2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom) | 978-1-6654-1658-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/TRUSTCOM53373.2021.00154

Constant-Time Loading: Modifying CPU Pipeline to Defeat Cache Side-Channel Attacks

Yusi Feng[1,2], Ziyuan Zhu[1,2], Shuan Li[1,2], Ben Liu[1,2], Huozhu Wang[1,2], and Dan Meng[1]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China
Email: {fengyusi, zhuziyuan, lishuan, liuben, wanghuozhu, mengdan}@iie.ac.cn

***Abstract*—Cache side-channel attacks exploit cache state changes to steal confidential information. The emergence of transient execution attacks, a new form of microarchitecture side-channel attack that can access any address, makes cache side-channel attacks more threatening. Most of these attacks infer information by measuring the execution time of load operations. Therefore, from the microarchitecture perspective, we propose a novel countermeasure against cache side-channel attacks by eliminating the access time difference caused by cache hits or misses. We use the constant-time loading mechanism to limit each load instruction's execution to a fixed time and specify this mechanism's wake-up condition to avoid excessive performance loss. We modify the CPU pipeline to simulate this design and run SPEC2006 applications. The results show that the performance loss of our mechanism is negligible.**

***Index Terms*—Cache side-channel attacks, Timing channels, Secure processor architectures**

I. Introduction

There is a large access speed gap between cache and memory, and the cache is shared among multiple threads and processes. Therefore, the cache is easy to be exploited by attackers to construct time side-channels to infer the memory access mode of critical codes. The attacker extracts secret information from the victim's process by elaborately constructing a specific cache line or cache sets, waiting for the victim's program to run, and observing how it affects cache. Cache side-channel attacks have been used to break cryptographic algorithms [1]–[7] or spy on user inputs [8]–[10], destroy system-level protection mechanism [11] and even steal secrets inside SGX enclave memory or its internal registers [12]. The attack environments cover mobile devices, desktop computers, cloud platforms, etc.

In 2018, the emergence of transient execution attacks such as Meltdown [13], Spectre [14] and their variants made cache side-channel attacks more threatening. Attackers use hardware-level acceleration techniques, including out-of-order execution and branch prediction, to read confidential data at any location transiently. Although incorrect speculative executions will be squashed, changes in the microarchitecture state such as cache remain. However, these transient attacks only construct a covert cache channel, and then they use traditional cache side-channel attacks such as Flush+Reload, Evict+Reload to detect the change of cache state in this channel.

Many countermeasures of cache side-channel attacks have been proposed. CEASER [15] randomizes the mapping of cache lines by encrypting physical addresses, which increases the time and difficulties of attacks. [16] exploits cache hierarchy-aware core assignment and page coloring to statically partition the cache. SecDCP [17] uses dynamic way-partition technology to isolate cache sharing between different security domains. They reduce the effective utilization of the cache and may still leak cache occupancy information. Although researchers have proposed software-based mitigations against cache side-channel attacks, most of them will cause performance degradation, and it's hard to ensure their security [18].

We found that most defense methods are aimed at the codes part or the leaked components to mitigate or eliminate side-channels. They require modifications to the hardware, operating system, compiler, or other software, which can easily interfere with regular programs' operation and cause a lot of performance loss. However, the vast majority of cache side-channel attacks exploit the differences in access time to infer victims' memory access patterns. The behavior of measuring time rarely occurs in regular programs. We focus on interfering with the attackers to measure time, which has less impact on traditional codes and instructions execution.

There are two ways to interfere with program measurement time. The first is to add noise, and the second is to perform a constant execution time. TimeWarp [19] proposes to add two random delays after the RDTSC instruction, but each of its delays is too large. Solutions based on constant-time execution have been used effectively in the NACI cryptographic library [20] and OpenSSL [21], while they are limited to cryptographic. Most cache side-channel attacks measure the time it takes to load data at different addresses, so the defense range limiting the load operation to a constant time is wide.

There are three ways to measure the time of cache side-channel attacks, and the most commonly used is RDTSC instruction, which reads the value of the accurate time-stamp counter. It is highly precise and easy to use and is widely used in various time cache side-channel attacks. The second measurement method bases on the performance counter, and it requires a privileged level. It is also susceptible to interference from external programs, which is generally not easy to use. Thirdly, Attackers can also time the clock by software, repeating self-adding operation to the value of an address to

2324-9013/21/$31.00 ©2021 IEEE
DOI 10.1109/TrustCom53373.2021.00154

simulate clock timing. This behavior is easy to detect. [19] slightly modified the structure of the cache to prevent such timing methods. In summary, our defense scheme focuses on cache side-channel attacks that use RDTSC-like instructions to measure time.

This paper proposes a slight hardware modification scheme for the CPU pipeline, which can identify potentially dangerous sensitive instructions sequences. Then, for a certain amount of time, regardless of cache hits or misses within a certain period, the memory load instruction will be executed for a fixed time. Attackers cannot obtain information from the measured time. This scheme prevents cache side-channel attacks that use RDTSC to measure time. Our defense thought can also be extended to other timing side-channel attacks.

Our design divides into two parts. The wake-up module judges the macro-operation code of the instruction in the fetch phase. When an RDTSC or RDTSCP instruction is detected, this module initializes a specific timer and starts the constant-time loading mechanism within a certain period. This mechanism is used in the execution and write-back stages so that the memory read instructions should be written back after a fixed number of cycles are executed. Compared to software modifications, it's more difficult to bypass hardware modifications. Our countermeasure only requires minimal hardware modifications, realizes constant-time execution of sensitive load instructions, does not rely on operating systems, compilers, and software, and has little impact on the regular program running.

In summary, this paper makes the following contributions:

- We propose a constant-time mechanism to mitigate the cache side-channel attack based on time measurement by changing the execution and write-back of sensitive memory read instructions in the pipeline.
- We regard the RDTSC or RDTSCP instructions detected in the fetch stage as the constant-time loading mechanism's wake-up condition.
- We simulated our design on gem5, and it successfully defended against most cache side-channel attacks such as Flush+Reload and Prime+Probe. The performance penalty for running 18 SPEC applications with inserted RDTSC instructions on our simulation is minimal.

II. Background

A. Cache-Based Side-Channel Attacks

Over the past few decades, various attack methods have emerged one after another. The attack environment has gradually changed from single-core to cross-core and from microprocessors to cloud environments and trusted execution environments. It was initially applied to attacks on encryption algorithms [22]. Ristenpart [23] et al. used Prime+Probe to steal user input information across virtual machines on the cloud platform, and Zhang [24] et al. realized cross-virtual machine private key extraction. Yarom and Falkner implemented a high-resolution, low-noise cross-core cache side-channel attack Flush+Reload [7], and then Liu and others

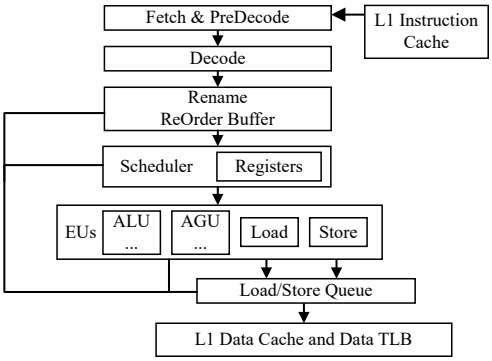

Fig. 1. Schematic diagram of generic CPU pipeline.

[4] improved the Prime+Probe attack to make it applicable to cross-core. Processor manufacturers all claim that their trusted execution environments are safe, but Lipp [9] and others attacked the trustzone of ARM and stole user information. Subsequently, Schwarz [25] and others used SGX to hide the cache attack to obtain the user's private key. In 2018, Meltdown [13] and Spectre [14] used cache as a covert channel to transmit information during the CPU's transient execution.

B. CPU Pipeline

Figure 1 depicts today's superscalar processors' general pipeline design and the main stages of instruction execution. First, the CPU fetches instructions from the instruction cache and decodes them into macro-ops and micro-ops in the predecode and decode phases. Then renames the logical registers of the instructions to physical registers to avoid delays caused by instruction dependencies (write-after-write and write-after-read delay). Record the instructions and related information in the reorder buffer (ROB) and corresponding issue queues, waiting for the source data for out-of-order execution. When all the instruction source operands are ready and the execution unit is available, execute the instruction. Finally, the result will be committed in order in the ROB.

C. Leakage Exploitation Techniques

This subsection introduces seven state-of-art leakage exploitation techniques of cache side-channel attacks. Prime+Probe exploits sets of the associative cache. It can be attacked on L1-data cache [5], [6], L1-instruction cache [26], and last level cache (LLC) [18]. Evict+Time [27] evicts the victim's specific cache lines and measure the runtime of the victim. Evict+Reload [8] attacks evict specific cache line. After the victim is executed, reload the cache line. The advantage of Flush+Reload [7] attacks is that the *clflush* instruction can precisely evict the specified memory line in the shared page from all cache levels. Flush+Flush [28] is a variant of the Flush+Reload attack, which exploits the time differences in the execution of *clflush* instructions. Prime+Abort [29] uses Intel TSX hardware. The victim process will be aborted if it evicts the cache entry prepared by the attacker. The newly

proposed Reload+Refresh [30] leaks information by exploiting the cache replacement strategy.

D. Transient Execution Attacks

Most of the Spectre attacks exploit control flow misprediction to create transient instructions. Spectre V4 [31] and MDS use address speculation and Meltdown uses transient execution caused by virtual memory exceptions. These attacks use transient execution to load information at any address into the cache and then use cache side-channel attacks such as Flush+Reload and Evict+Reload to obtain changes in the cache state, thereby stealing information.

III. Threat Model

This paper assumes that the attackers have no privileges and software-based time measurements have been prevented by hardware modifications, so the attacker can only use RDTSC-like instructions to measure the time.

Our defense strategy aims to prevent all cache-side channel attacks that use RDTSC-like instructions to measure memory load instructions' execution time. For the attacks discussed in section 2.3, in addition to Flush+Flush and Prime+Abort, the other five cache side-channel attacks all obtain secret information by monitoring the time difference of the load instruction execution. The constant-time loading mechanism proposed in this paper can resist the above five types of timing attacks. Simultaneously, transient execution attacks cannot use the above five cache side-channel attack methods to obtain their cache state changes.

We work with all x86 processors since the Pentium and Alpha systems can use RPCC instructions instead of RDTSC instructions. The attacker and victim processes run on the same thread, cross core, or SMT does not affect the execution of the defense mechanism proposed in this paper. Our mechanism will only affect instructions of a certain thread and will not affect the normal execution of other threads.

IV. Defense Strategy

As mentioned earlier, this paper aims to defend against cache side-channel attacks that use RDTSC instructions to measure the execution time of memory load instructions to infer information. We list the attacks that can be handled in section 3 and describe them in detail in section 2.3. We found that these attacks depend on the time differences in executing one or a set of memory load operations. This paper focuses on eliminating these time differences. We take the following two most commonly used attacks as examples.

A. Analysis of Existing Attacks

1) Flush+Reload:

Listing 1. Code for the Flush+Reload technique.

```
1 lfence
2 rdtsc
3 lfence
4 movl %%eax, %%esi
5 movl (%1), %%eax
6 lfence
7 rdtsc
8 subl %%esi, %eax
9 clflush 0(%1)
```

The target of Flush+Reload's attack is the memory area shared with the victim. It uses the unprivileged *clflush* instruction in x86 to evict the specified memory lines from all cache levels. After the victim process is executed, reload the memory line. If a cache hit occurs, it means that the victim has accessed this shared memory line.

The Flush+Reload code in Listing 1 shows that this attack only records the execution time of a specific address load at a time. By comparing this time with the access time threshold of a cache hit, the victim's access information can be obtained. This attack method has the advantages of cross-core, high bandwidth, and fine-grained.

2) Prime+Probe:

Listing 2. Code for probing one 8-way set.

```
1  lfence
2  rdtsc
3  mov %eax, %edi
4  mov (%r8), %r8
5  mov (%r8), %r8
6  mov (%r8), %r8
7  mov (%r8), %r8
8  mov (%r8), %r8
9  mov (%r8), %r8
10 mov (%r8), %r8
11 mov (%r8), %r8
12 lfence
13 rdtsc
14 sub %edi, %eax
```

Prime+Probe is a representative of attacks that are not based on shared pages. It uses its data to fill different cache sets. After the victim program is executed, the attacker detects the time of loading each set. If the victim uses a specific cache set, the data line filled by the attacker will be evicted, and it will take longer for the attacker to load it again. Therefore, the attacker can guess which cache set the victim used.

Listing 2 shows the assembly code Prime+Probe used to probe one set of the cache. They design all the lines in this set as a mutually dependent list so that the memory access instructions in the code are sequenced. Lfence instruction provides a barrier for the load instructions. After the previous lfence guarantees that all previous read instructions have been executed, the timing and access to the data set to start, and the latter lfence instruction guarantees that the timing ends after the data set access is completed. Use RDTSC instruction to record the execution time of this string of mov instructions in the eax register.

B. Design Overview

Obtaining the execution time of the specific load operations is a critical step in most cache side-channel attacks. Through our analysis , we have derived the following standards as the conditions for achieving a secure load mechanism:

- Each memory load instruction that may be dangerous should be executed for a constant time;

- The load instruction that has a data dependency with the previous instruction must wait for the previous load instruction to execute for a fixed period of time before it can be executed.

Therefore, we propose a hardware-based constant-time defense method, which enables fixed-time execution of load instructions under certain conditions. This method is not as easy to be bypassed as the software method, and the harsh opening conditions of the constant-time loading mechanism will minimize the impact on benign programs. In the meantime, we also strive to make it practical by maintaining acceptable implementation cost and complexity.

C. Constant-Time Loading Mechanism

Attackers need to obtain the execution time of specific load operations to perform cache side-channel attacks. This paper proposes a general design of the Core microarchitecture for the constant-time loading mechanism. After the mechanism is turned on, the write-back stage of load instructions is restricted so that the execution time of each instruction is constant as the load time of a normal last-level cache miss. The total time detected by the attacker is always close to a fixed value, so the victim's access information cannot be inferred by the detection time.

In the pipeline, after the load instruction obtains the data, the data is written back to the destination register, and then the dependent instruction is awakened. (If the source operand of the subsequent instruction comes from the destination operand of the previous instruction, there is a dependency between them.) Once the mechanism is enabled, no matter how long it takes to retrieve the data. Only after the load instruction is issued for a specific period, the scoreboard can be modified, and the dependency instruction can be awaked. At the same time, we can also ensure that load instructions without dependencies can be executed out of order to optimize performance.

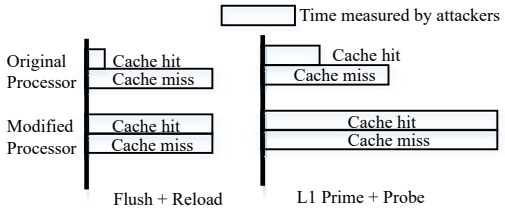

Fig. 2. Schematic diagram of time measured by attackers.

As shown in Figure 2, take Flush+Reload and L1-level Prime+Probe attacks as examples. In original processors, if the victim accesses the cache line flushed by the attacker, the attacker will gain the time of a cache hit. If the victim accesses a cache set that the attackers primed, the attacker will get the time of a set of L2 cache hit. After the constant-time load mechanism is enabled, no matter whether the cache misses or cache hits, attackers will always get a fixed measurement time.

D. Wake-up Condition

Always opening the constant-time loading mechanism will cause significant performance loss, so we propose a measurement-driven wake-up condition. As mentioned earlier, using RDTSC instruction is the most common and reliable time measurement method in attacks. This instruction is often used with serialized instructions such as lfence and mfence to ensure the measurement time's accuracy. It will not be affected by the out-of-order execution processor. When any thread detects the RDTSC instruction, we will wake-up the constant-time loading mechanism for this thread. Although a pair of RDTSC is needed to measure time, we will not close the mechanism after encountering a second or more RDTSC instruction because an attacker can execute multiple RDTSC instructions before the attack code fragment to close our mechanism, thus bypassing the mechanism.

In order to minimize the performance loss and ensure that subsequent attacks based on time readings are invalid, reasonable shutdown conditions need to be formulated. We analyzed the characteristics of the code snippets used by the attacker to measure time. In the process of measuring time, some attackers only need to perform a load operation, such as Flush+Reload, and attackers need to access each cache set at most, such as Prime+Probe. Too many load instructions will affect the fidelity of time measurement and bring complex analysis processes. Therefore, we choose to use the time it takes to access a set of cache lines under the constant-time loading mechanism as the maximum time that the mechanism can be woken up at a time.

V. Constant-Time Loading Implementation

A. Overview

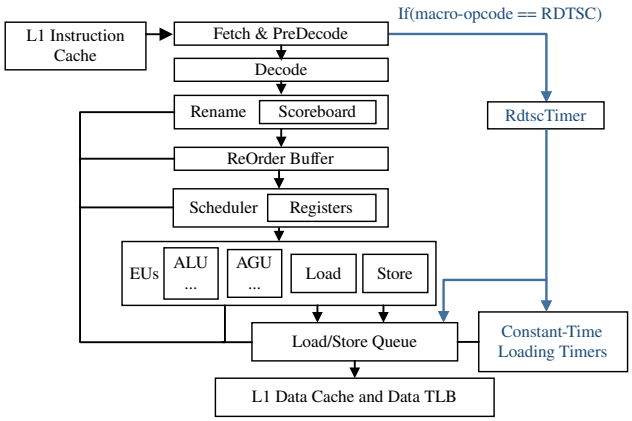

Fig. 3. The overall design process of constant-time loading.

Figure 3 shows the overall design process and integration with the microarchitecture pipeline of the Gem5 microprocessor. In the pre-decode stage, judge whether the instruction is RDTSC according to the macro-opcode. If so, activate the timer corresponding to the thread. Each timer is set with a countdown timer. After activation, the time will be reset

to the maximum value. Decrease by one every cycle until the timer output becomes 0. As long as the timer is not 0, the constant-time loading mechanism is always on. After the scheduler issues the instruction, it will be inserted into the load queue. Record the time in the load queue when this instruction starts to execute. After the execution is completed, write the value back to the target register after the instruction has been executed for a specific time, edit the scoreboard, and wake up the dependent instruction.

We set a flag bit for each load instruction in the load queue to mark whether the instruction should follow the constant-time loading mechanism. When inserting a load into the load queue, if the RDTSC timer is not 0, this flag bit will be 1.

B. Wake-Up Condition

When an RDTSC instruction is encountered in the pre-decoder, activate the RDTSC timer of the corresponding thread to decrement from the default setting's maximum value. After the set period, the value of the counter is reduced to 0. Every time it is activated, no matter what the counter's value is, it will reset to the maximum value. When the timer's value is not 0, the limit on the execution time of the load instruction is woken up.

C. Constant-time Loading Mechanism

1) Constant-time Loading Unit Design:

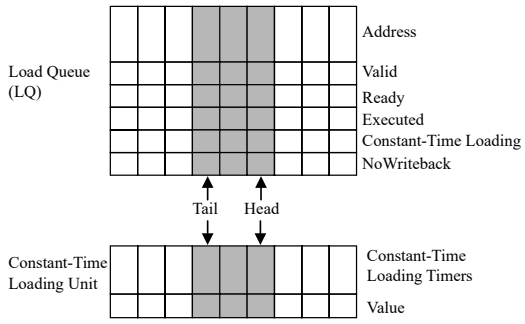

Fig. 4. Integration of the load queue and the constant-time loading units.

As shown in Figure 4, to minimize the modification to the original processor, we refer to the module design of InvisiSpec [32] to integrate the constant-time loading module into the load queue, map a constant-time unit one-to-one for each load instruction. This design makes it easy to assign a constant-time loading unit to each load entry in order, delete the value of load instructions to be committed, and squash load instructions.

The constant-time loading unit includes a time counter and a buffer for storing the destination address's value. Each LQ entry has some status bits: Valid, Ready, Executed, Constant-time Loading, and NoWriteback. Valid records whether the entry is valid. The Ready bit indicates that the source operand of the load instruction is ready. The value of Executed bit equals one means that the instruction has started to execute. The Constant-Time Loading bit reflects whether the instruction is a constant-time loading operation. NoWriteback bit is 0 indicates that the constant-time loading timer has not been cleared, so that the write-back operation will be delayed. The default values of Constant-Time Loading and NoWriteback are both 0.

2) Operations of Constant-time Loading:

Key events are as follows:

A load instruction is executed: Load queue executes the load instruction with the ready bit is 1 in the order of fetch, sets the executed bit to 1 when starting execution, and checks the value of the RDTSC timer of the corresponding thread. If the value is 1, activate the corresponding instruction constant-time loading timer, and mark the constant-time loading bit as 1.

A load instruction gets the memory line it requests: In the normal execution of the load instruction, after obtaining the requested value, it writes the value back to the register, and at the same time wakes up the instructions that dependent on this instruction, and marks the corresponding ROB entry as ready for committing. Under the mechanism proposed in this paper, the write-back operation needs first to access the value of the constant-time loading bit. If this bit is 0, write back as usual. If it is 1, check the value of the constant-time loading timer. If this timer is 0, it will continue as usual. If it is not 0, write the value into the constant-time loading unit and delay the write-back operation execution.

The constant-time loading timer returns to 0: This means that a fixed period has passed since the load instruction's execution starts. At this time, the NoWriteback bit is set to 1, and the load queue scans the NoWriteback bit in each cycle. Suppose a NoWriteback bit is set to 1. In that case, it will obtain the value of this instruction from the corresponding constant-time loading unit and execute the series of delayed write-back operations of the instruction.

VI. Experiment and Evaluation

A. Experimental Setup

We implemented our design on cycle-accurate computer-system architecture simulator Gem5 [33]. We have conducted a comprehensive evaluation of the safety and performance of Constant-time loading.

First, we have determined the specific time value of the constant time of the load instructions through experiments and the time that the constant-time loading mechanism should be turned on after the wake-up condition is opened. Secondly, to verify the design's safety, we run multiple Proof-of-Concept (PoC) codes on the original gem5 processor and the gem5 processor edited by constant-time loading. Experimental results show that our design resists these attacks. To ascertain the performance loss, we run SPECInt2006 and SPECFP2006 applications [34]. We use reference input sets, starting from the 100 millionth instructions of each test sample and then simulating the next 100 million instructions. Since there are almost no RDTSC instructions used in the SPEC2006 program, we randomly inserted RDTSC and RDTSCP instructions to simulate real execution environments and observed their impact on performance. We refer to the unmodified gem5

as Base and the modified version as Constant-time Loading. Table I lists the processor parameters we simulated.

TABLE I
PARAMETERS OF THE SIMULATED ARCHITECTURE.

Parameter	Value
Architecture	One core (SPEC) at 2.0GHz
Core	8-issue, out-of-order, no SMT, 32 Load Queue entries, 32 Store Queue entries, 192 ROB entries
Private L1-I Cache	32KB, 64B line, 4-way, one port, two cycles data latency
Private L1-D Cache	64KB, 64B line, 8-way, 3 Rd/Wr ports, 2 cycles data latency
Shared L2 Cache	Per core: 2MB bank, 64B line, 16-way, 20 cycles data latency
Coherence Protocol	Directory-based MESI protocol
DRAM	RT latency: 50ns after L2

B. Determination of Time Parameters

To determine the specific time value of the constant-time in the constant-time loading module, we run Spectre V1 attack based on Flush+Reload on the unmodified gem5 simulator and set the secret value to 100. After the transient execution of the misprediction operation ends, sequentially access the values of the probe array. After repeating the measurement 100 times, we record the average time spent accessing each address, as shown in Figure 5.

Fig. 5. Access time of Flush+Reload based spectre V1 attack.

It can be seen from the figure that in the unmodified processor, when the array index is 100, the access time is 25 cycles, and the access delays of the remaining addresses are basically between 160-190, and the attack is successful. Therefore, in this processor, we set the constant-time to 190 cycles. After turning on the load time limit of 190 cycles, we rerun the above program and find that the measured times are all 395 cycles, nearly equal to twice the 190 cycles. This is because the Spectre attack program we are running uses * address to access data. If an attacker wants to access an address's value, he must first obtain the address he wants to access. However, this address will not be an immediate number, so he must first access the variable storing the address to obtain the address and then access the address to get the value on this address. Therefore, access to the value at each address requires two load operations. If we do not use parameters to access, directly access its address, or run other data access operations that are not dependent, the access time obtained by the tests is 201 cycles.

Among the attacks we can resist, the Prime+Probe attack takes the longest time to measure. It needs to detect the time it takes to access a cache set, while other attacks only need to measure the time required to load a single address, and it also uses * address to access data. So, we use $constant{-}time \times ways\ of\ shared\ L2\ cache \times 2$ as the maximum time that the constant-time loading mechanism can be opened at a time.

C. Security Analysis

We run several representative and most commonly used attacks in the threat model. L1-level Prime+Probe attacks use the trojan array to fill up the L1-D cache. After the victim program is executed, access each cache set of the L1-D cache separately, use RDTSC instruction to record the access time, and then infer the victim's access information. LLC-level Prime+Probe uses the trojan array to fill up the last-level cache (here L2 is LLC). After the victim program is executed, access each cache set of the LLC and record the access time to obtain information. The address access of the Prime+Probe attack is nested layer by layer. The second cache line address is placed in the first cache line of the same cache set, forming a dependency relationship, preventing load instructions from being executed in parallel, and increasing time resolution. The Flush+Reload attack uses the *clflush* instruction to expel the cache of a specific address. After the victim's program is executed, access the address and record the access time to determine whether the victim has accessed the address. Figure 6 shows the average access time of 100 attacks in Base and Constant-time Loading modes.

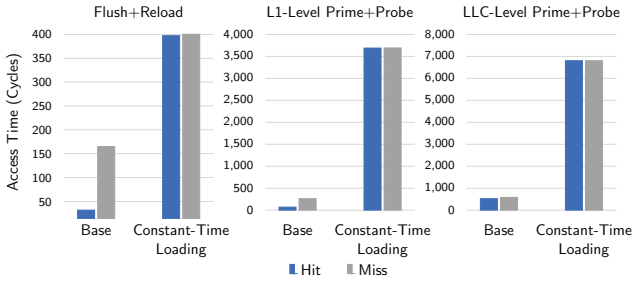

Fig. 6. The measured time of several attack codes on Base and modified processors.

Like most attack codes, all the attack programs we run use * address to access data, so it performs twice as many load operations as direct access. For L1-level Prime+Probe, in the Base mode, the average time measured when the attacker all hits L1 is 68 cycles, and the average access time for L1 misses 260 cycles. In Constant-time Loading mode, the hit time is 3686 cycles, and the missed time is 3689 cycles. The time difference measured by LLC-level Prime+Probe hit or miss LLC in Base mode is 61, and the time is the same in Constant-time Loading mode. In the Flush+Reload attack, the

two modes are 32, 165, 397, and 400, respectively. The results show significant differences in the access time measured in the Base mode, while in the Constant-time Loading mode, the access time measured by the attacker is the same. Our mechanism successfully defends against these attacks, and the attacker cannot obtain information through the attack code, which verifies the previous threat model.

Our defense method can defend against existing attack codes that use RDTSC-like instructions to measure the time of load operations. Suppose the attackers want to bypass our defense by changing their code. In that case, they must delay or wait for $constant{-}time \times ways\ of\ shared\ LLC\ cache \times 2$ cycles (the maximum time that the constant-time loading mechanism can be opened at a time) before executing one of their original malicous load instructions. The time required for attackers to complete the access operation is delayed by dozens of times. Taking our simulated processor's parameters as an example, the attacker originally takes no more than 190 cycles to complete an access operation. However, after adding our mechanism, this time has increased by at least 31 times, while the multiples in other processors are even more. Nowadays, commonly used processors run multiple threads simultaneously, and different cores share the last level of cache. The content and state of the cache are constantly changing over time. The attacker's subsequent access operations have lower accuracy rate and higher noise, and it may be necessary to repeat the attack multiple times to determine the status of a cache line. Although the attackers change the attack code for our defense measures, their attacks are challenging to succeed. We can also increase the maximum opening time of our mechanism to defend them. Besides, timers in our design are thread-specific, so attackers can only affect themselves threads.

D. Performance Evaluation

We run SPEC2006 in Base and Constant-Time Loading modes. However, because there are no RDTSC and RDTSCP instructions in SPEC2006, the execution time measured in the two modes is entirely equal, showing the frequency of RDTSC instructions in regular programs is very low. To perform performance evaluation of our design, we insert an RDTSC instruction every time 1 million SPEC2006 instructions are run. As shown in Figure 7, from left to right, we offer the time it takes to run the original version of SPEC and the SPEC program inserting RDTSC instructions on the processor modified by Constant-Time Load. They are all based on the time that SPEC2006 runs in Base mode. At the same time, we also show the average, highest and lowest number of load instructions affected by an RDTSC instruction for each SPEC application.

As can be seen from the figure, the performance loss caused by RDTSC instructions is minimal. The average execution time is 0.181% higher than the Base time, and an RDTSC instruction affects 392 load instructions on average. The lbm application has the least number of instructions affected, with an average of 208, but its performance loss is the highest

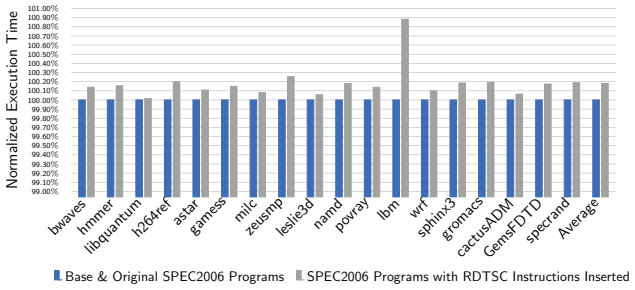

Fig. 7. The execution time of SPEC2006 applications with or without RDTSC instructions.

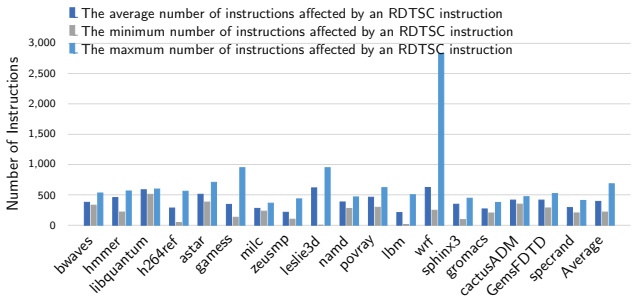

Fig. 8. The number of instructions affected by an RDTSC instruction.

at 0.882%, much higher than other applications. The wrf application has the largest number of affected instructions, but its performance loss is relatively low.

The reason for the additional performance loss caused by Constant-time Loading is that the load instruction is forced to delay the execution time. Independent load instructions can be executed simultaneously or consecutively, so they will only cause a constant-time delay in total. While load instructions that have data dependencies with previous instructions must wait for the previous instructions to execute for a fixed period before being executed, which doubles the program execution time.

To find out the reasons for different performance losses of various applications, we analyze all the fetched instructions when the Constant-Time Loading mechanism is turned on. The lbm application simulates incompressible fluids in 3D and has the highest performance loss. The load instruction in the lbm program loads A address's value into the t1d register and uses the OR operation to slightly change the value in t1d, and then writes it back to the A address. Most of the affected load instructions are the same, and similar instructions depend on them. These instructions need to wait for the load instruction to pass a fixed time before they start executing, which slows down the program.

The performance loss of libquantum application is the smallest, it has 58421 load instructions affected, but the performance loss is only 0.015%. By analyzing the micro-op, it can be found that its multiple load instructions read the values of multiple consecutive addresses simultaneously. There are no

dependent instructions. Therefore, numerous load instructions can be executed consecutively, and other instructions can also be run out of order usually. The degree of interference to this application is minimal. The wrf application with low-performance penalty accesses a large number of consecutive addresses and has no dependencies.

We also observe the bwaves application whose number of affected load instructions and performance loss is moderate. We found that it accesses many consecutive addresses, and each load instruction does not depend on each other. Occasionally, the instructions after the load instruction depend on the value of the load instruction.

VII. Related Work

There are already some countermeasures based on the idea of constant-time. [35] suggests that CPU designers add a new CPU instruction, which allows the process to load specific data into the L1 cache with a constant number of cycles. The number of cycles is not affected by hit or miss. This instruction is only for AES operations. [36] uses time padding to pad the execution time of a protected function to the worst-case runtime of the function. This solution only covers sensitive code that the programmer has marked. The offline profiling tool needs to be used in advance to test the worst-case execution time of each function on a specific hardware platform.

Some countermeasures interfere with the attacker's observation time. TimeWarp [19] proposes to add an actual random delay and another random virtual delay after the RDTSC instruction to limit the fidelity of fine-grain timekeeping. Simultaneously, it also modifies the cache structure to trigger a blank interrupt when an attack that uses the virtual Time Stamp Counter to measure time is detected. Its defense method is backward compatible, but its random delay time is 2^{12} to $2^{13} - 1$ cycles, and the performance loss is big. [37] proposes using the number of executed instructions instead of time to control threads' switching. It eliminates cache-based timing attacks with instruction-based scheduling, but the internal timing attacks it defends are too basic, and its use range is very narrow.

VIII. Conclusion

Cache side-channel attacks are a severe security problem. In recent years, these attack techniques can be exploited with microarchitecture designs' vulnerabilities to steal confidential information, bringing more significant security threats. It is difficult to defend against multiple attacks and maintain the low-performance loss.

This paper proposes a novel method and implementation defend against cache side-channel attacks from the microarchitecture perspective by slightly modifying instruction execution logic in the pipeline. Within a specific time slice, after the fixed instructions are recognized, we restrict the execution time of specific load instructions to a fixed value, thereby eliminating the access time difference caused by cache hits or misses. Our mechanism will only affect instructions of a certain thread and will not affect the normal execution of other threads. Our solution can prevent most types of cache side-channel attacks and transient execution attacks based on them. In addition, it can be combined with other detection or mitigation measures. The average performance loss of our simulations in 18 SPEC workloads is negligible.

IX. Acknowledgment

This work is supported by the national key research and development plan of China (2018YFB2202104).

References

[1] N. Benger, J. Van de Pol, N. P. Smart, and Y. Yarom, ""ooh aah... just a little bit": A small amount of side channel can go a long way," in *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 2014, pp. 75–92.

[2] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S $ a: A shared cache attack that works across cores and defies vm sandboxing–and its application to aes," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 591–604.

[3] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2014, pp. 299–319.

[4] 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.

[5] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*. Springer, 2006, pp. 1–20.

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

[7] 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.

[8] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 897–912.

[9] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *25th USENIX Security Symposium (USENIX Security 16)*, 2016, pp. 549–564.

[10] M. Schwarz, M. Lipp, D. Gruss, S. Weiser, C. Maurice, R. Spreitzer, and S. Mangard, "Keydrown: Eliminating software-based keystroke timing side-channel attacks," in *Network and Distributed System Security Symposium*. Internet Society, 2018.

[11] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida, "Aslr on the line: Practical cache attacks on the mmu." in *NDSS*, vol. 17, 2017, p. 26.

[12] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. Lai, "Sgxpectre: Stealing intel secrets from SGX enclaves via speculative execution," in *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*. IEEE, 2019, pp. 142–157. [Online]. Available: https://doi.org/10.1109/EuroSP.2019.00020

[13] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[14] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[15] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 775–787.

[16] H. Raj, R. Nathuji, A. Singh, and P. England, "Resource management for isolation enhanced cloud services," in *Proceedings of the 2009 ACM workshop on Cloud computing security*, 2009, pp. 77–84.

[17] Y. Wang, A. Ferraiuolo, D. Zhang, A. C. Myers, and G. E. Suh, "Secdcp: secure dynamic cache partitioning for efficient timing channel protection," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[18] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge aes against cache-based software side channel vulnerabilities." *IACR Cryptol. ePrint Arch.*, vol. 2006, p. 52, 2006.
[19] R. Martin, J. Demme, and S. Sethumadhavan, "Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 118–129.
[20] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 159–176.
[21] "Openssl," http://www.openssl.org, accessed Feb 15, 2021.
[22] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." *IACR Cryptol. ePrint Arch.*, vol. 2002, no. 169, pp. 1–23, 2002.
[23] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, 2009, pp. 199–212.
[24] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 305–316.
[25] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 3–24.
[26] O. Acıiçmez and W. Schindler, "A vulnerability in rsa implementations due to instruction cache analysis and its demonstration on openssl," in *Cryptographers' Track at the RSA Conference*. Springer, 2008, pp. 256–273.
[27] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, 2010.
[28] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: a fast and stealthy cache attack," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 279–299.
[29] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. Tullsen, "Prime+ abort: A timer-free high-precision l3 cache attack using intel TSX," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 51–67.
[30] S. Briongos, P. Malagón, J. M. Moya, and T. Eisenbarth, "Reload+ refresh: Abusing cache replacement policies to perform stealthy cache attacks," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1967–1984.
[31] "speculative execution, variant 4: speculative store bypass," https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, accessed Feb 15, 2021.
[32] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2018, pp. 428–441.
[33] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.
[34] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.
[35] D. J. Bernstein, "Cache-timing attacks on aes," 2005.
[36] B. A. Braun, S. Jana, and D. Boneh, "Robust and efficient elimination of cache and timing side channels," *arXiv preprint arXiv:1506.00189*, 2015.
[37] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières, "Eliminating cache-based timing attacks with instruction-based scheduling," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 718–735.