code on a sibling thread is likely to affect the attack. When targeted resources are partitioned statically, as is the reorder buffer on Intel processors [TRVT22], the attack is likely to work, albeit it will affect fewer instructions. Conversely, when resources are shared dynamically, for example the reorder buffer on AMD processors [TRVT22], the behavior is likely to depend on the nature of the code running on the sibling thread and its resource usage. In particular, this opens the possibility of instantiating cross-hyperthread attacks, where the attack code runs on one hyperthread, and forces contention on the shared resource to affect the behavior of the victim on the sibling hyperthread. We leave experimenting with hyperthreading to future work.

Attack Gadget. Similar to Spectre-type attacks [ASBB+23; BSN+19; KHF+19; KKSA18], we assume that the victim code contains a gadget that allows an attacker to manipulate the execution-time overlap. The gadget we use in this work is a secret-independent memory access that enables the attacker to cause eviction and control resource contention.

5 Evict+Spec+Time on AES with T-Table

In this section, we demonstrate the efficiency of our Evict+Spec+Time attack through recovering a secret key of an AES implementation with T-table. Specifically, we perform a first-round attack targeting a decryption process. Since T-table-based implementations of AES are known to be vulnerable to cache-timing attack [AES15; PTV21; TANA07], we only perform a proof-of-concept attack to recover half of the key.

5.1 Attack Procedure

In T-table-based implementations of AES, an index to a lookup table can be calculated from a ciphertext and a secret key. Conversely, knowing the index together with the ciphertext allows us to derive the key. We apply our Evict+Spec+Time attack to determine the index of the T-table by monitoring which cache line is accessed during the decryption.

Recall that there are four T-tables, each having 256 entries. Since a single cache line holds 16 entries, a total of 16 cache lines are required to hold all the entries for one T-table. Identifying which cache line has been accessed provides us with $\log_2(16) = 4$ bits of information. In our experiment, we choose to monitor accesses to cache line 0 of the T-table during the first-round of the decryption. This provides us with the information that the four most significant bits (MSBs) of the key byte must be identical to those of the corresponding ciphertext byte. Note that while we use cache line 0 in our experiments, our attack only requires minor adaptations to work with the other cache lines. Specifically, when monitoring cache line c, the four MSBs of the key can be determined by XORing the four MSBs of the ciphertext with c.

Our attack begins with evicting cache line 0 of the targeted T-table. We also create a long execution-time overlap using a pointer dereferencing operation (Line 1 of Listing 4)

and control it by exhausting the reservation station. We note that the AES code itself appears to exhaust the reservation station (no additional instructions needed). Since the size of the reservation station is relatively small compared to the number of instructions in the AES code, the reservation station is mainly occupied by the first-round instructions. This is essential for cache-miss latency hiding which enables us to distinguish if a cache miss happens in the first round or later.

Listing 4: T-table based AES decryption API

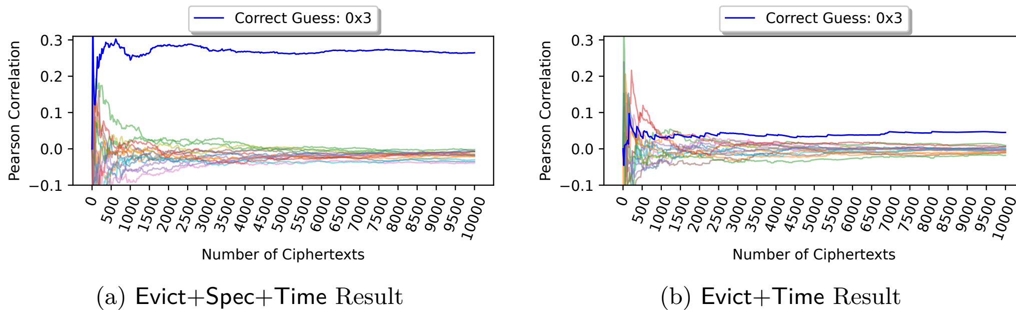
```
a = *ptr
AES_decrypt(pt, ct, &aeskey);
```

Once we have prepared the cache and set up a long execution time, we trigger the decryption process with random ciphertexts and measure the execution time. Recall that our aim is to determine if a cache miss occurs at cache line 0 in the first round of the decryption. Fast overall execution time implies that a cache miss either happens in the first round or does not happen at all. The probability that the monitored cache line has not been accessed is $(\frac{15}{16})^{40} \approx 7.6\%$. (We target AES-128 whose decryption consists of ten rounds, each having four accesses.) For simplicity, we assume that fast execution time means a cache miss, indeed, occurs in the first round. Even though this introduces some noise to our results, we show that our attack can still successfully recover half of the key with fewer ciphertexts than previous techniques.

To recover the key, we enumerate guesses for the four MSBs of each key byte for each of the random ciphertexts and calculate the Pearson correlation. We expect that the correct key guess will result in a high correlation score. Targeting a single T-table can recover the MSBs of the four key bytes that use the table in the first round. To recover the MSBs of all key bytes, we repeat the attack on all four T-tables.

5.2 Experiment Results

We perform our experiments on various processors. Specifically, we verify that our attack works on four Intel (Core i7-10710U, i7-1165G7, i9-11900K, i7-1255U) and three AMD (Ryzen 5600X, 5800X, 7950X) machines, all running Ubuntu 20.04 with default operating system configurations. As an example, we present the Pearson correlation of our experiment on i7-1165G7 in Figure 5a (for the first key byte). The figure clearly shows that the correlation of incorrect key guesses approaches zero as the number of ciphertexts increases. The correct key (0x3 in this case), however, has a significantly higher score. With approximately 250 ciphertexts, our Evict+Spec+Time can determine the correct key.



(a) Evict+Spec+Time Result (b) Evict+Time Result

Figure 5: Pearson correlation to recover 4 MSBs of an AES key byte

For comparison, we repeat our experiments but using Evict+Time attack [OST06]. The Pearson correlation of these experiments is shown in Figure 5b. The figure displays a similar trend where the Pearson correlation of incorrect key guesses approaches zero while

that of the correct one has a higher score. However, Evict+Time requires as many as 2 500 ciphertexts to identify the correct key, i.e. approximate ten times those of required by our Evict+Spec+Time.

While both attacks allow us to detect if a monitored cache line has been accessed by observing a cache miss, the Evict+Time attack only tells us if there is a cache miss in the *entire* decryption process. The probability of having a ciphertext that accesses a monitored cache line in any rounds is $(\frac{15}{16})^{39} \approx 8\%$. On the other hand, our Evict+Spec+Time attack can precisely focus on a cache miss at the first round. The probability of having a ciphertext that accesses a monitored cache line in the first round is $(1 - (\frac{15}{16})^{36}) \times (\frac{15}{16})^{3} \approx 74\%$. Since our Evict+Spec+Time attack has approximately ten time higher probability to obtain those ciphertexts, it does not come as a surprise that our Evict+Spec+Time attack requires approximately ten times fewer ciphertexts to identify the correct key.

6 Evict+Spec+Time on AES with S-Box

So far, we have shown that Evict+Spec+Time requires much fewer ciphertexts to perform a first-round attack on T-table-based AES implementations. We now show that Evict+Spec+Time is also efficient in breaking the S-box implementation of AES, which is considered to be less vulnerable to cache-timing attacks. Previous cache attacks on the S-box implementation of AES usually assume a strong attacker that has non-trivial control of the operating system [ARVM20; MIE17], allowing fine-grain interleaving of attack code within the victim execution. In contrast, our Evict+Spec+Time attack does not require such privileges. In this section, we show that Evict+Spec+Time allows enough fidelity for mounting an attack that does not interrupt the victim execution on this implementation.

We first describe a theoretical attack that shows how we perform the cryptanalysis under the assumption of a perfect oracle that precisely reveals which cache lines are accessed in each round. We then show how we use Evict+Spec+Time to realize such an oracle and explain how we deal with the noise that affects the realized oracle. Finally, we describe the results, demonstrating that our attack can recover the full key with 14 389 decryptions on average.

6.1 Theoretical Attack

Our theoretical attack assumes the availability of an oracle $\mathcal{O}_r^{cl}(c)$, which returns true if cache line cl of the S-box is not accessed during round r or any of the earlier rounds of the decryption of the ciphertext c. As our attack only queries the oracle about cache line 0, we simplify the notation by omitting cl.

Our attack consists of two main steps. In the first step, we recover the two most significant bits (MSBs) of each of the bytes of the first round key (k^0), which is used in the first decryption round. In the next step, we find a ciphertext that does not access the first cache line of the S-box (cache line 0) during the first two rounds of the decryption, and use this ciphertext to recover the remaining six least significant bits (LSBs) of each of the bytes of k^0. Below we describe these steps in further details.

6.1.1 First-Round Attack

Recall that given a ciphertext c, the AES decryption first performs the InvAddRoundKey operation, computing $c \oplus k^0$, before using the resulting bytes as indices to the S-box. (We ignore the InvShiftRows operation that just changes the bytes order but not their values.)

The S-box access of ith byte will hit cache line 0 of the S-box only if the two MSBs of $c[i]$, the ith byte of the ciphertext c, are the same as those of $k^0[i]$. Thus, the probability that decryption of a random ciphertext c will not hit cache line 0 when processing byte i

in the first round is $\frac{3}{4}$, and the probability that $\mathcal{O}_1(c)$ returns true is $(\frac{3}{4})^{16} \approx 1\%$. We use the term *witness* to refer to a ciphertext c for which $\mathcal{O}_1(c)$ returns true.

Note that for each $0 \le i < 16$, a witness c witnesses that the two MSBs of key byte $k^0[i]$ are not the same as the two MSBs of ciphertext $c[i]$. Hence, to recover the two MSBs of each key byte, we repeatedly select random ciphertexts and use the oracle $\mathcal{O}_1$ to search for a witness. We then use the witness to rule out a possible value of the two MSBs of each key byte. We repeat the process until we find enough witnesses to rule out three possible values for the two MSBs of each key byte. The remaining value is the correct two MSBs.

We note that once we find the first witness, we can use an adaptive technique to optimize the search for the key bits much more efficiently. Specifically, we can change the two MSBs of a single ciphertext byte until we get a ciphertext that is not a witness, revealing the two MSBs of the relevant key byte. While efficient, this technique does not lend itself well to a noisy oracle. A noisy oracle, might result in false positives, where the oracle incorrectly identifies a non-witness as a witness, necessitating the use of statistical approaches for removing noise. Since our realization of the oracle is noisy, we elected not to use this adaptive approach.

6.1.2 Second-Round Attack

For the second round attack, we first search for a ciphertext c for which both $\mathcal{O}_1(c)$ and $\mathcal{O}_2(c)$ return true. Considering that the first-round attack recovered the two MSBs of k^0, we can easily choose a random c for which $\mathcal{O}_1(c)$ returns true. By the same argument in Section 6.1.1, the probability that $\mathcal{O}_2(c)$ return true is approximately 1%. Hence, the expected number of ciphertexts we need to generate to find a suitable c is 100.

Once we find c, we target each byte of k^0 to recover the missing six LSBs. In the description below, we explain how we target the first byte of the key, i.e. $k^0[0]$. Adapting the description to other bytes is straightforward.

At a high level, for the attack we query $\mathcal{O}_2$ with the 64 ciphertexts that match all the bits of c, except for the six LSBs of the targeted byte $c[0]$. We then compare the oracle queries with the results of guessing the missing bits of the key byte to recover the full key. However, because each second-round S-box access depends on more than one key byte, we need to guess more key bits. We now describe the process.

Let c_j be the jth ciphertext for $0 \le j < 64$, i.e. $c_j[0] = c[0] \oplus j$ and $c_j[i] = c[i]$ for $i \ne 0$. (Note that $c_0 = c$.) Furthermore, let s_j be the state of the decryption of c_j just before the InvSubBytes step in the second decryption round. That is, $s_j = k^1 \oplus MC^{-1}(SR^{-1}(SB^{-1}(k^0 \oplus c_j)))$. We note that, due to the choice of c_j, for all $i > 3$ we have $s_j[i] = s_0[i]$. Consequently the S-box access for these bytes of the state miss cache line 0 of the S-box. Thus, $\mathcal{O}_2(c_j)$ is true if the two MSBs of $s_j[0]$, $s_j[7]$, $s_j[10]$, and $s_j[13]$ are non-zero, which happens with a probability of $(\frac{3}{4})^4 \approx 32\%$.

We now need to guess enough key bits to allow us to match the oracle queries to determine the correct key. Following the steps of the decryption, we can compute $s_j[0]$ as:

$$\begin{aligned} s_j[0] \quad = \quad & 14 \cdot SB^{-1}(c_j[0] \oplus k^0[0]) \oplus 11 \cdot SB^{-1}(c_j[13] \oplus k^0[13]) \oplus \\ & 13 \cdot SB^{-1}(c_j[10] \oplus k^0[10]) \oplus 9 \cdot SB^{-1}(c_j[7] \oplus k^0[7]) \oplus k^1[0] \end{aligned}$$

Repeating the calculation for $s_j[1]$, $s_j[2]$, and $s_j[3]$ reveals that their values depend on the eight key bytes, namely, four from k^0 and four from k^1. The first-round attack has already recovered two of the bits of each of the first round key bytes. Hence, with a naive approach of guessing the missing key bits, we need to guess a total of $4 \times 6 = 24$ first round key bits. Moreover, the cache line accessed also depends on the two MSBs of each of the second round bytes involved. Hence, a naive approach will require guessing a total of 32 key bits.

We can reduce the search space significantly by observing that for s'_0 defined as

$$\begin{aligned} s'_0 &= 11 \cdot SB^{-1}(c_0[13] \oplus k^0[13]) \oplus 13 \cdot SB^{-1}(c_0[10] \oplus k^0[10]) \oplus \\ & 9 \cdot SB^{-1}(c_0[7] \oplus k^0[7]) \oplus k^1[0] \end{aligned}$$

we have $s_j[0] = s'_0 \oplus 14 \cdot SB^{-1}(c_j[0] \oplus k^0[0])$. Similarly, we can find s'_1, s'_2, and s'_3 such that for $0 \leq i < 4$ we have $s_j[i] = s'_i \oplus C_i \cdot SB^{-1}(c_j[0] \oplus k^0[0])$, where C_i is the corresponding entry in the InvMixColumns matrix (see Figure 2).

Based on this calculation, we can determine whether decrypting c_j accesses cache line 0 in the second round of the decryption by guessing a total of 14 bits, i.e. two MSBs of each of s'_0, s'_1, s'_2, and s'_3, as well as the six LSBs of $k^0[0]$. Testing in practice shows that given the 64 oracle queries, we can correctly recover the key byte.

6.2 Realizing the Oracles

In Section 6.1 we assume oracles $\mathcal{O}_r$ that return `true` if decrypting a ciphertext does not access a specific cache line of the S-box during round r or any of the earlier rounds. In this section, we focus on using the Evict+Spec+Time technique to realize such oracles.

Recall that in both the Evict+Time and our Evict+Spec+Time attacks, the attacker evicts a cache line that the victim may use and then times the execution of the victim code to determine whether the monitored cache line has been accessed. However, unlike Evict+Time, our Evict+Spec+Time also exploits the interaction between execution-time overlap and cache-miss latency hiding to determine if the monitored cache line has been accessed earlier in the victim execution.

To use Evict+Spec+Time for realizing the oracles we need for our attack on S-box AES, we exploit the limited number of entries in the reorder buffer. A typical scenario that realizes the $\mathcal{O}_1$ oracle is depicted in Figure 6. In this scenario, an instruction that causes delay, e.g. a cache miss, appears several hundreds of instructions before the decryption code. While this instruction executes, the processor fills the reorder buffer with younger instructions from the program code. Due to the limited space in the reorder buffer, only the code of the first round of the decryption fits in the reorder buffer and gets executed out-of-order. Eventually, the delay-causing instruction completes its execution, at which time it is retired, allowing the processor to proceed and execute instructions beyond the first round of the decryption. Thus, this scenario masks the latency of the execution of the first round of the decryption, but not of any subsequent rounds. A similar scenario can be used to realize $\mathcal{O}_2$.

As a proof-of-concept, we use the code in Listing 5. The code consists of two memory accesses (Lines 1 and 3) and an invocation of the decryption code (Line 5). Sequences of 100 `nop` instructions separate the two memory accesses, as well as between the accesses and the decryption code. By evicting a memory location accessed in Lines 1 or 3, the attacker can cause delays that fill the reorder buffer. On our machine, evicting the earlier location (Line 1) allows only the first round of encryption to fit in the reorder buffer during the delay, thus realizing $\mathcal{O}_1$. Conversely, evicting the location accessed in Line 3 leaves more space in the reorder buffer, allowing the code of the second round to execute concurrently with the delay-causing instruction, realizing $\mathcal{O}_2$.

Listing 5: S-box based AES decryption API

```
1 a = *ptr
2 NOP x 100
3 a = *a
4 NOP x 100
5 AES_decrypt(pt, ct, &aeskey);
```

It should be noted that the salient feature of the code in Listing 5 is having the two delay-causing instructions separated by some other instructions. That is, the separating

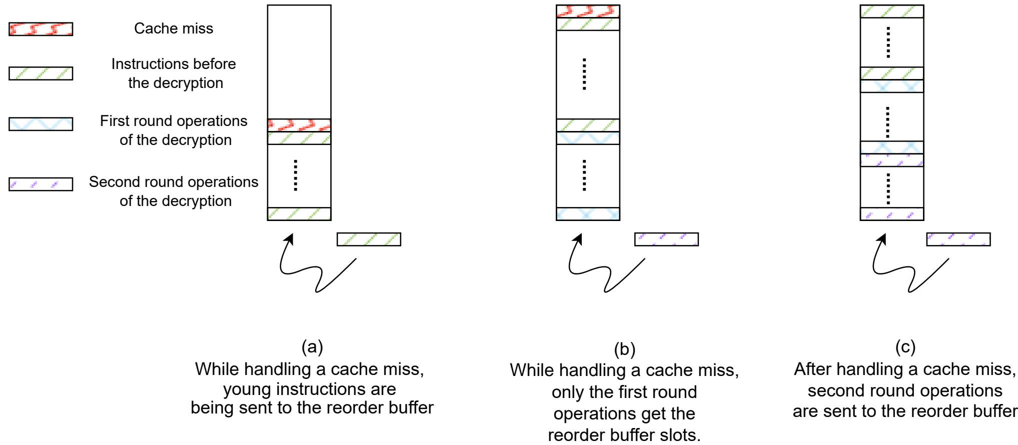


Figure 6: State of the reorder buffer while using Evict+Spec+Time to realize $\mathcal{O}_1$.

instructions need not be nops, and we could use other instructions. Moreover, the exact number of separating instructions is not too critical. Adding or removing some instructions may create measurement noise due to inclusion of some instructions from subsequent rounds or omission of some instructions from the targeted round. Thus, we believe that while the exact implementation of our proof-of-concept code is unlikely to be found in real code, vulnerable code is likely to exist in practice. We leave the task of searching for such code to future work.

6.3 Recovering the Two MSBs of Key Bytes

We proceed to adapting the theoretical attack (Section 6.1) to use the realized oracles (Section 6.2) and evaluate the results. In this section, we investigate the first-round attack. In the next section, we demonstrate the second-round attack. Experiments in this section use the same machines and configurations as Section 5.2, namely, four Intel (Core i7-10710U, i7-1165G7, i9-11900K, i7-1255U) and three AMD (Ryzen 5600X, 5800X, 7950X) machines, all running Ubuntu 20.04 with default operating system configurations.

Recall that the theoretical attack repeatedly generates random ciphertexts and uses the oracle $\mathcal{O}_1$ to identify witnesses. It then uses the witnesses to reject impossible values of the two MSBs of each key bytes until only one possibility remains for each key byte.

The main problem with carrying this approach over to the concrete oracle, which we realize through Evict+Spec+Time, is that the measurements are noisy. Specifically, while witnesses do incur a longer measurement time, some non-witness ciphertexts may also execute longer. This is evident in Figure 7, where we show the execution time of 10 000 random ciphertexts, collected on the i7-1165G7 machine. The figure groups the ciphertexts by the MSBs of the first byte and uses green triangles to mark real witnesses. We note that we first remove 332 outliers that execute for longer than 1 200 cycles. These are not shown in the figure and are ignored in the analysis.

As Figure 7 shows, the vast majority of ciphertexts executing the realized oracle take between 720 and 790 cycles. However, there is a non-negligible number of ciphertexts that take longer. Many of those longer measurements are witnesses, where the speculative execution does not overlap with the access to the evicted cache line. There is also a large number of non-witnesses that take longer due to measurement noise.

Excluding outliers, the total number of longer measurements is relatively small (584 out of the total of 10 000 ciphertexts), and the non-witness measurements are concentrated in a very narrow band. We now need to distinguish the witnesses from the non-witnesses.

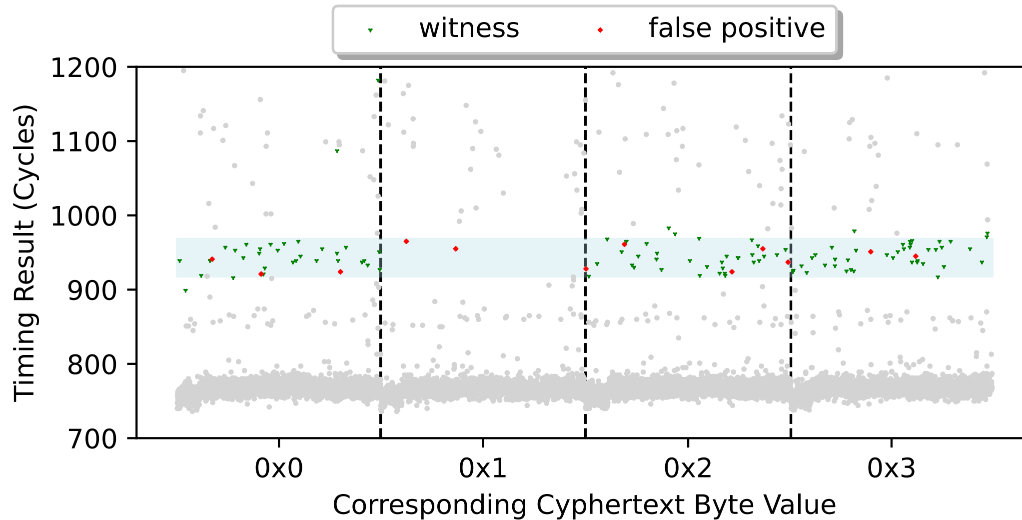


Figure 7: Execution time of the realized oracle with 10 000 random ciphertexts

We observe that the timing difference between a witness and a non-witness is caused by the latency of a single cache miss that is not masked in the case of witnesses. Experimentally, we determine that on the i7-1165G7 machine this timing difference is approximately 175 cycles. We further find that by focusing on the band between 150 and 200 cycles above the sample median, we can capture most of the witnesses. Figure 7 highlights this band in light blue.

Recall that the two MSBs of the target byte in a witness are never the same as the two MSBs of the corresponding key bytes. This is clearly evident in Figure 7, where no witness has MSBs 0x1. However, some noisy measurements that fall in the witness band do have the MSBs 0x1. Because we cannot distinguish true witnesses from noisy samples that fall in the witness band, we cannot use any single ciphertext to rule out a key value. However, within the witness band, noisy measurements are expected to be distributed uniformly across MSBs, whereas witnesses do not have the MSBs of the key. Thus, if we group the ciphertexts that are in the witness band by the MSBs of the target byte, we can expect that the number of samples in the group of the key MSBs will be the smallest.

To determine the number of ciphertexts required, we generate 10 000 random keys and measure the success rate of recovering the two MSBs of all key bytes for different ciphertext counts. As Figure 8 shows, the success rate increases with the increase of ciphertexts and it varies between machines. We explicitly present the success rate with 5 000 and 10 000 ciphertexts in Table 1.

Table 1: Success rate (in percentage) with 5 000 and 10 000 ciphertexts

	i7-10710U	i7-1165G7	i9-11900K	i7-1255U	5600X	5800X	7950X
5 000	52.65	87.92	87.08	19.58	92.50	98.01	67.03
10 000	91.64	98.57	95.08	65.75	98.65	99.95	97.36

6.4 Recovering the Full Key Bytes

We proceed to adapting the theoretical second-round attack of Section 6.1 to use our Evict+Spec+Time-based $\mathcal{O}_2$ oracle.

Recall that in the theoretical attack, we first find a ciphertext c for which $\mathcal{O}_2$ returns

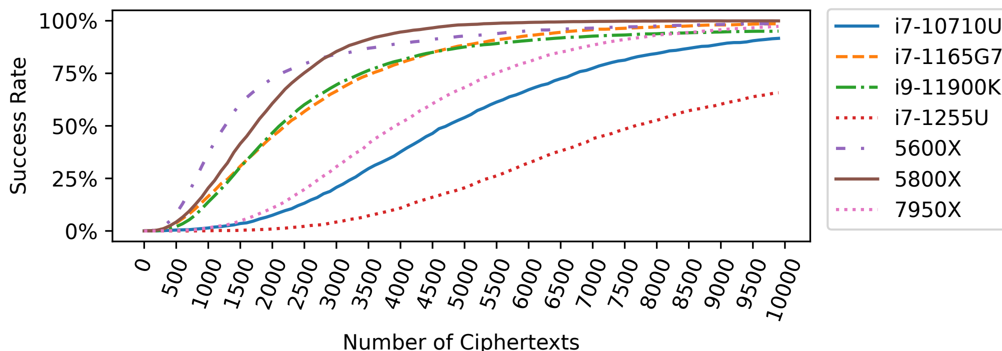


Figure 8: Success rate in determining the two MSBs of key bytes.

true, and then use this c to find the key. Unfortunately, due to measurement noise, determining whether a ciphertext satisfies $\mathcal{O}_2$ is not easy. Instead, the approach we follow is to select a random c for which $\mathcal{O}_1(c)$ returns true. (We can do that because in Section 6.3 we recovered the two MSBs of each key byte.) We then assume that this ciphertext satisfies $\mathcal{O}_2$ and proceed to recover the first key byte. If we manage to recover the first key byte with a high level of confidence, we proceed to recover the rest of the key bytes. Otherwise, we repeat the process with a new random ciphertext. We now describe these steps in further details.

To realize the $\mathcal{O}_2$ oracle, we use the code in Listing 5, but this time the attacker evicts the location accessed in Line 3 in addition to cache line 0 of the S-box. Evicting the location accessed in Line 3 forces a long execution of the memory access, allowing only instructions that fit in the reorder buffer to execute concurrently. Due to the design of the code in Listing 5, this allows only the first two rounds of decryption to proceed. Consequently, if the decryption accesses cache line 0 in the first two rounds, the latency of the cache miss will be masked and execution will be fast. Conversely, if the decryption does not access cache line 0 in the first two rounds, i.e. when the ciphertext satisfies $\mathcal{O}_2$, the cache miss will not be masked, and execution will be longer.

To overcome the effects of noise, we combine the oracle query with the recovery of the first key byte. That is, given a ciphertext c, which satisfies $\mathcal{O}_1$ but may or may not satisfy $\mathcal{O}_2$, we generate 64 c_j, as described in Section 6.1, and measure the decryption time with the code in Listing 5. If c satisfies $\mathcal{O}_2$, we expect about 32% of the c_js to satisfy $\mathcal{O}_2$.

We now guess the six LSBs of the key byte $k^0[0]$, and the two MSBs of each of $k^1[0]$, $k^1[1]$, $k^1[2]$, and $k^1[3]$. If we assume that $\mathcal{O}_2(c)$ returns true, we can use the guess to determine $\mathcal{O}_2(c_j)$. We then expect a high correlation between the determined values of $\mathcal{O}_2(c_j)$ and the measurements of the decryption times.

Figure 9 shows the Pearson correlation between the measurement times and the determined $\mathcal{O}_2(c_j)$ for all possible key guesses (collected on i7-1165G7). (Key guesses grouped by the value of the six LSBs of $k^0[0]$.) The figure highlights the samples for which the guess of $k^0[0]$ matches the ground truth. It further shows that the correct guess has a very high correlation (0.87). Overall, we find that the Pearson correlation is typically above 0.7 if $\mathcal{O}_2(c)$ is true and the guess is correct. Conversely, if either $\mathcal{O}_2(c)$ is false or the guess is incorrect, the Pearson correlation tends to be below 0.7. Thus, if the Pearson correlation for all 2^{14} guesses is below 0.7, we can conclude with a high likelihood that $\mathcal{O}_2(c)$ is false.

It turns out that there exists a scenario where $\mathcal{O}_2(c)$ is false, yet there is a key guess that shows a high correlation with the measurement. Specifically, this scenario occurs if $\mathcal{O}_2(c_j)$ is true for one or more j. Consequently, when finding a high correlation, we cannot conclude that $\mathcal{O}_2(c)$ is true, or that the correlating key guess is correct.

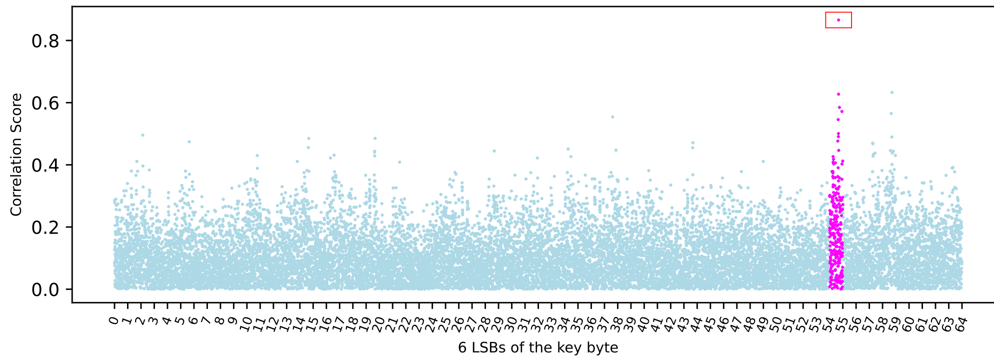


Figure 9: Pearson correlation to determine the value of $\mathcal{O}_2(c_j)$

To handle this case, once we find a key guess for byte 0 that shows a high correlation, we attempt to recover byte 1 of the key. If for byte 1 we also find a key guess that has a high correlation (>0.7) with the observed timing, we conclude that $\mathcal{O}_2(c)$ is `true`, and proceed to recover the remaining 14 bytes. If we do not find a good key guess for byte 1, we replace the first byte of c, and repeat the process. We find that by following this process, we eventually arrive at a ciphertext c that satisfies $\mathcal{O}_2$, and can recover the key.

To test the attack, we generate 10 000 random keys. For each, we use the ground truth to simulate the result of a successful first-round attack and perform a second-round attack, counting the number of oracle queries performed until we recover the key. Figure 10 shows the cumulative distribution function (CDF) of the number of oracle queries. As the figure shows, with approximately 5 600 and 6 500 queries we achieve a success probability of above 80% for i7-1165G7 and i9-11900K respectively. For the i7-1255U, we only achieve a success probability above 50%, at 6 300 queries, with little improvement beyond that point. For other CPU models, we find that the attack does not work, presumably because the reorder buffer is not big enough to store the instructions of the second round.

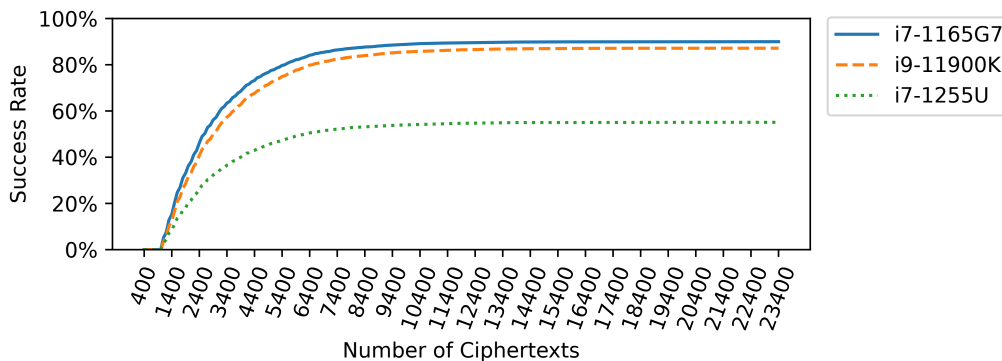


Figure 10: Success rate of the full-key recovery in relation to the number of ciphertexts.

6.5 Complexity of the Full Attack

With the results in Figures 8 and 10, we can now compute an upper bound of the expected number of oracle queries required for recovering the full key. Specifically, we assume that the attacker performs a first-round attack with a fixed number of queries N_1, finding the

correct key bits with a probability of S_1. The attacker then proceeds to the second round, performing at most N_2 queries, yielding a probability S_2 of finding the correct key if the first round was successful. If the attack fails, the attacker repeats the process.

Under this attack model, the attacker performs at most $N_1 + N_2$ queries, achieving a probability of $S_1 \cdot S_2$ of recovering the key. Hence, the expected number of queries the attacker needs to perform for finding the key is $\frac{(N1_+N_2)}{(S_1 \cdot S_2)}$.

We experimentally determine the values of S_1 and S_2 for different values of N_1 and N_2 on our processors. We then use those to find the best combination for each processor, i.e. the combination that minimizes the expected number of queries. We present the expected number of queries for different processors in Table 2.

Table 2: Expected number of queries for different processors

Processor	N_1	N_2	S_1	S_2	#queries
i7-1165G7	4 600	4 500	0.85	0.74	14 389
i9–11900K	3 900	4 100	0.80	0.66	15 343
i7-1255U	9 900	5 900	0.65	0.49	49 122

7 Conclusions

In this work we present the Evict+Spec+Time attack, a variant of Evict+Time that can learn not only that a cache miss occurred while executing victim code, but also where in the victim code the miss occurred. We show that when attacking the T-table implementation of AES, our Evict+Spec+Time provides an order of magnitude improvement in attack efficiency over Evict+Time. We further demonstrate that our Evict+Spec+Time can recover the key from the S-box implementation of AES, an implementation that was hitherto thought to be resilient to the Evict+Time attack.

Evict+Spec+Time combines properties of two very different processor optimizations: caches and speculative out-of-order execution. It demonstrates that interactions between microarchitectural components can augment attacks. We believe that research into such interactions is still in its infancy and further work is required to fully understand the potential implications of such interactions.

Our work demonstrates once again the folly of relying on perceived limitations of contemporary attacks when assessing the security of implementations. Like all cache-based attacks, Evict+Spec+Time leaks addresses. Hence, constant-time programming provides a solid defense against the attack. Considering that most modern processors support instructions for secure execution of AES, and that efficient constant-time implementations of AES exist for processors that do not have dedicated instructions, we see no justification for the use of vulnerable implementations.

Acknowledgements

This work has been supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the ARC Discovery Early Career Researcher Award DE200101577; the ARC Discovery Project number DP210102670; the Defense Advanced Research Projects Agency (DARPA) under contract W912CG-23-C-0022; the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972. the National Science Foundation under grant CNS-1954712; and gifts from Qualcomm and Cisco.

Parts of this work were undertaken while Yuval Yarom was affiliated with the University of Adelaide.

References

[ABF+16] Thomas Allan, Billy Bob Brumley, Katrina E. Falkner, Joop van de Pol, and Yuval Yarom. Amplifying side channels through performance degradation. In *ACSAC*, 2016.

[ABG10] Onur Aciiçmez, Billy Bob Brumley, and Philipp Grabher. New results on instruction cache attacks. In *CHES*, 2010.

[ABH+19] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE SP*, 2019.

[AES15] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$A: a shared cache attack that works across cores and defies VM sandboxing - and its application to AES. In *IEEE SP*, 2015.

[AGL19] Ben Amos, Niv Gilboa, and Arbel Levy. Spectre without shared memory. In *SAC*, 2019.

[ARVM20] C Ashokkumar, Bholanath Roy, M Bhargav Sri Venkatesh, and Bernard L. Menezes. "S-box" implementation of AES is not side channel resistant. *HASS*, 4, 2020.

[AS08] Onur Aciiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *CT-RSA*, 2008.

[ASBB+23] Basavesh Ammanaghatta Shivakumar, Jack Barnes, Gilles Barthe, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Sioli O'Connell, Peter Schwabe, Rui Qi Sim, and Yuval Yarom. Spectre declassified: reading from the right place at the wrong time. In *IEEE SP*, 2023.

[BBM+21] Samira Briongos, Ida Bruhns, Pedro Malagón, Thomas Eisenbarth, and José Manuel Moya. Aim, wait, shoot: how the cachesniper technique improves unprivileged cache attacks. In *EuroS&P*, pages 683–700, 2021.

[BBZ+19] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. SpecShield: shielding speculative data from microarchitectural covert channels. In *PACT*, 2019.

[Ber05] Daniel J Bernstein. Cache-timing attacks on AES. https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[BFM+22] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: on the effectiveness of hardware mitigations against cross-privilege Spectre-v2 attacks. In *USENIX Security*, 2022.

[BP92] Michael Butler and Yale N. Patt. An investigation of the performance of various dynamic scheduling techniques. In *MICRO*, 1992.

[BSN+19] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: exploiting speculative execution through port contention. In *CCS*, 2019.

[BSP+21] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos V. Rozas, Adam Morrison, Frank McKeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa R. Alameldeen. Speculative interference attacks: breaking invisible speculation schemes. In *ASPLOS*, 2021.

[CY22] Md Hafizul Islam Chowdhuryy and Fan Yao. Leaking secrets through modern branch predictors in the speculative world. *IEEE TC*, 71(9), 2022.

[DHS22] Shuwen Deng, Bowen Huang, and Jakub Szefer. Leaky frontends: security vulnerabilities in processor frontends. In *HPCA*, 2022.

[EPA16] Dmitry Evtyushkin, Dmitry V. Ponomarev, and Nael B. Abu-Ghazaleh. Jump over ASLR: attacking branch predictors to bypass ASLR. In *MICRO*, 2016.

[GBHLY16] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and reload - a cache attack on the BLISS lattice-based signature scheme. In *CHES*, 2016.

[GLG22] Guillaume Goy, Antoine Loiseau, and Philippe Gaborit. A new key recovery side-channel attack on HQC with chosen ciphertext. In *PQCrypto*, 2022.

[GMF+16] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: bypassing SMAP and kernel ASLR. In *CCS*, 2016.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *DIMVA*, 2016.

[GPS+20] Daniel Genkin, Romain Poussier, Rui Qi Sim, Yuval Yarom, and Yuanjing Zhao. Cache vs. key-dependency: side channeling an implementation of Pilsung. *CHES*, 2020.

[GRBG18] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: defeating cache side-channel protections with TLB attacks. In *USENIX Security*, 2018.

[Gro00] J. P. Grossman. Cheap out-of-order execution using delayed issue. In *ICCD*, 2000.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: automating attacks on inclusive last-level caches. In *USENIX Security*, 2015.

[HSC+23] Senyang Huang, Rui Qi Sim, Chitchanok Chuengsatiansup, Qian Guo, and Thomas Johansson. Cache-timing attack against HQC. *CHES*, (3), 2023.

[Kap23] David A. Kaplan. Optimization and amplification of cache side channel signals. *CoRR*, abs/2303.00122, 2023.

[KHF+19] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: exploiting speculative execution. In *IEEE SP*, 2019.

[KKC+23] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: improving cache attacks with transient execution. In *USENIX Security*, 2023.

[KKSA18] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre returns! speculation attacks using the return stack buffer. In *USENIX Security*, 2018.

[KKS+19] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael B. Abu-Ghazaleh. Safe-Spec: banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*, 2019.

[KM21] Ofek Kirzner and Adam Morrison. An analysis of speculative type confusion vulnerabilities in the wild. In *USENIX Security*, 2021.

[LNM+21] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: securing speculation with the principle of transient non-observability. In *USENIX Security*, 2021.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: reading kernel memory from user space. In *USENIX Security*, 2018.

[MES18] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: a false dependency attack against constant-time crypto implementations in SGX. In *CT-RSA*, 2018.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: how SGX amplifies the power of cache attacks. In *CHES*, 2017.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: practical cache attacks in JavaScript and their implications. In *CCS*, 2015.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *CT-RSA*, 2006.

[PBPV23] Antoon Purnal, Marton Bognar, Frank Piessens, and Ingrid Verbauwhede. ShowTime: amplifying arbitrary CPU timing side channels. In *AsiaCCS*, 2023.

[Per05] Colin Percival. Cache missing for fun and profit. In *BSDCon 2005*, Ottawa, CA, 2005.

[PSHC21] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: leaking control-flow in SGX via the CPU frontend. In *USENIX Security*, 2021.

[PTV21] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: overcoming the observer effect for high-precision cache contention attacks. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS*, 2021.

[RGG+19] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache attacks on TLS implementations. In *IEEE SP*, 2019.

[RM15] Chester Rebeiro and Debdeep Mukhopadhyay. Micro-architectural analysis of time-driven cache attacks: quest for the ideal implementation. *IEEE TC*, 64(3), 2015.

[SAO+21] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+Probe 1, JavaScript 0: overcoming browser-based side-channel defenses. In *USENIX Security*, 2021.

[SKK+18] Young-joo Shin, Hyung Chan Kim, Dokeun Kwon, Ji-Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *CCS*, 2018.

[TANA07] Kris Tiri, Onur Aciiçmez, Michael Neve, and Flemming Andersen. An analytical model for time-driven cache attacks. In *FSE*, 2007.

[TKK+22] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: combining Spectre and Rowhammer for new speculative attacks. In *IEEE SP*, 2022.

[Tom67] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1), 1967.

[TP08] Francis Tseng and Yale N. Patt. Achieving out-of-order performance with almost in-order complexity. In *ISCA*, 2008.

[TRVT22] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean M. Tullsen. Secsmt: securing SMT processors against contention-based covert channels. In *USENIX Security*, pages 3165–3182, 2022.

[TTGB22] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *USENIX Security*, 2022.

[TTMM02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. Cryptanalysis of block ciphers implemented on computers with cache. In *ISITA*, 2002.

[VBMW+18] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 2018.

[VFP+22] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. Augury: using data memory-dependent prefetchers to leak data at rest. In *IEEE SP*, 2022.

[WQAK19] Daimeng Wang, Zhiyun Qian, Nael B. Abu-Ghazaleh, and Srikanth V. Krishnamurthy. PAPP: prefetcher-aware prime and probe side-channel attack. In *DAC*, 2019.

[YF14] Yuval Yarom and Katrina Falkner. Flush+Reload: a high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 2014.

[YFT20] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. Cache telepathy: leveraging shared resource attacks to learn DNN architectures. In *USENIX Security*, 2020.

[YPW22] Yuanyuan Yuan, Qi Pang, and Shuai Wang. Automated side channel analysis of media software with manifold learning. In *USENIX Security*, 2022.

[ZBC+23] Zhiyuan Zhang, Gilles Barthe, Chitchanok Chuengsatiansup, Peter Schwabe, and Yuval Yarom. Ultimate SLH: taking speculative load hardening to the next level. In *USENIX Security*, 2023.

[ZKE20] Tao Zhang, Kenneth Koltermann, and Dmitry Evtyushkin. Exploring branch predictors for constructing transient execution Trojans. In *ASPLOS*, 2020.

[ZMFT22] Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, and Josep Torrellas. Binoculars: contention-based side-channel attacks exploiting the page walker. In *USENIX Security*, 2022.

[ZTO+23] Zhiyuan Zhang, Mingtian Tao, Sioli O'Connell, Chitchanok Chuengsatiansup, Daniel Genkin, and Yuval Yarom and. BunnyHop: exploiting the instruction prefetcher. In *USENIX Security*, 2023.