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

Yun Chen*        Ali Hajiabadi*        Trevor E. Carlson

National University of Singapore

* Equal contribution first authors
**GADGETSPINNER: Overview**

• Introducing a new transient execution primitive: **Loop Stream Detector**

• Proposing **GADGETSPINNER**; an attack methodology based on the LSD
  
  • Bypassing secure Branch Prediction Unit (BPU) designs
  
  • Providing more **practical cross-core transient execution attacks**
Frontend in Intel x86 CPUs

1. Translation Path (Slow)
   - Instructions fetched from memory and decoded
2. Micro-op Cache Path (Fast)
   - Instructions hit in the micro-op cache
3. Loop Path (Very Fast)
   - Instructions part of a loop body and repeat
LSD: Operation and Transient Window

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

1. **1st iteration (delivered by Decode)**
   - `leak(array[0])`
   - `i = 0`

2. **2nd iteration (delivered by LSD)**
   - `leak(array[1])`
   - `i = 1`

3. **3rd iteration (delivered by LSD)**
   - `leak(array[2])`
   - `i = 2`
   - **Branch Misprediction**

**Transient Window**
Spectre-v1 Leaks via PHT Speculation Primitive

**Attacker** (Core-0)

```c
void mistrain() {
    for (j=0; j<1000; j++) {
        if (true) //do sth
    }
}
```

**Victim** (Core-0)

```c
int array[2];
void victim (int index) {
    if (index < 2) {
        leak(array[index]);
    }
}
```

- **Key Requirement:** Sharing the PHT (i.e., co-location on the same core)

**Examples of BPU-based mitigations:**

- **BPU Partitioning**
  - e.g., Half&Half [S&P’23]

- **BPU Flushing/Randomization**
  - e.g., HyBP [HPCA’22]

- **BPU Encryption**
  - e.g., STBPU [DSN’22]
GADGETSPINNER Leaks via LSD Speculation Primitive

**Attacker** (Core-0)

```
1 //no mistraining
2 victim (123456);
```

**Victim** (Core-1)

```
1 int array[2];
2 void victim (int offset) {
3     for (i = 0; i < 2; i++) {
4         index = ((2 ^ i) - 1) & offset;
5         leak(array[index]);
6     }
7 }
```

BPU-based mitigations are not effective because BPU is disabled during LSD operations

- **BPU Partitioning**
  - e.g., Half&Half [S&P’23]

- **BPU Flushing/Randomization**
  - e.g., HyBP [HPCA’22]

- **BPU Encryption**
  - e.g., STBPU [DSN’22]
Loop Stream Detector Out-of-Loop-Bounds Access

```c
uint8_t A[8];
void victim (uint64_t offset){
    uint64_t index = 0;
    for (int i = 0; i < 8; i++) {
        temp = A[index];
        index = ((8 ^ i) - 1) & offset;
    }
}
```

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

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

(1) Expected behavior (LSD disabled)

(2) Illegal out-of-loop-bounds access (LSD enabled)
How to trigger LSD speculation primitive?

1. Loop body smaller than 64 micro-ops (size of the LSD)

2. All micro-ops in loop body align with micro-op cache lines

3. Absence of hard-to-predict branches in loop body

LSD Qualified Loops

```
int value = 0, B[1] = {2};
for (i = 0; i < 2; i++){
  index = ((2 ^ i) - 1) & offset;
  leak(array[index]);
  if (value < B[0]) x+=2; else x+=1;
}
flush(&B);
```

(a) Aligned loop body

(b) Misaligned loop body

Disables LSD
GADGETSPINNER: Threat Model and Setup

Threat Model

- Gadget existence in the victim (i.e., LSD qualified loops)
- Co-location with the victim not needed (enabling cross-core attacks)
- Triggering the victim with LSD qualified loops
- A transmission/extraction channel (e.g., cache primitives)

<table>
<thead>
<tr>
<th>Specification</th>
<th>System 1</th>
<th>System 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cloud Provider</td>
<td>AWS EC2</td>
<td>Microsoft Azure</td>
</tr>
<tr>
<td>Processor</td>
<td>Xeon Platinum 8375C</td>
<td>Xeon Platinum 8370C</td>
</tr>
<tr>
<td>Architecture</td>
<td>Ice Lake (Sunny Cove)</td>
<td>Cascade Lake</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 20.04</td>
<td></td>
</tr>
<tr>
<td>SGX</td>
<td>Not supported</td>
<td>SDK:2.19.100.3</td>
</tr>
</tbody>
</table>
**GADGETSPINNER: Attack Methodology and PoCs**

1. **Channel Preparation**
2. **LSD Qualified Loop**
3. **Illegal Behavior/Leakage**
4. **Leakage Transmission**
5. **Channel Extraction**

**Our Proof-of-Concept Attacks:**

1. **Illegal Read Attack**
2. **Cross-Core Attack**
3. **Extracting CNN weights**
4. **Breaking KASLR**

**Steps:**

1. **Illegal Read Attack**
2. **Normal / Allowed Transfers**
3. **LSD Transient Execution (out-of-loop-bounds)**
4. **Breaking KASLR**
PoC #1: Illegal Read from Protected Memory

```c
uint8_t A [8 * CACHE_LINE] = {0};
uint8_t B [8 * CACHE_LINE] = {42};
uint8_t *page_B = B & 0xfffffffffffff000;
mprotect(page_B, PAGE_SIZE, PORT_WRITE | PORT_EXEC);
```

**Flush+Reload**

**Cache hit**

(Confidential value is 42)
PoC #2: Cross-Core Illegal Arbitrary Read

Cross-core and cross-process arbitrary reads are feasible

1. **Challenge:** How to determine the virtual mapping of the victim?
   - Take a guess for victim’s virtual address
   - Running the victim with the guess
   - No mapping found?
     - No
     - Yes
       - Arbitrary read GADGETSPINNER attack

2. Only requires LSD, GADGETSPINNER qualified loops
3. No need for branch Mistraining (i.e., no need to share BPU)

16GiB search space in Linux

~30 minutes runtime with SoTA search algorithms

1. N. Vella, “Breaking 64 bit ASLR on linux x86-64,” https://github.com/nick0ve/how-to-bypass-aslr-on-linux-x86_64
**PoC #3: Extracting CNN Weights in SGX DNNL**

### Untrusted Zone (Attacker)

```
// carefully crafting data and params
int cpu_cnn_train_f32 (void *data, void *params) {
  // training cnn
  init_net_data(data, dim, params); // clean unused data
}
```

### Trusted Zone (Victim)

```
// training cnn
int cpu_cnn_train_f32 (void *data, void *params) {
  // attacker-chosen inputs
}
```
PoC #4: Breaking Kernel ASLR (KASLR)

- **KASLR**: randomizes kernel memory placement at each system boot

512 possible kernel placements\(^1\)

```c
uint8_t probe[2 * 4096];

int loop_function (int offset) {
    uint64_t idx = 0;
    for (int i = 0; i < 8; i++) {
        uint8_t p = *(uint8_t*)(&array[0] + idx);
        uint8_t value = probe[p + (idx / offset) * 4096];
        idx = ((ARRAY_SIZE ^ i) - 1) & offset;
    }
}
```

Mitigating GADGETSPINNER for Existing Hardware

• We implemented three LLVM compiler passes and tested SPEC2017:

1. Naive-Fence-Protection
   Inserting a fence before all loop branches
   +103% overhead

2. Fence-Protection
   Inserting a fence before qualified loop branches
   +72% overhead

3. NOP-Protection
   Inserting NOPs to qualified loop bodies to disable the LSD
   +19% overhead
## Mitigating GADGETSPINNER for Future Hardware

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
<th>Protection against Spectre-v1</th>
<th>Protection against GADGETSPINNER</th>
</tr>
</thead>
</table>
| 1    | Secure BPU Design and Usage  
Partitioning, randomizing, flushing the BPU  
Examples: HyBP [HPCA’22], Half&Half [S&P’23] | ✓ | X |
| 2    | Securing Specific Channels  
E.g., protecting caches against Spectre  
Examples: InvisiSpec [MICRO’18], CleanupSpec [MICRO’19] | ✓ | ✓ |
| 3    | Securing All Potential Channels  
E.g., restricting execution of speculative instructions  
Examples: STT [MICRO’19], DOLMA [USENIX Sec’21] | ✓ | ✓ |
Conclusions

• Loop Stream Detector (LSD) creates a transient window, unrelated to BPU decisions

• We propose GADGETSPINNER, a new transient attack primitive exploiting the LSD transient window:
  • It bypasses Secure BPU protections
  • It makes cross-core transient execution attacks more practical

• We demonstrate threats of GADGETSPINNER via four different PoC attacks

• We investigate different compiler mitigations for GADGETSPINNER in existing hardware
Thanks for your Attention
Questions?

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

Yun Chen*  Ali Hajiabadi*  Trevor E. Carlson

National University of Singapore

* Equal Contribution