**[1]Wuzheng Tan**
**[1]Zuobo Xiao**

# Side Channel Research on CPU Privilege Mode Switching Boundary

Journal of Electrical Systems

***Abstract: -*** Modern computer systems have evolved from basic page table isolation to kernel page table isolation in order to isolate the memory of individual modules, which not only allows the memory of individual processes to be isolated, but also the memory of the kernel and the user to be isolated. In this paper, we introduce a side-channel attack at the boundary of privileged mode, which utilizes the chaotic execution strategy of the CPU instruction pipeline and the timing before the kernel page table is switched to leak the kernel data into the micro-architecture and extract it using the cache side-channel to break the kernel page-table isolation mechanism in an ingenious way. This special micro-architecture-side channel does not depend on any software vulnerability and is OS independent. Using this attack, the kernel memory can be read freely in the user process space, affecting a large number of cloud service computer users as well as personal computer users.

***Keywords:*** micro-architectural side channel; kernel leakage; KPTI

## I. Introduction

Modern CPUs often use aggressive pipeline strategies[1-2] (e.g., out-of-order execution, speculative execution) in exchange for higher IPS (instructions per second) performance. The CPU does not execute only one instruction at a time in the pipeline. It has multiple computing units and matches as many instructions as possible for simultaneous execution. The execution order of these instructions is often inconsistent with the original order, which can lead to the possibility of user-mode instructions being inserted into kernel-mode execution. Although the final result of out-of-order execution will not be retained at the memory level, this situation inevitably brings about the side effect of micro-architectural side channels.Under the strategy of enabling KPTI[3] (kernel page table isolation) in the operating system, the kernel only maps the necessary addresses to the user address space. Even if there is a micro-architectural side channel in user mode, kernel data cannot be obtained because the user space cannot find the kernel address. Thus, the side effect of the micro-architecture is alleviated to a certain extent. However, this is not always the case. If user-mode instructions can be executed instantly in kernel mode, then kernel data can be obtained with the help of kernel page tables, breaking the restrictions of KPTI and allowing kernel data to be read arbitrarily.This paper studies the Meltdown vulnerability based on x86 CPU. The main research contents and innovations are as follows: For the first time, the special side effects of the pipeline out-of-order execution strategy at the CPU privilege switching boundary are studied. This side effect is caused by out-of-order execution. If user-state instructions and kernel-state instructions are fetched into the pipeline simultaneously, and the execution dependency of the user-state instructions is satisfied before the kernel-state instructions, the user-state instructions that would not have been executed will be executed out of order and instantaneously. The side channel is used to leak micro-architectural information, which constitutes this special side effect.

## II. Materials and Methods

This chapter mainly introduces the preliminary knowledge required for this article. First, it covers topics related to the CPU pipeline, including the CPU pipeline's out-of-order execution strategy, CPU cache side channels, KPTI (Kernel Page-Table Isolation) mechanism, and CPU privilege levels.

### *A. Out-of-order execution*

In order to increase the number of instructions that can be executed per second, modern CPUs almost all use pipeline mechanisms. The execution of an instruction is generally divided into five stages: fetching, decoding, executing, accessing memory, and writing back. To compensate for the disadvantages of the reduced instruction set in pipeline

*Corresponding author: Zuobo Xiao, xiaozuobo@stu2021.jnu.edu.cn

[1] College of Cyber Security,Jinan University,Guangzhou 510632,China.

Copyright © JES 2024 on-line : journal.esrgroups.org

processing, the complex instruction set CPUs of the x86 architecture use more aggressive multi-issue and out-of-order execution strategies. In the late stages of the pipeline decoding stage, multiple micro-instructions that are ready for execution are simultaneously issued to the execution ports (various ALU units are responsible for the execution of instructions). The executed instructions will be re-sorted in the ROB (reorder buffer) and retired according to the original order of fetching, that is, written to memory. For the program, it is as if the out-of-order execution has never happened.

However, current research has found that out-of-order execution in the pipeline leaves traces in the micro-architecture. These traces can be used to leak data stored in the micro-architecture through techniques such as flush+reload[4-6] and other related methods.

As shown in Figure 1, pipeline out-of-order execution (OOO) is designed to improve instruction-level parallelism and execution efficiency. In traditional processor design, instructions are executed in sequence according to the program order, but due to data dependency and control dependency between instructions, this execution method may lead to a waste of processor resources and reduced efficiency. Therefore, pipeline out-of-order execution technology is introduced to allow the processor to rearrange the instruction execution order without violating program semantics.

The implementation of pipeline out-of-order execution involves complex hardware mechanisms and algorithms, which aim to solve the performance bottlenecks caused by instruction dependencies and resource dependencies. The main technologies include out-of-order execution, instruction renaming, register renaming, branch prediction, and load/store instruction reordering. In pipeline out-of-order execution, the processor improves execution efficiency by reordering instructions without violating program semantics. Out-of-order execution in the CPU requires the combination of the following components to function properly:

Out-of-order execution: The processor can continue to execute instructions that do not depend on the results of certain other instructions while waiting for those results. This allows the processor to better utilize available resources, thereby improving execution efficiency.

Register renaming: The processor maps logical registers in instructions to physical registers to prevent conflicts caused by data dependencies. This prevents pipeline stalls while waiting for the results of certain instructions and allows instructions to be executed out of order.

Branch prediction: The processor predicts the execution path of branch instructions and rolls back if the prediction is incorrect. With accurate branch prediction, the processor can avoid wasting time in the pipeline due to mispredictions.

Retired instruction reordering: When an instruction is retired, the processor reorders the execution order of the instructions so that the program behaves as if the instructions were executed sequentially, despite out-of-order execution.

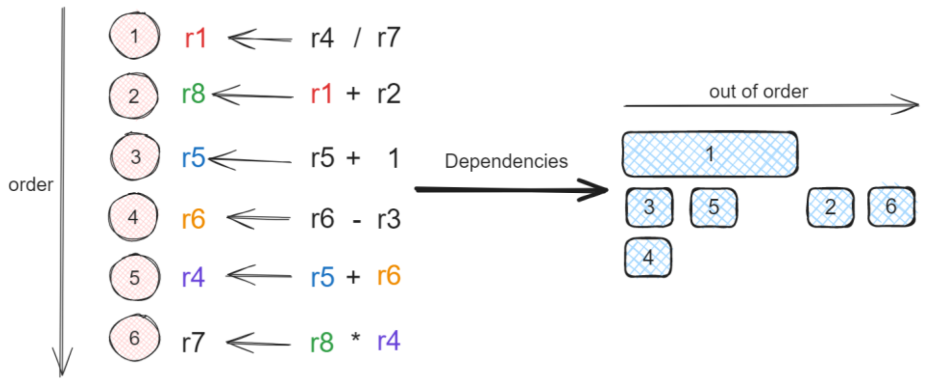

**Figure 1 Pipeline out-of-order execution**

*1) Refresh pipeline time*

As shown in Figure 2, pipeline flush is an important concept in processors, used to handle pipeline interruptions caused by exceptions, branch prediction errors, or other unpredictable events. In a processor, the pipeline executes instructions in parallel by breaking down instruction execution into multiple stages to improve execution efficiency. However, when an exception or branch prediction error occurs, the pipeline needs to be cleared and restored promptly to ensure the correct execution of the program and the consistency of the data.

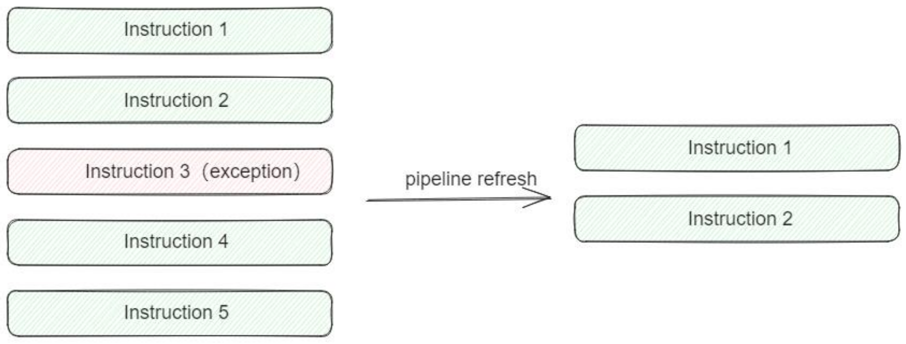

**Figure 2 :After the pipeline is flushed, instructions 3, 4, and 5 are cleared.**

The timing of refreshing the pipeline is a key consideration in processor design. It needs to ensure execution efficiency while minimizing the performance loss caused by refreshing the pipeline. Under normal circumstances, the pipeline will not be refreshed. Pipeline refresh indicates that the execution of an instruction has encountered an abnormal situation (for example, an instruction predicted to be executed is found to be incorrect during the pipeline process and should not be executed at all. In this case, all subsequent instructions in the ROB should be invalidated). In a CPU architecture with a deep pipeline, this can cause significant performance loss. Additionally, the side channel information of the micro-architecture may also be lost during the refresh. Mastering the timing of pipeline refresh is crucial for designing a reasonable side channel instruction sequence. The following outlines the timing of refreshing the pipeline:

When an exception occurs: When the processor detects an exception during instruction execution (such as a division by zero error, memory access out of bounds, etc.), it needs to immediately flush the pipeline, stop the execution of the current instruction, and perform exception handling. After flushing the pipeline, the processor transfers to the exception handler and performs the necessary operations to handle the exception.

When branch prediction is wrong: Branch prediction is a commonly used optimization technique in processors to predict the execution path of branch instructions. When branch prediction is incorrect, meaning the predicted branch path does not match the actual execution path, the pipeline needs to be flushed and re-executed. This prevents incorrect instructions from being executed and ensures the correctness of the program.

When a conditional transfer instruction is executed: When processing a conditional transfer instruction, the processor needs to wait for the result of the conditional judgment and determine the execution path based on that result. If the result of the conditional judgment cannot be determined immediately (for example, it needs to wait for the calculation result of the condition register), the processor may pause the pipeline and wait for that result. After the result of the conditional judgment is available, if the execution path needs to be changed, the processor will flush the pipeline and restart the execution of a new instruction sequence.

When memory access is abnormal: When the processor initiates a memory access operation, access exceptions (such as page faults or access permission exceptions) may occur. When these exceptions occur, the processor needs to immediately stop the execution of the current instruction and flush the pipeline. Then, the processor transfers to the exception handler and executes the corresponding exception handling process.

When instruction conflicts or data dependencies are detected: The processor detects data dependencies between instructions and adjusts the execution order based on these dependencies. When data dependencies are found, the processor may pause the pipeline and wait for the results of related instructions. After the data dependencies are resolved, if some instructions need to be re-executed, the processor will flush the pipeline and restart execution.

The timing of refreshing the pipeline is an issue that needs to be carefully considered in processor design. It also has a significant impact on the experiments in this article. If the pipeline undergoes a refresh operation before the micro-architectural side channel is established, all micro-architectural information will be lost. Therefore, the information needs to be leaked before the pipeline is refreshed.

*2) Cache Attack*

To speed up memory access and address translation, the CPU contains a small memory buffer called a cache, which stores frequently used data[7]. CPU caches hide slower memory access latency by buffering frequently used data in smaller, faster internal memory. Modern CPUs have multiple levels of caches that are either private or shared. Modern processors use a cache hierarchy consisting of multiple caches[8]. For example, the cache hierarchy of the Core i7-6700k processor, shown in Figure 3, consists of three cache levels: L1, L2, and L3.

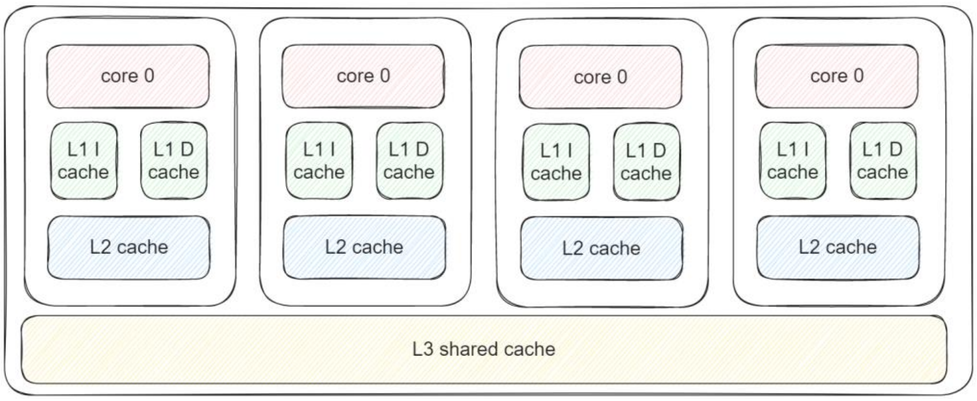

**Figure 3 CPU Cache Structure**

As shown in Figure 4, Flush+Reload is one of the side-channel attack techniques widely used in the field of computer architecture and operating system security[9]. This attack exploits the cache behavior of modern processors and infers the execution status of other processor cores or threads by observing the loading (or flushing) pattern of data in the cache, which may leak sensitive information.

Modern multi-core processors typically share cache resources between multiple cores to improve performance. When one core accesses data in memory, the processor loads that data into the nearest cache level, usually the L1 or L2 cache. If another core also accesses the same data, the processor attempts to provide the data from the shared cache instead of loading it from the slower main memory. This cache sharing mechanism is key to improving the efficiency of multi-threaded and multi-core processors.

The basis of the Flush+Reload attack is the shared nature of the processor cache. An attacker can use this feature to monitor the access pattern of another core or thread to a specific memory address. The attack steps usually include the following stages:

Flush phase: The attacker first needs to ensure that the target data line (i.e. a block of data in the cache) is not already in the processor's cache. This is usually achieved by writing to a special address that triggers the replacement (or "flush") of the cache line.

Reload phase: After the target data line is flushed, the attacker will try to reload the data line into the cache. This can be done by reading or writing the addresses associated with the target data line.

Monitoring phase: The attacker monitors the state of the target data line in the cache. If the target thread or core accesses the memory address associated with the target data line, the data line is loaded back into the cache. The attacker can infer the execution state of the target thread by observing the access pattern (e.g., when the access occurs or whether the access occurs).

Flush+Reload attacks can be used to leak encryption keys, user credentials, or other sensitive information. For example, if an encryption algorithm accesses different memory addresses under different keys, an attacker can infer certain parts of the key by monitoring the access patterns of these addresses.

To defend against Flush+Reload attacks, operating system and hardware manufacturers have taken a variety of measures. One approach is to introduce new hardware features in processors, such as Intel's TSX (Transactional Synchronization Extensions)[10] and SGX (Software Guard Extensions) [11], which are designed to reduce the possibility of cache side-channel attacks. Operating systems can also reduce the risk of attacks through software-level isolation, such as by providing stronger memory isolation between different virtual machines or containers. In addition, researchers and developers are also exploring new programming models and libraries, which can help programmers write more secure code and reduce the risk of side-channel attacks. For example, the use of memory randomization techniques[9, 12] can make it difficult for attackers to predict the address of the target data line, thereby increasing the difficulty of the attack. Despite the existence of these defenses[13, 14], Flush+Reload attacks remain an active research area, with new variants and techniques being proposed. As computing devices become more complex, the challenge of protecting systems from such attacks is also increasing.

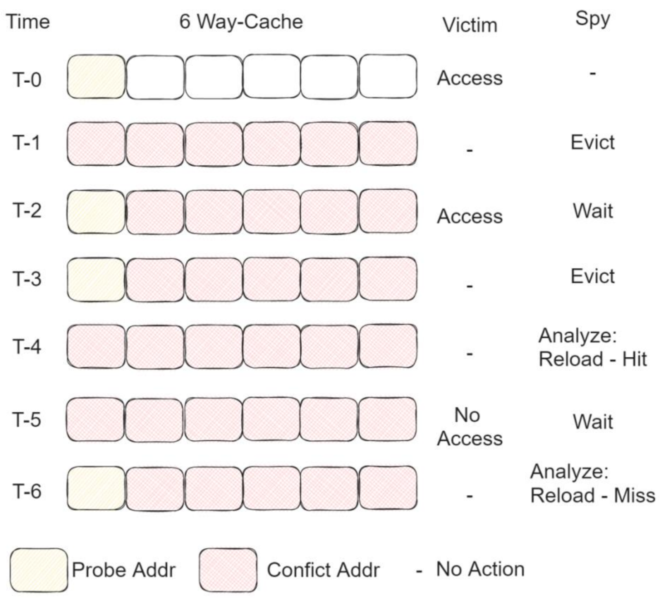

**Figure 4 Flush + Reload attack**

*B. KPTI（Kernel Page Table Isolation）*

The primary software mitigation for Meltdown is to enable KPTI in the kernel. When KPTI is not enabled, the top of the user address space is mapped to the kernel, so transient instructions can successfully access kernel data through the page table, causing side channel attacks. After KPTI is enabled, the page table in user mode only has a small number of address mappings of the entry points of the kernel's key system call functions. Meltdown cannot access any other address space that is missing mapping in the page table, thereby alleviating the impact of meltdown on the kernel (enabling KPTI usually results in a 5%-20% performance loss).

As shown in Figure 5, after the KPTI mechanism is enabled, the page table will be switched when the user state enters the kernel state; when the kernel state is restored to the user state, the page table will also be switched. If the code snippet for switching the page table executed when the kernel returns to the user state can be controlled, it can also return to the user state normally.

KPTI (Kernel Page Table Isolation) is a security mechanism designed to prevent side-channel attacks that leak sensitive information by analyzing the processor's cache behavior. KPTI reduces the risk of kernel data leakage by isolating the page tables of kernel space from the page tables of user space. In modern operating systems, especially the Linux kernel, memory management is a complex process involving a hierarchy of page tables. Page tables are data structures used to map virtual addresses to physical addresses. Without KPTI, the kernel and user space share the same page tables, which means that programs in user space may be able to infer the memory addresses that the kernel is accessing by observing the pattern of kernel accessing memory. This information may include sensitive keys and data, providing a potential entry point for attackers.

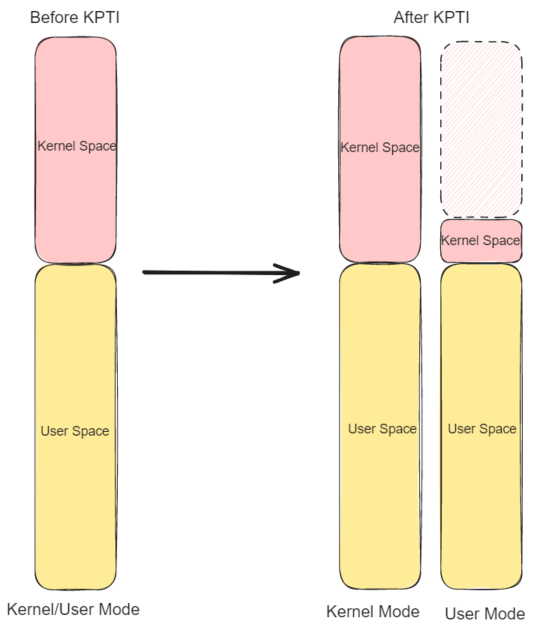

**Figure 5 KPTI strategy**

KPTI works by creating an independent page table for the kernel, which is stored separately from the user space page table. In this way, even if user-space programs can somehow observe the behavior of the cache, they can only see their own page tables and not the kernel's page tables. This isolation significantly increases the difficulty for attackers to obtain sensitive information through side-channel attacks. The implementation of KPTI usually involves a series of modifications in the operating system kernel. First, the operating system must be able to maintain a separate page table for the kernel. This means that the kernel needs to use specialized kernel page tables instead of user space page tables when performing memory management operations. Secondly, the operating system needs to ensure that user space programs cannot access the kernel's page tables under any circumstances.

After enabling KPTI, the performance of the operating system may be affected to some extent. Because the kernel and user space page tables are separate, the processor requires additional context switches between kernel and user space. This can lead to increased memory access latency, especially in scenarios where system calls are made frequently. However, for most users, this performance impact is acceptable because it provides greater security. KPTI was designed in response to a series of side-channel attacks discovered in recent years, such as Specter and Meltdown. These attacks take advantage of the processor's speculative execution characteristics to leak information by analyzing cache access patterns. While KPTI does not protect against all types of side-channel attacks, it provides an important layer of protection that makes it more difficult for attackers to exploit these vulnerabilities. KPTI is an important security feature that significantly improves system security by providing page table isolation between kernel and user space.

*C. CPU privileged mode*

As shown in Figure 6 the CPU has at least three privileges (Ring0 - Ring3). The privileged mode of the CPU refers to the privilege level of the processor when executing instructions, which is usually divided into two modes: user mode and kernel mode. These modes determine the instructions that the processor can execute and the range of resources it can access.

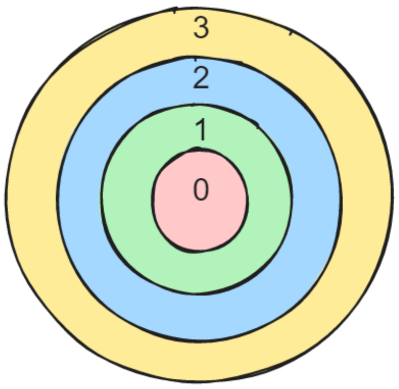

**Figure 6: Privilege level,User programs are located in Ring3, kernel programs are located in Ring0, and the higher the privilege level, the smaller the value.**

User state (Ring 3): User state is a low-privilege level mode of the processor used to execute ordinary user programs. In user state, the processor can only execute a restricted instruction set and has limited access to system resources. Normally, user programs can only access their own address space and cannot directly access the resources of the system kernel or other processes.

Kernel state (Ring 0): Kernel state is a high-privilege level mode of the processor, used to execute the operating system kernel code. In kernel state, the processor has higher permissions, can execute the entire instruction set, and can directly access system resources and hardware devices. The operating system kernel usually runs in kernel state to manage system resources, schedule processes, handle interrupts and other core tasks.

*1) Page table switching timing*

During the system call process, since the user program needs to use the kernel function, it will fall into the kernel state. After the kernel function is executed, it will return to the user program. Therefore, after KPTI is turned on, there will be page table switching before and after the system call. In general, as long as the system function is to be used, it will fall into the kernel state, accompanied by page table switching. General exception handling is also this type of operation. For example, when the program execution encounters a page missing exception, the kernel will intervene and load the missing page into the memory. Page table switching is an important concept in the operating system. It is used to manage the page table structure in the virtual memory system to achieve the mapping of virtual addresses to physical addresses. In a multitasking operating system, when a process switch or address space switch occurs, the page table switching operation is involved. In the user state page table, since there is no kernel page table mapping, it is impossible to obtain kernel data. Understanding the timing of page table switching is also the key to breaking the KPTI mechanism. The following is about the timing of page table switching:

Process switching: In a multitasking operating system, the processor needs to switch between different processes to achieve concurrent execution of multiple tasks. When the operating system scheduler decides to switch to another process, a page table switch is required. This is because different processes have different virtual address spaces and require different page tables to manage address mapping relationships.

Switching from user state to kernel state: When the processor switches from user state to kernel state, a page table switch is usually required. This is because user state and kernel state have different address spaces and require different page tables to manage address mapping relationships. For example, when a process falls into the kernel and executes a system call or interrupt handler, the operating system needs to switch to kernel state and use the page table of the kernel address space.

Interrupt handling: When the processor receives an external interrupt or exception, it usually needs to switch the page table. This is because the interrupt handler may need to access data structures or code in the kernel address space, so it needs to switch to the page table of the kernel address space to implement address mapping.

## III. THE PROPOSED SCHEME

This chapter extends from the micro-architectural side channel of the jump instruction boundary in user space to the micro-architectural side channel of the permission switching boundary in kernel space. This article will start with a simple example to illustrate some problems that may exist at the CPU permission switching boundary. Suppose there is a simple operating system with a user program and a kernel program. The user program wants to send some data to the kernel program for processing through a system call. A common way is to use the sysenter instruction to trigger the switch from user space to kernel space. However, there may be some problems with this switch. For example, the user program may use micro-architectural features such as cache or branch prediction to detect information about kernel program execution, thereby leaking sensitive data. Specifically, when the user program enters the kernel space through the sysenter instruction, if some cache lines are used by the kernel and these cache lines exist in the user program's cache, the user program can infer the execution of the kernel program by monitoring the state changes of the cache lines. In addition, branch prediction may also become a problem. When the user program executes the sysenter instruction, the branch prediction unit may try to predict the next execution path of the kernel program. The user program can infer the execution of the kernel program by observing the behavior of branch prediction, thereby obtaining sensitive information.

### *A. A Toy Example*

As shown in Figure 7, this is a simple instruction execution sequence. From the programmer's perspective, the instructions are executed sequentially as shown on the left side of the figure. However, the actual execution in the superscalar CPU pipeline is often not necessarily the case[15].

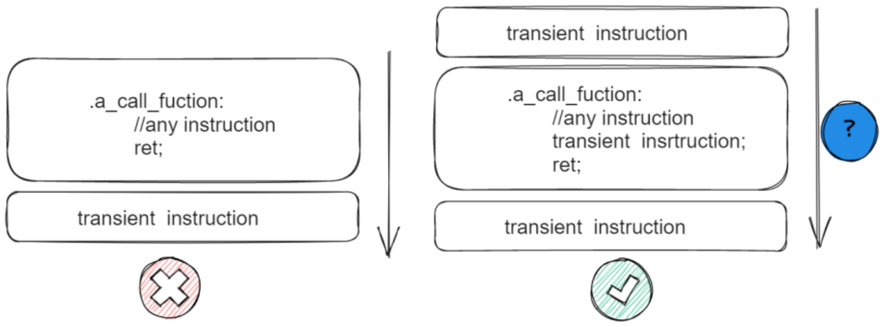

**Figure 7 The execution order of transient instructions in a CPU pipeline is indeterminate**

In this preliminary example, the instantaneous instruction is not a complete instruction like an assembly instruction inside the CPU. In the complex instruction set architecture of x86, an assembly instruction is often decoded into several micro-codes with very short execution cycles[16]. When all the micro-codes decoded from an assembly instruction are executed, the assembly instruction is considered to have completed execution. Figure 7 only shows the possible logical position of the assembly instruction when it is completed.

Since the processor adopts an out-of-order execution strategy, it can be known that if the instruction meets the conditions for instruction issuance before its previous instruction, then this instruction will be issued first[17]. If some memory access instructions are arranged in the far jump function at this time, since memory access instructions require multiple instruction cycles, the instantaneous instruction can easily be executed before some instructions in the function. At this time, the execution context of the instantaneous instruction is the environment in which the function is executed.

The formal logic of transient instructions used to leak micro-architectural information has a fixed strategy, as shown in Algorithm 1. The first instruction is an illegal instruction. It loads the value at any address that the attacker wants to access out of bounds into the register. Subsequent instructions use the value of the register to index an array and load the data at the corresponding position into the CPU cache. Since the location of the data cache is linearly related to the index of the data, this will lead to the deduction of the array index that has just been accessed from the location that can be hit in the CPU cache, thereby deducing the value of the register. This value is the data that the attacker wants to access out of bounds, and the current cache location of the data can be obtained by flush+reload. At this point, the scheme can leak micro-architectural data.

It should be noted that even if the first instruction generates an exception, the subsequent instructions can still be executed instantaneously at the micro-architecture level. This is due to the aggressive multi-issue strategy of the CPU pipeline, instruction reordering cache, and delayed permission check[1]. All of them are indispensable. The multi-issue strategy of the CPU pipeline requires that the instruction fetching component fetches values each time to meet the requirements of the maximum number of concurrent instructions of the CPU pipeline. Otherwise, multiple components of the CPU will have no instructions to process, wasting processor performance. At the same time, although the CPU fetches and executes multiple instructions each time, when each instruction exits the pipeline, it will affect the memory components. They will write the execution results of the instructions in the order in which the instructions were fetched. In this way, the out-of-order execution of instructions is the same as the sequential execution in the programmer's view. The last point to be pointed out is that the processor will not stop executing when it detects that the instruction is illegal during execution, but will continue to execute the instruction. However, when saving the execution result of the instruction, the execution result exception flag of the instruction is set to 1. In this way, all executed instructions will definitely affect the micro-architecture state. At this time, when the micro-architecture state at this time is leaked using the side channel, any data in the system can be obtained.

**Algorithm 1 Disordered execution to get kernel data instruction sequence**

```
input:target kernel-space address(target_addr)
output:target kernel-space address data
1:    mov [target_addr] to target_reg
2:    //The first instruction causes an exception to be thrown
3:    access( probe_array[ target_reg * 4096])
4:    flush and reload the whole probe_array
```

## *B. Micro-architectural side channels at the user-mode to kernel-mode boundary*

As we know out-of-order execution is possible at the function boundary in user mode. When an instruction outside a function exists near the function, the instruction outside the function may be executed before the function is executed or in the function at the micro-architecture level, and the micro-architecture information will not be cleared after returning. If the function is a function in kernel mode, does it mean that the instantaneous instruction can be executed out of order in the kernel function, and the system information is stored in the micro-architecture and not cleared? Since the kernel function belongs to the kernel mode, the corresponding CPU permission is Ring0, and the instantaneous instruction execution environment has also become the kernel mode environment, and the kernel page table can be used.

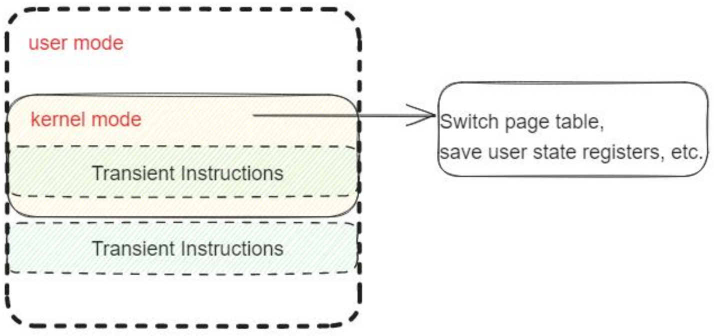

**Figure 8:User-mode code is executed out of order in kernel mode**

As shown in Figure 8, out-of-order execution of user-mode code at the permission switching boundary is the key to breaking the KPTI mechanism. The principle of the KPTI mechanism is to minimize the overlap between the kernel address space and the user address space. As shown in Figure 9, in an operating system without KPTI, since the high address part of the user-mode page table is the mapping of the kernel space, although the program in user mode cannot directly access the high address space, it can always use the micro-architecture covert channel of out-of-order execution instructions to get the data in the high address space, that is, to get the kernel data. However, after KPTI is enabled, since there are only a few necessary entry address mappings of the kernel system call function interface in the user page table at this time, other kernel address mappings do not exist at all, that is, the page table P bit of most high-order address spaces is 0. When translating virtual addresses through the memory management unit (MMU, Memory Manage Unit), the page table is an inaccessible translation medium. In the absence of page table mapping, the target data cannot be accessed even through the side channel of the micro-architecture.

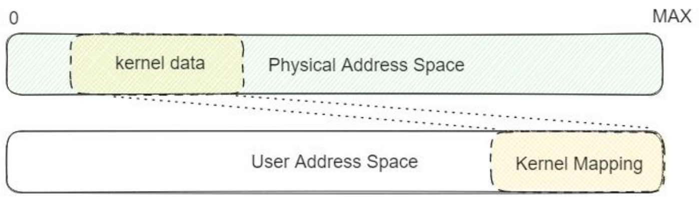

**Figure 9:The kernel is mapped to the high address of user space**

When a user-mode instruction is executed out of order in kernel mode, the execution environment of the instruction is also the kernel-mode environment. Under the KPTI mechanism, the page table has been switched to the kernel's page table. At this time, through the micro-architecture covert channel, it is undoubtedly possible to obtain data from any kernel.

However, in the specific implementation, this article takes into account more details:

First, consider the depth of the CPU pipeline. When the instantaneous instruction is too far away from the kernel state instruction, the instantaneous instruction may not be in the pipeline at the same time as the instruction in the kernel state environment, resulting in the inability to use the kernel state environment. Therefore, the number of instructions between the kernel state environment and the instantaneous instruction must be reduced as much as possible. At the same time, due to the limitation of the pipeline depth, the number of instantaneous instructions also needs to be reduced as much as possible. On this basis, experiments need to be conducted for different instruction intervals to achieve the best leakage effect.

The second is to consider the location of transient instructions, which refers to two aspects. First, we should know that there are three ways to enter the kernel state: system calls, interrupts, and exceptions. Interrupts are generally used by the system to respond to external events, such as I/O devices such as keyboards and mice. In the x86-32 architecture, int 0x80 soft interrupts were used to respond to system calls. However, in the x86-64 architecture, this system call method with huge overhead has been replaced by sysenter and other instructions[18]. At this time, it is no longer necessary to find the location of the system call routine by searching the linked list in the memory. x86-64 provides dedicated model-specific registers (MSR) to save the system call entry address. The exception handling routine exists in the kernel, and the calling process is too complicated to inject transient instructions into it. Therefore, interrupts and exceptions cannot be combined with transient instructions to form a micro-architecture side channel. Secondly, we need to place transient instructions after the system call, because during the CPU instruction fetching process, the data of the entire cache line size is usually fetched into the L1 instruction cache according to the address. If it is placed before the system call, it may not be fetched at all.

### *C. A holistic solution model to break through the KPTI mechanism*

This section introduces the solution of micro-architectural side channel to break through KPTI restrictions in three stages: (1) construction stage; (2) waiting stage; and (3) data collection stage. This section focuses on the construction stage, focusing on the relative position of system calls and instantaneous instructions, exception handling, and timing difference collection.

#### *1) Side channel construction scheme*

In this scheme, the flow direction of secret data can be clearly seen. In this way, secret data will no longer be secure. As shown in Figure 10, this is an overall scheme that uses the out-of-order execution of the CPU privilege switching boundary to break through the KPTI mechanism. The transient instruction uses the kernel page table to load the content of the inaccessible kernel location into a register and access the kernel data in the micro-architecture. The transient instruction accesses the cache line based on the secret value of the register. The attacker uses Flush+Reload to determine the cached cache line and thus the secret stored in the selected memory location. At this time, the kernel data has been restored through the covert channel and finally the kernel data is obtained. In the attacker scenario, the attacker can execute arbitrary user-mode code on the attacked system, that is, the attacker can run any code with the privileges of an ordinary user. In addition, it is assumed that the system has enabled protection strategies such as SMAP[19], SMEP[20], NX[21], KPTI, and KASRL[22].

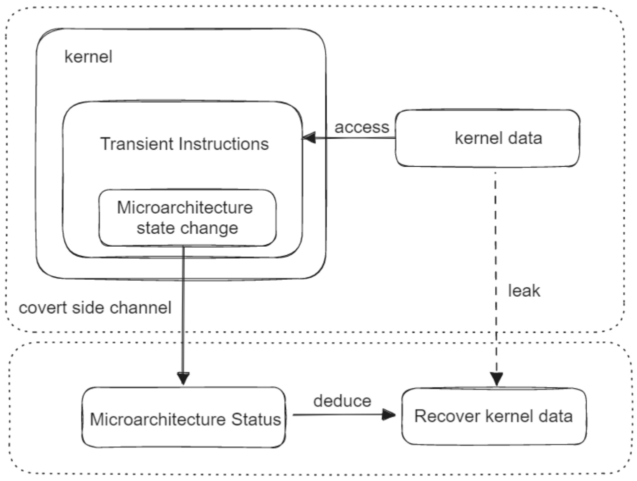

**Figure 10:Overall Program Architecture**

#### *2) Build Phase*

As shown in Algorithm 2, this article will construct a scenario where a transient instruction is adjacent to the kernel code. In the analysis in Section 3.3, the system call is generally selected as an example of the kernel code. Considering that the amount of code between the system call from the library function to the actual system call instruction sysenter instruction in the actual Linux kernel is not small, it is necessary to add a custom system call and place the transient instruction after the sysexit instruction of the system call. At this time, it is also necessary to add a global variable accessible to the user state to pass parameters to the transient instruction, as well as a global array accessible to the user state to act as a carrier for flush + reload.

Unlike general meltdown attacks, transient instructions near kernel-mode code are not in the execution flow, so they do not actually change the memory or affect the system state. They only leave traces on the micro-architecture, so they will not trigger segmentation fault exceptions, and therefore there is no need to handle the exceptions.

**Algorithm 2 out of order execution to Get Kernel Data**

```
Input:Target kernel space address target_addr
Output: target kernel address data
1:    //Save user-state registers and other operations
2:    //Set up the passing parameter registers
3:    sysenter
4:    //Any code sequences in kernel state
5:    sysexit
6:    mov [target_addr] to target_reg
7:    access( probe_array[ target_reg * 4096])
8:    flush and reload the whole probe_array
```

*3) Waiting phase*

At this stage, the attacker continuously executes customized system calls. Due to the optimization of system calls by x86-64 system, the code of the system call itself does not cause much interference. At this time, it is necessary to wait for the completion of the system call. Each completion is an access to kernel data, and the micro-architecture state will make corresponding changes.

*4) Data Collection Phase*

As discussed previously, we exploit the flush+reload attack to obtain a high-bandwidth and low-noise covert channel constructed by the processor cache[23], based on which the transient instruction sequence must encode the kernel secret data into the micro-architecture cache lines. A probe array is pre-allocated in memory and ensured to be completely uncached. To transfer the kernel secret data, the transient instruction sequence contains an indirect memory access to the probe array, the address of which is calculated based on the kernel secret data that is otherwise inaccessible. In line 6 of Algorithm 2, the kernel secret value in line 5 is multiplied by a regular page size (i.e., 4 KB). This multiplication ensures that different index accesses to the array have a large spatial distance from each other, which prevents the cache lines corresponding to different secret data from being loaded duplicated. Therefore, the final probe array is 256×4096 bytes.

Since line 6 of Algorithm 2 is a memory access instruction sequence that requires more clock cycles, reducing the execution time of this line of instructions can improve the efficiency of the attack. The optimization method is to cache the address of the probe array in the TLB in advance. In this way, when the MMU translates the probe address, it only needs to access the TLB, without querying the page table and actually accessing the memory, which can improve the attack performance on some systems[24].

The detection of the data stored in each address should be measured at least 100 times, and each detection should be repeated at least 100 times. The results of each judgment should be collected, and the index value with the most accumulated hits should be used as the secret data. In this way, the data of the entire kernel can be completely restored. In order to maximize the data acquisition bandwidth, the experiment can be repeated to reasonably reduce the number of loops of some codes.

## IV. Result

This section of the experimental analysis will evaluate the actual performance of the out-of-order execution concept verification at the processor privilege switching boundary. Table 1 shows the configuration list of the experiment. During the verification process, this article tested 3 local machines and 2 cloud servers with Intel x86-64 processor architectures. Under the Linux 4.15 kernel, the KASLR policy was not enabled. After the KPTI policy was enabled, the kernel read verification, the side channel of the privilege switching boundary successfully read the user and kernel data.

**Table 1 experimental settings**

| environment | Kernel version | CPU model | Number of cores | Cache line size |
|---|---|---|---|---|
| local | Linux 4.15 | i7-6700K | 4 | 64（byte） |
| local | Linux 4.15 | i7-6600U | 2 | 64（byte） |
| local | Linux 4.15 | i5-7500 | 4 | 64（byte） |
| cloud | Linux 4.15 | Xeon E5-2676 v3 | 12 | 64（byte） |
| cloud | Linux 4.15 | Xeon E5-2650 v4 | 12 | 64（byte） |

## A. *User state read user state data verification*

This is a most basic verification experiment. At the beginning, this article did not directly target the kernel permission switching boundary. If there is no way to build an out-of-order execution side channel at the function boundary in user mode, then it is impossible to obtain kernel data.

In user mode, this experiment uses the side channel to obtain data from other arbitrary addresses in the current user space. Because illegal addresses may be accessed (for example, some page tables do not exist), the TSX instruction set is needed to eliminate the impact of exceptions on the program. It should be noted that the TSX instruction set is generally not enabled in cloud servers. In this case, this article adopts the general method of capturing exceptions and jumping to the beginning of the code to reset the process state. As shown in Table 2, this article obtained a read rate of more than 540k/s in the experiment, which can detect any data in the entire user process address space of the program. At the same time, when each secret value is measured 10,000 times, the error rate of each secret value is very low without causing a crash.

**Table 2 user state data leakage rate**

| CPU model | target space | data leakage rate | error rate |
|---|---|---|---|
| i5-7500 | user space | 712kb/s | 0.01% |
| i7-6700k | user space | 620kb/s | 0.04% |
| i5-6600U | user space | 340kb/s | 0.02% |
| Xeon E5-2676 v3 | user space | 650kb/s | 0.05% |
| Xeon E5-2650 v4 | user space | 540kb/s | 0.05% |

## B. *Verification of kernel state data read from user state*

This experiment finally tests the side channel of the privileged mode switch boundary on multiple versions of the Linux kernel (from version 4.13 to version 5.13). On all these versions of the Linux kernel, the virtual address of the kernel base address is fixed without KASLR, so the entire kernel space can be leaked through the privileged switch boundary side channel attack. Before kernel 4.12, kernel address space layout randomization (KASLR) was turned off by default. If the KASLR policy is turned on, we can still find the kernel by searching the address space. Because the actual address randomization policy only targets the lower 5 bytes of the address, our entire search space is not very large. If KASLR is not turned on, the physical kernel base address is fixed from 0xffff8800000000000 and linearly maps the entire physical memory. On such a system, an attacker can read the data of the entire kernel address space starting from 0xffff8800000000000. In this case, as shown in Table 3, the kernel data leakage rate reaches at least 210kb/s. In order to obtain the total number of measurement errors for each secret data, a record array is set up in this paper, and all measurement results are stored in the array. Regardless of whether the result is correct or not, the error rate of incorrect results can be obtained through this array. The setting of this array is necessary because multiple measurements must be performed to find the result with the most identical measurements for each secret value as the predicted result.

**Table 3 kernel state data leakage rate**

| CPU model | target space | data leakage rate | error rate |
|---|---|---|---|
| i5-7500 | Kernel space | 210kb/s | 0.02% |
| i7-6700K | Kernel space | 347kb/s | 0.03% |
| i5-6600U | Kernel space | 324kb/s | 0.05% |
| Xeon E5-2676 v3 | Kernel space | 417kb/s | 0.07% |
| Xeon E5-2650 v4 | Kernel space | 433kb/s | 0.08% |

In order to evaluate the impact of the kernel version on the side channels, as shown in Table 4, this experiment tried 4 different Linux versions for the same processor. The multitasking environment has a significant impact on the effectiveness of the attack, but in most cases we can get the expected data (if the target kernel data is loaded in the cache). In this case, this article only targets one processor Intel Core The i7-6700K conducted kernel version experiments because the same kernel has little impact on x86 processors of the same architecture. When using a lower kernel version, Linux 4.13, the average read speed was 380 kb/s, the fastest among all versions. As the version increases, the leak rate slows down. There are many possibilities for the reason here (kernel Instructions such as clfluch have been safely processed or system call boundary instructions have changed). There is an optimization method that can be used to increase the kernel leak rate: running a program synchronously and accessing the addresses of locations around the kernel target memory, placing the kernel secret data in the cache in advance. There are also optimization measures for side channel noise. Since the code in the kernel space of the system call is fixed, the noise pattern generated by it is also fixed. Then the noise pattern caused by the system call can be recorded and the results obtained Subtracting this noise is the result of eliminating system call noise.

**Table 4 Effect of kernel version on kernel leakage rate**

| CPU model | target space | data leakage rate | error rate |
|---|---|---|---|
| i7-6700K | Linux 4.13 | 380kb/s | 0.02% |
| i7-6700K | Linux 4.15 | 347kb/s | 0.03% |
| i7-6700K | Linux 5.3 | 324kb/s | 0.05% |
| i7-6700K | Linux 5.13 | 214kb/s | 0.07% |

This paper basically adopts an experimental configuration similar to the meltdown paper for experimental comparison. As shown in Table 5, under the same processor configuration, compared with the meltdown experimental results, the rate of user-mode data leakage in this paper's scheme is not much improved, and is only 51kb/s higher than the leakage rate of the meltdown scheme. When KPTI is turned on, the meltdown attack scheme has become invalid.

**Table 5 Comparison of experimental results with the meltdown scheme**

| scheme | CPU model | goal | leakage rate |
|---|---|---|---|
| Meltdown | i7-6700K | leakage of user data | 569kb/s |
| My paper | i7-6700K | leakage of user data | 620kb/s |
| Meltdown | i7-6700K | leakage of kernel data | 0kb/s |
| My paper | i7-6700K | leakage of kernel data | 347kb/s |

## V. Conclusions

For the first time, we combined the focus of cache side channels with the CPU privilege switching boundary theory, creating a new perspective on common CPU vulnerabilities, which has a significant impact both in the field of security research and CPU design.

Out-of-order execution based on the permission switching boundary successfully bypassed the KPTI defense strategy, allowing us to obtain arbitrary kernel data at high speed. KPTI is a technology commonly used by current operating systems to defend against kernel vulnerabilities. It prevents attackers from accessing kernel data by isolating the kernel address space from the user address space. Our research shows that attackers can use out-of-order execution to execute malicious code at the permission switching boundary, thereby bypassing the protection of KPTI and obtaining arbitrary kernel data. This is a huge achievement for operating system vulnerability exploitation. It shows that the existing KPTI defense strategy is not completely reliable, and attackers can still bypass its protection through carefully designed attacks. This will prompt operating system vendors to further improve the KPTI defense strategy and enhance the security of the system.

**Data Sharing Agreement**

The datasets used and analyzed during the current study are available from the corresponding author on reasonabl e request.

**Competing Interests**

The authors have no relevant financial or non-financial interests to disclose.

**Acknowledgment**

This article was supported by the National Natural Science Foundation of China under Grant 62272199.

## References

[1]. Ramamoorthy, C.V. and H.F. Li, Pipeline architecture. ACM Computing Surveys (CSUR), 1977. 9(1): p. 61-102.

[2]. Hartstein, A. and T.R. Puzak, The optimum pipeline depth for a microprocessor. ACM Sigarch Computer Architecture News, 2002. 30(2): p. 7-13.

[3]. Müller, L., Kpti a mitigation method against meltdown. Advanced Microkernel Operating Systems, 2018: p. 41.

[4]. Yarom, Y. and K. Falkner. {FLUSH+ RELOAD}: A high resolution, low noise, l3 cache {Side-Channel} attack. in 23rd USENIX security symposium (USENIX security 14). 2014.

[5]. Gruss, D., et al. Flush+ Flush: a fast and stealthy cache attack. in Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13. 2016: Springer.

[6]. Liu, F., et al. Last-level cache side-channel attacks are practical. in 2015 IEEE symposium on security and privacy. 2015: IEEE.

[7]. Smith, A.J., Line (block) size choice for CPU cache memories. IEEE transactions on computers, 1987. 100(9): p. 1063-1075.

[8]. Przybylski, S., M. Horowitz and J. Hennessy, Characteristics of performance-optimal multi-level cache hierarchies. ACM SIGARCH Computer Architecture News, 1989. 17(3): p. 114-121.

[9]. Randolph, M. and W. Diehl, Power side-channel attack analysis: A review of 20 years of study for the layman. Cryptography, 2020. 4(2): p. 15.

[10]. Diegues, N. and P. Romano. {Self-Tuning} Intel Transactional Synchronization Extensions. in 11th International Conference on Autonomic Computing (ICAC 14). 2014.

[11]. Costan, V. and S. Devadas, Intel SGX explained. Cryptology ePrint Archive, 2016.

[12]. Gruss, D., et al. Kaslr is dead: long live kaslr. in Engineering Secure Software and Systems: 9th International Symposium, ESSoS 2017, Bonn, Germany, July 3-5, 2017, Proceedings 9. 2017: Springer.

[13]. Ge, J., et al. More secure collaborative apis resistant to flush+ reload and flush+ flush attacks on armv8-a. in 2019 26th Asia-Pacific Software Engineering Conference (APSEC). 2019: IEEE.

[14]. He, M., et al. Flush-Detector: More Secure API Resistant to Flush-Based Spectre Attacks on ARM Cortex-A9. in 2020

IEEE Symposium on Computers and Communications (ISCC). 2020: IEEE.

[15]. Dodiu, E. and V.G. Gaitan. Custom designed CPU architecture based on a hardware scheduler and independent pipeline registers—Concept and theory of operation. in 2012 IEEE International Conference on Electro/Information Technology. 2012: IEEE.

[16]. Hu, S., et al. An approach for implementing efficient superscalar CISC processors. in The Twelfth International Symposium on High-Performance Computer Architecture, 2006. 2006: IEEE.

[17]. Peleg, A. and U. Weister. Future trends in microprocessors: out-of-order execution, speculative branching and their CISC performance potential. in 17th Convention of Electrical and Electronics Engineers in Israel. 1991: IEEE.

[18]. Forrest, S., S. Hofmeyr and A. Somayaji. The evolution of system-call monitoring. in 2008 annual computer security applications conference (acsac). 2008: IEEE.

[19]. Elwell, J., et al., Rethinking memory permissions for protection against cross-layer attacks. ACM Transactions on Architecture and Code Optimization (TACO), 2015. 12(4): p. 1-27.

[20]. Wu, Y., et al. Evaluation on the security of commercial cloud container services. in Information Security: 23rd International Conference, ISC 2020, Bali, Indonesia, December 16–18, 2020, Proceedings 23. 2020: Springer.

[21]. Kuznetsov, D. and A. Morrison. Privbox: Faster system calls through sandboxed privileged execution. in 2022 USENIX Annual Technical Conference (USENIX ATC 22). 2022.

[22]. Canella, C., et al. KASLR: Break it, fix it, repeat. in Proceedings of the 15th ACM Asia Conference on Computer and Communications Security. 2020.

[23]. Yarom, Y. and N. Benger, Recovering OpenSSL ECDSA nonces using the FLUSH+ RELOAD cache side-channel attack. Cryptology ePrint Archive, 2014.

[24]. Margaritov, A., et al. Virtual address translation via learned page table indexes. in Workshop on ML for Systems at NeurIPS. 2018.