# Overtake: Achieving Meltdown-type Attacks with One Instruction

Yu Jin[†], Pengfei Qiu[†§*], Chunlu Wang[†], Yihao Yang[†]
Dongsheng Wang[‡§], Xiaoyong Li[†], Qian Wang[¶], Gang Qu[‖]

[†]Key Laboratory of Trustworthy Distributed Computing and Service (BUPT), Ministry of Education, Beijing, China
[‡]Tsinghua University, Beijing, China [§]Zhongguancun Laboratory, Beijing, China
[¶]Lehigh university, PA, USA [‖]University of Maryland, College Park, MD, USA
lambda.jinyu@gmail.com, {qpf, wangcl}@bupt.edu.cn, khaosyg@gmail.com
wds@tsinghua.edu.cn, lixiaoyong@bupt.edu.cn, wangqian2010bupt@gmail.com, gangqu@umd.edu

***Abstract*—In early 2018, the Meltdown attack was reported, which steals secret data by loading and then encoding them into the cache covert channel during the invisible transient executions. After that, a set of Meltdown-type attacks are proposed; those attacks largely threaten the security of modern processors.**

**In this study, we review Intel's x86-64 Instruction Set Architecture (ISA) and find two vulnerable instructions (`CMPSB` and `SCASB`) that can be exploited to achieve the Meltdown-type attacks with few instructions. Especially, the `CMPSB` instruction itself is enough to implement the core part of the Meltdown-type attacks. We design a special cache-based and Performance Monitor Unit (PMU)-based covert channel to recover the secret data for the two instructions. In our experiments, we demonstrate the availability of the two instructions by implementing the Meltdown and ZombieLoad attacks with them. Compared to the original Meltdown-type attacks, the proposed attack can be considered as an attack that does not rely on the transient executions from the perspective of the macro instruction level because no more macro instruction is executed after triggering the exception. Therefore, we name our attacks Overtake. Our experiments indicate that the average data leakage speed of Overtake attack could reach 770.1 KB/s with an error rate of 0.4 %.**

***Index Terms*—Meltdown, processor security, transient execution, instruction set architecture, secret data leakage**

## I. Introduction

Modern processors integrate advanced technologies to optimize the instruction pipelines for high performance, including out-of-order execution, speculative execution, and Microarchitectural Data Sampling (MDS), etc. These technologies enable processors to execute subsequent instructions before previous instructions have retired. In situations where the processor determines that these later instructions should not have been executed, it will revert the execution to maintain correctness. Such rollbacks occur when exceptions are triggered during out-of-order execution [1], mispredictions happen in speculative execution [2], or when microcode-assisted operations are needed in MDS [2]. This process of executing "invisible" instructions is known as transient execution.

Lipp et al. [1] disclose a processor vulnerability related to transient execution driven by out-of-order execution in 2018. This vulnerability can cause data-related cache side-effects, and the processor does not roll back such effects after the transient execution. They proposed the Meltdown attack to exploit this vulnerability, giving attackers the opportunity to gain unauthorized access to the secret data through transient execution and bring this "invisible" data to the architecture level. Since then, several Meltdown-type attacks have been proposed and implemented, such as Foreshadow, Fallout, RIDL, and ZombieLoad [3]–[6]. Unfortunately, complete repairs at the software level have proven difficult [7], and these transient execution attacks pose serious threats to mainstream processors.

The process of transient execution attacks [8] encompasses primarily four key steps: (1) crafting a microarchitecture ($\mu$arch) state to establish a cache covert channel; (2) utilizing data-access instructions to load confidential information; (3) generating cache side-effects by accessing a specific cache unit correlated with the secret data; and (4) extracting the secret data by analyzing the cache side-effects. Steps (1) and (4) serve as conventional techniques to establish the cache covert channel, enabling the transfer of secret data from the microarchitecture ($\mu$arch) to the architecture (arch) domain. Steps (2) and (3) are the key steps of transient execution attacks. In step (2), an exception is provoked due to the unauthorized access of secret data by the attacker. Notably, step (3) is transiently executed to encode the secret data into cache side-effects even if step (2) is not fully executed (retired). These steps are designed based on the fact that current transient execution attacks need to use multiple macro instructions to achieve steps (2) and (3) respectively.

In this study, our hypothesis revolves around the potential efficacy of combining step (2) and step (3) into a singular instruction, which could potentially circumvent the widely advocated software-based mitigations and offer novel insights into microarchitecture ($\mu$arch) security. Motivated by this hypothesis, we meticulously analyze Intel's ISA [9] to identify instructions that exhibit $\mu$arch effects once secret data has been retrieved from memory. We refer to these instructions as potentially vulnerable instructions. Through investigation and empirical experiments, we unearth two such vulnerable

*Corresponding author

instructions: `CMPSB` and `SCASB`. Given that these instructions determine how secret data is encoded, the development of specialized covert channel mechanisms becomes imperative. To this end, we construct both a cache covert channel and a Performance Monitoring Unit (PMU) covert channel to leak secrets using those instructions. We successfully utilized both of the two vulnerable instructions to implement Meltdown-type attacks across multiple Intel processors, including Meltdown and ZombieLoad. We named this novel attack Overtake because it accomplishes complex attack objects through a few instructions. Additionally, these instructions trigger more micro-operations ($\mu$OPs) to be emitted in comparison to other instructions at the decode front end of the processor.

Through experiments, we demonstrated that the $\mu$OPs generated by a single macro instruction are sufficient to execute the transient phase of the Meltdown attack. Once the exception is triggered at the transient level, no additional execution of macro instructions is necessary. When comparing the byte-level bandwidth performance with paper Overtake, we achieved a noteworthy bandwidth ranging from 34.1 KB/s to 770.1 KB/s in different scenarios shown on the Intel i7-7700 processor. Our experimental results also demonstrated that any processors that are vulnerable to Meltdown-type attacks are also susceptible to our Overtake attack. Moreover, our investigation confirms the utility of both cache timing-based and PMU-based covert channels in recovering secret data within the context of the Overtake attack.

In summary, our work makes the following contributions:

- We identify two susceptible instructions (`CMPSB` and `SCASB`) in the x86-64 ISA that can be leveraged for Meltdown-type attacks.
- Given the complex operations of these two instructions, effectively recovering the secret data becomes a significant challenge.
- We have effectively deployed the Overtake attack to execute conventional Meltdown-type attacks on four distinct Intel processors, encompassing both Meltdown and ZombieLoad vulnerabilities.

## II. Background

### A. Instruction Set Architecture and Microarchitecture

Instruction Set Architecture (ISA) defines the CPU's abstract model and how it is controlled by macro instructions. Complex instruction sets such as x86-64 allow single instructions to execute several low-level operations in order to simplify development, improve performance, reduce program size, etc. The term $\mu$arch refers to how a processor implements its ISA, including the decoding frontend, execution backend, and memory subsystem. The design of a CPU's $\mu$arch plays a significant role in determining the efficiency and performance of the processor [10]. Resulting in variations in performance, power consumption, and other aspects, such as security. Most ISAs provide a mechanism known as PMU to monitor the processor's performance. Unfortunately, PMU has also been exploited as a covert channel for transient execution attacks [11].

### B. Cache Side-Channel

Modern processors utilize multi-level caches to improve performance [10]. The cache line can be associated with different memory addresses in the main memory, marked by the Translation Lookaside Buffer (TLB). Since caches are shared resources for multi-process and multi-physical cores, they pose security risks as attackers can exploit them to infer secret information. Various cache side-channel attacks have been proposed in the past decades, including Flush+Reload [12], Prime+Probe [13], etc. These attacks can be utilized as high-performance covert channels.

### C. Transient Execution Vulnerabilities

Transient execution is considered an architectural "invisible" execution because it would not retired or committed by the processor. In certain processor design implementations, the transient execution can handle unexpected data before being granted authorization. This architectural behavior has been exploited by attacks like Meltdown and Spectre, which have garnered significant attention after being disclosed. Moreover, the out-of-order execution and speculative execution optimized mechanisms employed in modern processors may trigger transient execution, potentially leading to vulnerabilities. Attackers can exploit these vulnerabilities by utilizing cache covert channels or other covert channels, potentially resulting in the leakage of sensitive $\mu$arch data.

## III. Overview of Overtake

In this section, we present Overtake, which achieves the Meltdown-type attack with only one or two macro instructions.

### A. Inspiration

As shown in Fig. 1, during the front-end instruction decode step for the x86-64 architecture (arch), a single instruction could be decoded into several $\mu$OPs [14]. These $\mu$OPs can then be executed out-of-order in the back end if there is no data dependency and the execution unit is idle. The existence of $\mu$OPs enables a singular macro instruction to execute complex operations, facilitating the potential for out-of-order execution.

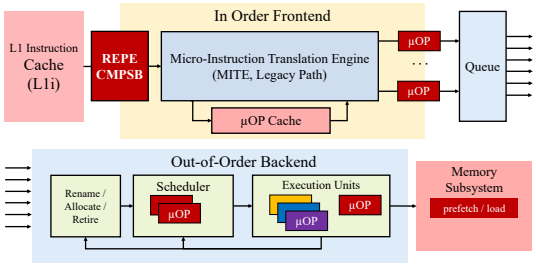


Fig. 1: In Intel's x86-64 chip, the `REPE CMPSB` instruction is decoded into several $\mu$OPs. These $\mu$OPs are then executed out-of-order in the backend if their requirements are met.

### B. Assumption and Threat Model

The threat model for Overtake attack is similar to Meltdown [1] and Zombieload [6]. In this model, we assume that the attacker can execute the instructions on the same machine as the victim, for the attack using cache covert channel, the attacker process operates in an unprivileged environment and exploits the Overtake attack to extract confidential information from the victim process's memory, including data from kernel memory or $\mu$arch data sampling. As for using the PMU covert channel, the attacker is privileged.

### C. Vulnerable Instructions

In a Meltdown-type attack, a critical functional instruction needs to trigger an exception and enable data leakage through memory access. The original core code of Meltdown [1], shown at the beginning of Listing 1, obtains the secret data in line 2 and then left-shifts the secret data, encoding it into the cache covert channel in lines 3 and 4. We have identified two vulnerable instructions, namely `CMPSB` and `SCASB`. When combined with the `REP` prefix to form a complex instruction, they can serve as a facility for achieving a Meltdown-type attack in one or two instructions, as demonstrated at the end of Listing 1 for `CMPSB`. Unlike previous work that focuses on speculate operations [15] for Spectre-type attacks, which are also a subset of transient execution attacks orthogonal to Meltdown-type attacks, we focus on their capability to achieve Meltdown-type attacks with cache and PMU covert channels.

```
1 ; Original code of Meltdown
2 ;RCX = kernel address, RBX = probe array
3 MOV AL, BYTE [RCX]; Trigger Exception
4 SHL RAX, 0xC
5 MOV RBX, QWORD [RBX + RAX]
6
7 ; Overtake using CMPSB
8 ;RSI = kernel address, RDI = test value address
9 REPE CMPSB; Trigger Exception
```

Listing 1: Comparison of Transient Execution Code

The `REPE` and `REPNE` are `REP` prefixes that can be added to string instructions to repeat them several times, as specified in the `RCX` count register or the indicated condition of the `EFLAGS` register is no longer met.

*a) CMPSB:* Compares two memory operands of byte size and sets the status flags in the `EFLAGS` register according to the results. The source operands are located in memory and specified by the `RSI` and `RDI` registers. The source operand registers for memory load would automatically increment or decrement based on the `DF` flag.

*b) SCASB:* Is used to compare a byte in memory specified by the `RDI` register with the value in the `AL` register and sets the status flags in `EFLAGS` accordingly. Similar to `CMPSB`, the source operand register `RDI` would be automatically incremented or decremented based on the `DF` flag.

### D. Attack Steps

To execute a transient execution attack, we need to trigger and cope with an exception that creates a transient execution window that allows us to gain and send $\mu$arch information to the arch through a covert channel. We discuss our three attack steps as follows:

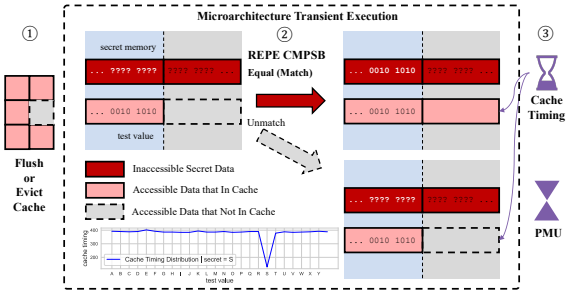


Fig. 2: Illustration of how CMPSB leaks secret data. If the test value is equal to the secret (match), the cache timing for the probe cache block would be much faster, which will also produce a different PMC difference value for some events.

1) Prepare the attack environment, which includes setting up the covert channel by flushing or evicting the cache, recording the Performance Monitoring Counter (PMC), etc.
2) Execute the attack instruction to trigger the exception and handle it. During this process, a transient attack window is created, allowing access to illegal secret data.
3) In the final step, the attacker receives information from the covert channel. This is accomplished by probing the cache state or recording the PMC value.

In detail, as shown in Fig. 2 and Listing 2 with stage annotated, for the cache covert channel, we allocate two continuous blocks (pages) of memory. Next, we flush the second block out of the cache, while keeping the first block in the cache. The last byte of the first block is set to different test values, and then the only attack instruction `REPE CMPSB` is executed to encode the secret data into the states of the second block. If the test value in the last byte of the first block matches the secret data, the second block will be loaded into the cache during transient execution. Otherwise, the second block would not be cached. Finally, we can recover the secret data by distinguishing whether the second block is cached, which can be achieved by measuring the access time of the second block. As for the PMU covert channel, we don't need to probe the state of the second block memory on the arch. Instead, we analyze the changes in the PMC in different test values.

## IV. Overtake Meltdown-type Attack Implement

In this section, we demonstrate how we utilize `CMPSB` and `SCASB` instructions to achieve our goal of conducting a Meltdown-type attack within one or two macro instructions. We refer to these attacks as Overtake Meltdown and Overtake Zombieload.

### A. Attack Template

For convenience, we have unified both the Flush+Reload and PMU covert channels into a single template that can be

used to covertly transfer $\mu$arch data during transient execution into the arch. The template is presented in Listing 2. In this template, Line 4 and Line 8 are used to record the PMC values before and after transient execution, respectively. Line 10 corresponds to the reload step of the Flush+Reload. The attack gadget is incorporated as the Transient Execution Gadget.

```
1  for (test_value=0; test_value < 255; test_value++) {
2    *(basic_addr - 1) = test_value;              // (1)
3    flush(basic_addr); // For Covert Channel       |
4    s_pmc = pmu_get_rdpmc(pmu_id);               // (1)
5    if (xbegin() == (~0u)) { // or !setjmp(jbuf)
6      // Transient Execution Gadget                (2)
7    }
8    e_pmc = pmu_get_rdpmc(pmu_id);               // (3)
9    if (reload_t(basic_addr) < cache_threshold) //  |
10       cache_channel[test_value]++;             //  |
11   pmu_channel[test_value] = (e_pmc - s_pmc);} // (3)
```

Listing 2: Pseudocode for Covert Channel Template.

As mentioned in Section III-D, the `basic_addr-1` and `basic_addr` represent the addresses of the last byte of the first block and the first byte of the second block, respectively. We place the `test_value` at the address of `basic_addr-1` and then ensure that the next memory block is kept out of the cache by flushing `basic_addr`. By utilizing the attack template, we can successfully decode the channel data into `pmu_channel` and `cache_channel` at the same time.

### B. Gadget of Overtake Meltdown

As shown in Listing 3, we utilize `CMPSB` and `SCASB` instructions to achieve Meltdown [1]. The `secret_addr` represents the Linux kernel direct mapping address of the secret data. Since the attacker does not have privileged access to `secret_addr`, an exception is triggered. By combining with the `RPEE` prefix, the `CMPSB` instruction will execute repeatedly and prefetch the subsequent data until the `EFLAGS.ZF` flag is set to zero. If the value at `secret_addr` equals the value at `basic_addr-1`, it triggers the prefetch to the next memory block. Otherwise, the `EFLAGS.ZF` flag will be set to zero, ceasing the `REPE` execution. The transient behavior of `SCASB` is similar to that of `CMPSB`.

```
1  // Overtake Meltdown one instruction gadget
2  asm ("REPE CMPSB\n"::
3       "RSI"(secret_addr+offset),"RDI"(basic_addr-1));
4  // Overtake Meltdown two instruction gadget
5  asm ("MOV (%%RBX), %%EAX \n"
6       "REPZ SCASB\n"::
7       "RBX"(secret_addr+offset),"RDI"(basic_addr-1));
8  // Overtake Zombieload one instruction gadget
9  flush(mapping_addr);
10 asm ("REP CMPSB\n"::
11      "RSI"(mapping_kaddr),"RDI"(basic_addr-1));
```

Listing 3: Pseudocode for Transient Execution Gadget.

### C. Gadget of Overtake Zombieload

We utilize `CMPSB` to achieve Zombieload [6] showed in Listing 3. The `mapping_addr` is a pointer to a legal memory, and the `mapping_kaddr` is a pointer to the Linux kernel direct mapping address of `mapping_addr`. Line 7 will cause a faulting load as it runs in unprivileged mode. Since we have flushed the `mapping_addr` in advance, there is no corresponding data in the cache that can be loaded for Line 7 during transient execution, but the Intel processor will radically forward the stale data in Line Fill Buffer (LFB) in transient execution, so it will transiently obtain the in-flight data of other programs LFB. If the in-flight data is equal to the test value, the cache block of `basic_addr` would be prefetched. We demonstrate that `CMPSB` also has MDS optimization.

## V. Experiment and Evaluation

### A. Environment Setup and Test Coverage

We have listed our tested environments in Table I. The Overtake Meltdown and Overtake Zombieload attacks were successfully implemented on Intel Xeon 6133, Core i7-6700, i7-6800, and i7-7700 processors. As the primary aim of Overtake is not to break through existing mitigation, we disabled the kernel page-table isolation (KPTI, a.k.a KAISER) during our experiments.

### B. Experiment Stages

Firstly, we utilize the cache covert channel to verify the feasibility of achieving the Overtake attack on the target machine. Subsequently, we optimize the attack code to improve throughput and reduce error rates. Once the optimization is completed, we consolidate the findings into an attack template. To obtain available PMU events, we developed a PMU automatic traversal and evaluation tool. This tool facilitates testing the PMU events provided by Intel to identify exploitable events. Leveraging the attack template developed earlier, we are able to rapidly test PMU events with high efficiency.

### C. Throughput and Error Rate

As the `CMPSB` and `SCASB` can only produce byte-level checks which require at most $2^8$ retries to match the checks. If the attacker knows the data range of secret data will greatly reduce the number of retries needed. To better evaluate the performance ceiling, we assume there are three levels of range for secret data: (1) only contain ASCII char from 48('0') to 57('9') which means 10 retries are required, (2) only contain ASCII of printable char that from 32 ('SP') to 126 ('~') which means 94 retries required, (3) random bytes that range from 0 to 255. For better performance and lower noise, we use Intel Transactional Synchronization Extensions (TSX) to cope with the execution. Replacing the TSX with the system exception signal handler could achieve a similar result but require thousands of retries to reduce high noise, resulting in a decline in the final speed.

For our Overtake Meltdown using `CMPSB`, we try to leak 1k random chars at their level above from another process and record the error rate. We define error rate as stealing the incorrect number of bytes divided total number of bytes. The original Meltdown could achieve speed up to 582 KB/s. For Overtake, in Intel i7-7700 CPU, we can achieve 770.1 KB/s with only an error rate as low as 0.4 % at level 1, 88.9 KB/s

TABLE I: Tested environments. A '✓' indicates that we have achieved an attack or carried out a specific analysis.

| Setup | Processor | $\mu$arch | Linux Kernel | Overtake Meltdown | Overtake Zombieload | Speed Analysis | PMU Analysis |
|---|---|---|---|---|---|---|---|
| Desktop | Xeon 6133 | Sky Lake | 5.10.0-0 | ✓ | ✓ | - | - |
| Desktop | i7-6700 | Sky Lake | 4.15.0-142 | ✓ | ✓ | - | ✓ |
| Desktop | i7-6800 | Sky Lake | 5.4.0-146 | ✓ | ✓ | - | - |
| Desktop | i7-7700 | Kaby Lake | 5.4.0-150 | ✓ | ✓ | ✓ | ✓ |

TABLE II: PMU events that can be utilized to leak 1k random characters with an error rate lower than 5%.

| Instruction | Event Code | Event Name | #PMC if Match | | #PMC if Unmatch | | Error Rate (%) |
|---|---|---|---|---|---|---|---|
| | | | $\mu$ | $\sigma$ | $\mu$ | $\sigma$ | |
| REPE CMPSB | 0x104110A3 | CYCLE_ACTIVITY.CYCLES_MEM_ANY | 340 | 124 | 115 | 21 | 1.7 |
| | 0x144114A3 | CYCLE_ACTIVITY.STALLS_MEM_ANY | 264 | 90 | 38 | 9 | 1.7 |
| | 0x004102A6 | EXE_ACTIVITY.1_PORTS_UTIL | 82 | 7 | 69 | 12 | 1.7 |
| | 0x004104A6 | EXE_ACTIVITY.2_PORTS_UTIL | 51 | 6 | 45 | 8 | 1.7 |
| | 0x04412479 | IDQ.ALL_MITE_CYCLES_4_UOPS | 70 | 2 | 60 | 12 | 1.7 |
| | 0x01413079 | IDQ.MS_CYCLES | 98 | 3 | 88 | 16 | 1.7 |
| | 0x00412079 | IDQ.MS_MITE_UOPS | 281 | 10 | 244 | 50 | 1.7 |
| | 0x01453079 | IDQ.MS_SWITCHES | 44 | 8 | 35 | 9 | 2.0 |
| | 0x00413079 | IDQ.MS_UOPS | 342 | 9 | 302 | 57 | 1.7 |
| | 0x004101A1 | UOPS_DISPATCHED_PORT.PORT_0 | 62 | 12 | 42 | 9 | 4.3 |
| | 0x004102A1 | UOPS_DISPATCHED_PORT.PORT_1 | 65 | 12 | 44 | 10 | 1.7 |
| | 0x004120A1 | UOPS_DISPATCHED_PORT.PORT_5 | 72 | 11 | 52 | 12 | 1.7 |
| | 0x004140A1 | UOPS_DISPATCHED_PORT.PORT_6 | 100 | 38 | 52 | 11 | 4.4 |
| | 0x014101B1 | UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC | 173 | 5 | 151 | 23 | 1.7 |
| | 0x024101B1 | UOPS_EXECUTED.CYCLES_GE_2_UOPS_EXEC | 83 | 4 | 75 | 12 | 1.7 |
| | 0x004101B1 | UOPS_EXECUTED.THREAD | 295 | 9 | 262 | 40 | 1.7 |
| REPZ SCASB | 0x104110A3 | CYCLE_ACTIVITY.CYCLES_MEM_ANY | 370 | 99 | 117 | 20 | 1.7 |
| | 0x144114A3 | CYCLE_ACTIVITY.STALLS_MEM_ANY | 259 | 108 | 38 | 9 | 1.7 |
| | 0x004102A6 | EXE_ACTIVITY.1_PORTS_UTIL | 90 | 10 | 72 | 13 | 4.2 |
| | 0x004104A6 | EXE_ACTIVITY.2_PORTS_UTIL | 55 | 5 | 47 | 8 | 1.7 |
| | 0x01413079 | IDQ.MS_CYCLES | 109 | 3 | 94 | 17 | 1.7 |
| | 0x00412079 | IDQ.MS_MITE_UOPS | 279 | 8 | 234 | 47 | 1.7 |
| | 0x01453079 | IDQ.MS_SWITCHES | 38 | 10 | 30 | 9 | 1.7 |
| | 0x00413079 | IDQ.MS_UOPS | 340 | 7 | 292 | 54 | 1.7 |
| | 0x0241019C | IDQ_UOPS_NOT_DELIVERED.CYCLES_LE_2_UOP_DELIV.CORE | 53 | 19 | 50 | 19 | 4.4 |
| | 0x004101A2 | RESOURCE_STALLS.ANY | 426 | 134 | 43 | 34 | 2.1 |
| | 0x004101A1 | UOPS_DISPATCHED_PORT.PORT_0 | 71 | 17 | 43 | 9 | 1.7 |
| | 0x004102A1 | UOPS_DISPATCHED_PORT.PORT_1 | 71 | 13 | 43 | 11 | 1.7 |
| | 0x004104A1 | UOPS_DISPATCHED_PORT.PORT_2 | 25 | 2 | 22 | 4 | 1.7 |
| | 0x004120A1 | UOPS_DISPATCHED_PORT.PORT_5 | 85 | 16 | 51 | 10 | 1.7 |
| | 0x004140A1 | UOPS_DISPATCHED_PORT.PORT_6 | 99 | 29 | 54 | 12 | 1.7 |
| | 0x014101B1 | UOPS_EXECUTED.CYCLES_GE_1_UOP_EXEC | 184 | 9 | 151 | 23 | 4.3 |
| | 0x024101B1 | UOPS_EXECUTED.CYCLES_GE_2_UOPS_EXEC | 86 | 5 | 72 | 11 | 4.3 |
| | 0x004101B1 | UOPS_EXECUTED.THREAD | 303 | 14 | 253 | 37 | 4.4 |
| | 0x0041010E | UOPS_ISSUED.ANY | 324 | 4 | 294 | 46 | 1.7 |

with only an error rate of 0.1 % at level 2, 31.4 KB/s with an error rate as 0.5 % at level (3). In addition, we also found that many PMU events can be used to achieve Overtake attack by using the PMU counter as a covert channel.

### D. Exploitable PMU Events

We listed the PMU events in Intel i7-6700 that can be exploited by `CMPSB` and `SCASB` to leak 1k random chars with an error rate lower than 5 % via Overtake Meltdown in Table II. And we got similar results on i7-7700. For this table, match means that the test value is equal to the secret data, thus the following cache block might be prefetched. Note that we listed only a subset of the exploitable PMU events.

Although PMU generally requires a privileged state to read, it can be used for potentially vulnerable analysis or attack a security environment higher than a privileged state such as a trusted execution environment (TEE).

### E. Experiment Result Analysis

We verified the effectiveness of the two instructions we found to achieve Meltdown-type attacks and can achieve a throughput of up to 770.1KB/s with a very low error rate in a specific victim data scenario.

From Table II, We can see that in different cases of whether the `test_value` matches the secret data, There is a gap between the mean ($\mu$) of #PMC changes, but the standard deviation ($\sigma$) is large., so more repeated are needed to reduce noise. For our PMU experiment, we retry 1000 times for each byte. Calculate the average value obtained from repeated attempts as the change of the PMU count value tested under each test value. Since the test value matches, more $\mu$OPs will be performed in the microarchitecture, so we assume that if matches the PMU change will be the largest.

Furthermore, via PMU analysis, we verify that the `REPE CMPSB` could emit load uncached data behavior, which depends on the secret value within transient execution, as

we could observe significantly different memory-related cycle stall between match and unmatch.

## VI. Discussion and Countermeasures

### A. Discussion

We have shown that using `REP CMPSB` and `REPZ SCASB` can be used to achieve a Meltdown-type attack with only one or two instructions. This approach surpasses the speed of the original Meltdown attack. Furthermore, our work changes the previous implementation pattern of the Meltdown attack, enabling all data-encoded instructions to only exist in the execution stage, and eliminating the need for macro instructions executed in the transient stage. This design change makes the attack's feature different from the original Meltdown [16], so it could be more stealthy and harder to detect.

Our proposed attack is successfully demonstrated on CPUs that are vulnerable to transient execution vulnerability. While researchers have proposed KPTI as a mitigation measure for these machines, recent studies indicate that Meltdown can still be carried out even on systems where KPTI is deployed [7]. Also, since KPTI will bring performance loss, leading some server operators will choose to disable it to make a trade-off between security and performance. Even though newer processors have addressed the Meltdown vulnerability at the hardware design level, older vulnerable processors remain widespread. And it will take time to replace all of them, making continued research on Meltdown attacks essential. In addition, our research can also bring insights to prevent similar attacks that may occur in the future.

### B. Countermeasures

*1) Hardware:* As the detail of the $\mu$arch implements for instruction is not public, we are not able to generate a solution on the hardware level. However, we can give some insights for the hardware implementers: executing the micro-OPs with more strict permission checks and stopping transient execution of $\mu$arch early.

*2) Software:* Despite being challenged, deploying KPTI remains the most effective mitigation for addressing the Overtake Meltdown attacks. With KPTI, a significant portion of the kernel memory is isolated from the user space, making it extremely difficult for transient execution attacks to leak the kernel data. As for the Overtake Zombieload vulnerability, Intel has released microcode updates that offer mitigation. This update introduces a side effect that clears the fill buffers and store buffer, to the rarely used `VERW` instruction. Additionally, operating systems could issue a dummy `VERW` instruction on every context switch to prevent the potential data leak.

## VII. Conclusion

Our research demonstrates that a Meltdown-type attack can be accomplished with just a single macro instruction. As a result, our work breaks the previous cognition and pattern to the Meltdown attack, making them more stealthy and more difficult to detect. The implications of this study underscore the urgency for ongoing research in processor security, calling for the development of robust defense mechanisms and the design of security processors.

## Acknowledgment

This work was supported in part by the NSFC General Technology Fundamental Research Joint Fund (Grant No. U1836215), National Natural Science Foundation of China (Grant No. 62072263), the Fundamental Research Funds for the Central Universities (Grant No. 2023RC71), Tsinghua University Initiative Scientific Research Program, BUPT Innovation and entrepreneurship support program (2023-YC-A163), and Zhongguancun Laboratory.

## References

[1] 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*, 2018.

[2] 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 *S&P*, 2019.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *Usenix Security*, 2018.

[4] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking data on Meltdown-resistant CPUs," in *CCS*, 2019.

[5] 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 *S&P*. IEEE, 2019.

[6] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad : Cross-privilege-boundary data sampling," in *CCS*, 2019.

[7] Y. Cheng, Z. Zhang, Y. Gao, Z. Chen, S. Guo, Q. Zhang, R. Mei, S. Nepal, and Y. Xiang, "Meltdown-type attacks are still feasible in the wall of kernel page-table isolation," *Computers & Security*, vol. 113, p. 102556, 2022.

[8] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A systematic evaluation of transient execution attacks and defenses," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

[9] I. Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual," 2022.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Sixth Edition: A Quantitative Approach*, 6th ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017.

[11] P. Qiu, Q. Gao, C. Liu, D. Wang, Y. Lyu, X. Li, C. Wang, and G. Qu, "Pmu-spill: A new side channel for transient execution attacks," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–12, 2023.

[12] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, 2014, pp. 719–732.

[13] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[14] A. Abel and J. Reineke, "uops.info: Characterizing latency, throughput, and port usage of instructions on Intel microarchitectures," in *ASPLOS*, 2019.

[15] O. Oleksenko, M. Guarnieri, B. Kopf, and M. Silberstein, "Hide and seek with spectres: Efficient discovery of speculative information leaks with random testing," in *2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2023, pp. 1737–1752.

[16] S. Mirbagher-Ajorpaz, G. Pokam, E. Mohammadian-Koruyeh, E. Garza, N. Abu-Ghazaleh, and D. A. Jiménez, "Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 1124–1137.