Side-Channel Attack on Kernel Space ASLR

Eldon Ng Kai Foo
National University of Singapore
A0154997H
e0031134@u.nus.edu

Li Yunfan
National University of Singapore
A0162493B
e0134080@u.nus.edu

Low Tian Wei
National University of Singapore
A0139329X
e0003214@u.nus.edu

Yeo Zhuan Yu
National University of Singapore
A0168723X
e0177156@u.nus.edu

ABSTRACT

In order to prevent attacks where attackers know the specific address of a target code or data at runtime, Address Space Layout Randomization (ASLR) is implemented on most modern computers nowadays. ASLR randomizes the address space of the program on runtime, causing the address space to be unknown to the attacker. However, this method of obscuring the address space may be susceptible to side-channel attacks, allowing an attack to identify the address space layout of a computer at runtime.

In this paper, we explored a type a side-channel timing attack called the Double Page Fault attack, as mentioned in [1], an Intel TSX variant of the attack, in order to uncover the kernel space ASLR. We developed a proof-of-concept of the attack. We demonstrated the proof-of-concept attack on an Ubuntu 18.04 Operating System, and managed to uncover the kernel space address layout, successfully derandomizing the kernel ASLR.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Breaking kernel space address space layout randomization.

General Terms

Security

Keywords

Address Space Layout Randomization, Timing Attacks, Side-Channel Attacks, Kernel Vulnerabilities

1. INTRODUCTION

Most modern devices and operating systems utilise ASLR to randomize its address layout, adding a layer of obfuscation in order to prevent attacks such as buffer overflow attacks and many other memory corruption attacks. ASLR works by randomizing both the user and kernel address space layout, such that the components of the program, such as the stack, heap and its libraries are different every runtime. As such, it will be difficult for attackers to perform their exploit, as they do not know the address of a target program until it is being run.

However, there have been workarounds in order to bypass ASLR, such as a prefetch side-channel attack [7], branch target buffer attack [8], rendering ASLR useless against these attacks. .

With the workarounds mentioned above, we want to find out if such side-channel attacks still work on ASLR. To do so, we decided to look into them and understand how they work.

Hence, we explored a type of side-channel timing attack called the Double Page Fault attack, and an Intel TSX variant of the double page fault attack. In this paper, we created a proof-of-concept code for both attacks mentioned above, and conducted the attack on several Linux operating systems and Virtual Machines to evaluate the effectiveness of the attack. Finally, we discuss mitigation and limitations of the aforementioned attack.

2. TECHNICAL BACKGROUND

In this section, we will discuss about some relevant background information about the attack, specifically Address Space Layout Randomization, the Translation Lookaside Buffer and the Intel Transactional Synchronization Extensions.

2.1 Address Space Layout Randomization

ASLR is a memory-protection process for operating systems that guards against buffer-overflow attacks. It helps to ensure that the memory addresses associated with running processes on systems are not predictable, thus flaws or vulnerabilities associated with these processes will be more difficult to exploit. The entirety of the address space layout that remains unknown to the attacker will determine the effectiveness of ASLR.

The Linux kernel has its own implementation of ASLR, called the KASLR. It allows the kernel to be decompressed into a random address. The Linux kernel starts of by doing a page table initialization in order to place the randomized kernel load address afterwards. Then, it uses the chose_random_location() function to generate the random addresses. This function takes in five parameters, input_data, input_len, output, output_len and virtual_addr. input_data is a pointer that points to the compressed kernel image, input_len is the length of the compressed image, output is a pointer that points to the decompressed kernel image,

output_len is the length of the decompressed image, and *virtual_addr* is the virtual address of the kernel load address.

To randomize physical address, it finds all suitable memory ranges with fully accessible memory to load kernel. The kernel will randomly choose a memory location from above to decompress the kernel. For randomization of virtual address, it calculates the amount of virtual memory ranges required to hold the kernel image and uses the same method to generate random physical addresses.

1.1 Translation Lookaside Buffer (TLB)

Translation Lookaside Buffer is a memory cache that stores the most recent translations of virtual memory to physical addresses for faster retrieval.

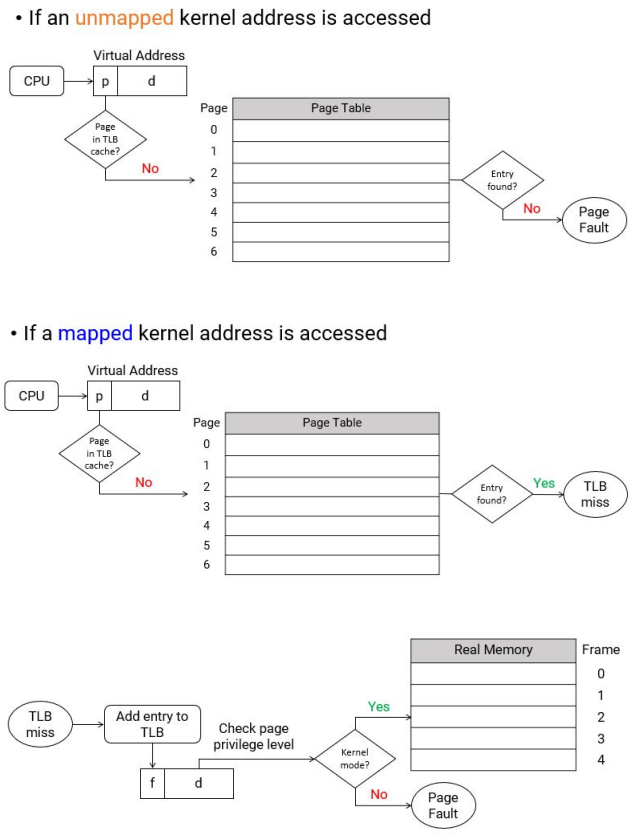


Figure 1: Translating Virtual Address

With reference to Figure 1, the CPU starts a search when the virtual memory address is referenced by a program. The instructional caches are checked. If the required memory is not in these very fast caches, the system has to look up the memory's physical address. At this point, TLB is checked for a quick reference to the location in physical memory. When an address is searched in the TLB and not found, the physical memory must be searched with a memory page crawl operation. As virtual memory addresses are translated, values referenced are added to TLB. When a value can be retrieved from TLB, speed is enhanced because the memory address is stored in the TLB on processor. Most processors include TLBs to increase the speed of virtual memory operations through the inherent latency-reducing proximity as well as the high-running frequencies of current CPU's.

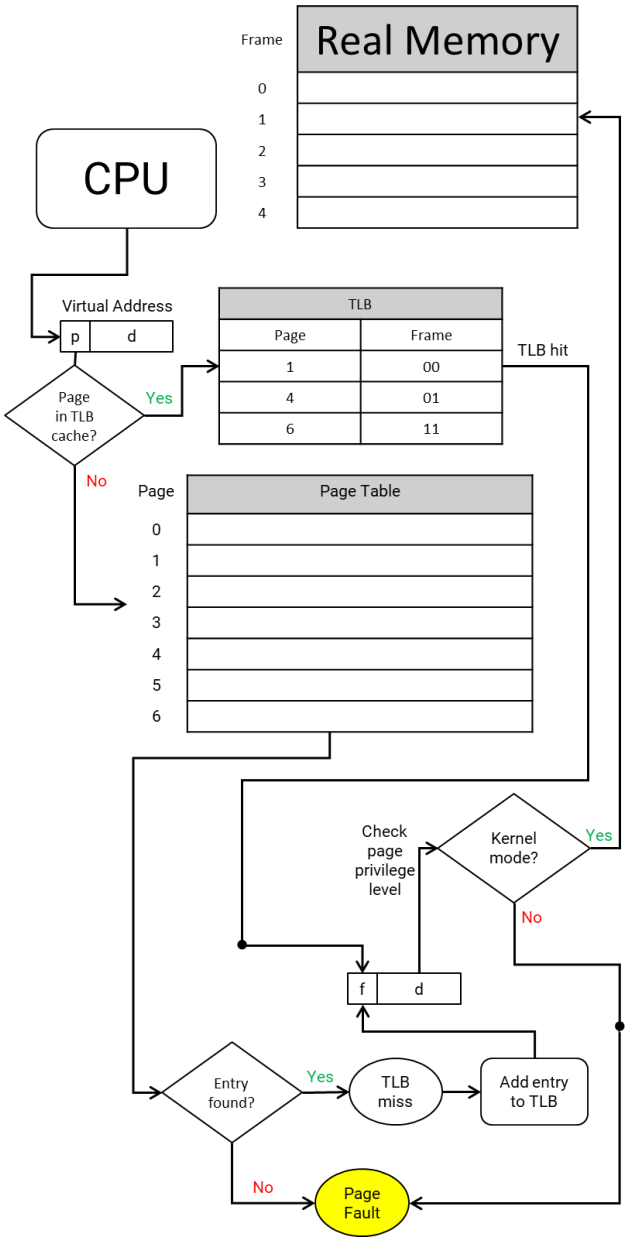


Figure 2: Overview With TLB

If the requested address is present in the TLB, the CAM search yields a match quickly and the retrieved physical address can be used to access memory. This is a page hit. If the requested address is not in the TLB, the CPU checks the page table for page table entry. If the present bit is not set, that implies that the desired page is not in the main memory, and a page fault will be issued. When a page fault occurs, an exception will be raised for handling. Handling page faults usually involves bringing the requested data into physical memory, setting up a page table entry to map the faulting virtual address to the correct physical address, and resuming the program.

1.2 Intel Transactional Synchronization Extensions (TSX)

Intel Transactional Synchronization Extensions is an extension to the x86 instruction set introduced in June 2013. The Intel TSX Extensions were featured in the Haswell Intel Processors onwards.

Intel TSX [6] allows the processor to only perform serialization when it is necessary. It performs this check dynamically at runtime and is able to discover and exploit concurrency in running applications, in order to improve performance.

In Intel TSX, instructions are executed inside transactions. If a transactional execution is successful, the memory operations will seem to be executed instantaneously. However, if a transactional execution fails, the CPU will rollback the execution, discarding the changes from the execution and reverts itself back to its previous state and continue with non-transactional execution.

Intel TSX provided two new software interfaces in order to perform the transactional executions. They were Hardware Lock Elision (HLE) and Restricted Transactional Memory (RTM).

HLE is a legacy-compatible instruction set extension. It introduces two new prefixes, the XACQUIRE and XRELEASE. The two prefixes are able to perform execution in a critical section without acquiring the lock in order to perform transactional execution in advance.

RTM is an instruction set interface that can be used to define more flexible transactional executions. It introduces three new prefixes, XBEGIN, XEND and XABORT. The XBEGIN and XEND prefix signifies the start and end of a transactional execution, whereas the XABORT prefix aborts a transactional execution. Upon the failure of a transactional execution, the processor will rollback to the XBEGIN instruction.

With the new features, Intel TSX is able to boost CPU processing speeds of up to 40%. However, bugs were soon discovered in the implementation of the Intel TSX implementation in 2014, and the TSX features on the Haswell and Broadwell CPUs to be disabled by microcode updates.

2. IMPLEMENTATION

In this section, we will discuss about the implementation of the attack. Firstly, we will mention about the original double page fault attack, followed by the TSX variant of the attack, and provide the following proof-of-concept for the attack to work.

2.1 Double Page Fault attack

Double Page Fault attack allows the reconstruction of allocation of the entire kernel space from user mode. To achieve this goal, we take advantage of the behaviour of the TLB cache. When referring to an allocated page, it means that a page that can be accessed without producing an address translation failure in the Memory Management Unit (MMU). Due to TLB misses, every time a memory access results in a successful page walk, the MMU replaces an existing TLB entry with the translation result. Accesses to non-allocated virtual pages will induce a page fault and the TLB entry is not created. When page translation is successful, but the access permission check fails, the MMU creates a TLB entry.

This behaviour can be exploited to reconstruct the entire kernel space from user mode by accessing each kernel space twice and measuring the access time. The first traversal involves all the kernel pages. For each kernel space page, we first access it from user mode. This results in a page fault that is handled by the operating system and forwarded to the exception handler of the process. The kernel page can refer to an allocated page, which implies that the translation is successful and a TLB entry is created for the kernel page even though the succeeding permission check fails. The other scenario would be that the translation fails, no TLB entry was created by the MMU for that particular kernel page. Hence, for the first traversal, we traversal through the entire kernel space in order to ensure that all kernel mapped pages have a TLB entry, so that we can differentiate between the two scenarios for all the kernel pages in the kernel space for subsequent traversals.

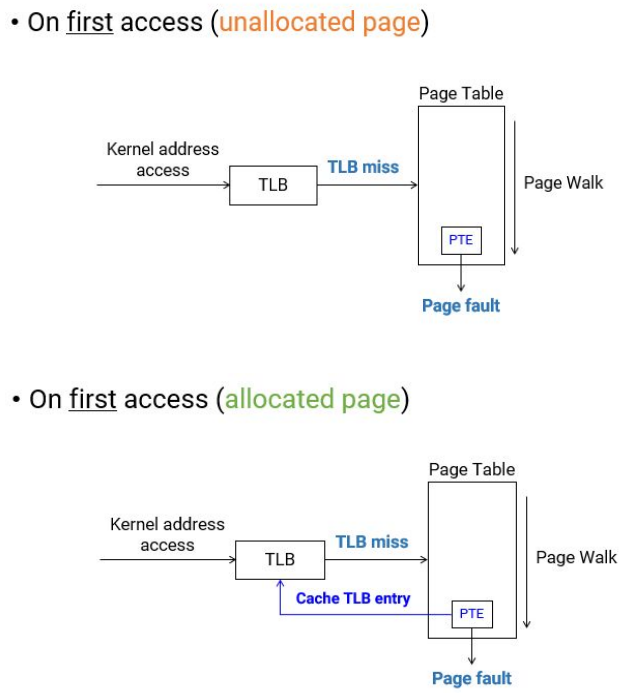
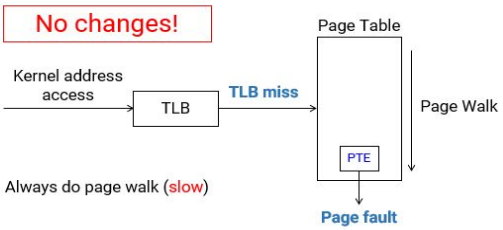


Figure 3: Double Page Fault First Access

For the second traversal, directly after the first page fault, the kernel page is accessed again and the time duration until the second page fault is delivered to the process' exception handler is measured. This time, if the kernel page refers to an allocated kernel page (first scenario), then the page fault will be delivered faster due to the inherent TLB hit due to the existing TLB entry that has been created during the first traversal. Otherwise, it is an unallocated kernel page and there will be no TLB entries in the MMU, and the page fault will be delivered at a later time as it has to go through the physical memory.

The timing difference between the two different types of page fault for allocated and unallocated pages allows an attacker to identify the kernel address space, successfully derandomizing the kernel ASLR.

• On second access (unallocated page)



• On second access (allocated page)

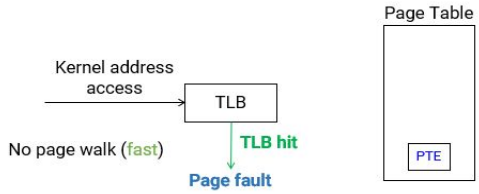


Figure 4: Double Page Fault Second Access

2.2 TSX Variant Attack

Despite the benefits Intel TSX offers, there is a security loophole in the RTM implementation. When the TSX instruction fails due to a critical error, such as a page fault or memory access violation, it does not inform the kernel, and does a rollback to the XBEGIN state. As such, the kernel is not called to handle the segmentation fault, bypassing the overhead due to kernel intervention.

Attackers can exploit this loophole in TSX to create a precise side-channel timing attack on the kernel ASLR and was able to obtain consistent and distinct differences between kernel address space and non-kernel address space. Compared to the normal double page fault attack, the differences between mapped and unmapped kernel pages are much more distinct and have a lower standard deviation, causing lesser overlaps between a mapped and unmapped kernel space.

2.3 Proof-of-Concept

```
void handler(int nSignum, siginfo_t* si, void* vcontext) {
  uint64_t x, time_taken, *rec_time;
  ucontext_t* context = (ucontext_t*)vcontext;
  x = context->uc_mcontext.gregs[REG_RIP];
  time_taken = rdtsc_end() - beg;
  if(x >= 0xffffffff80000000 && loop_count < NUM_LOOPS) {
    // The first run is to store valid addresses in TLB.
    // Results are to be recorded from the second run.
    if(loop_count > 0) {
      rec_time = &(meas[(int)(((x & 0xffffffff) >> 12) - 0x80000)]);
      if (loop_count == 1) *rec_time = time_taken;
      else if (*rec_time > time_taken) *rec_time = time_taken;
    }
    context->uc_mcontext.gregs[REG_RIP] += 0x1000;
  } else if (loop_count < NUM_LOOPS) {
    // Reset the address scan loop
    context->uc_mcontext.gregs[REG_RIP] = 0xffffffff80000000;
    loop_count++;
  } else {
    context->uc_mcontext.gregs[REG_RIP] = (uint64_t)&dummyFn;
  }
  beg = rdtsc_beg();
}
```

Figure 5: Signal Handler Code Snippet

The code in Figure 5 shows a snippet of how the signal handler work. The signal handler is part of the main *aslrattack.cpp* code. The system will compile the attack code into a dynamic library and call it to obtain the desired results.

```
// TSX RTM routine that measures one probe on the address
uint64_t tsxMeasure(void *addr) {
  uint64_t beg = rdtsc_beg();

  if (_xbegin() == _XBEGIN_STARTED) {
    ((int(*)())addr)();
    _xend();
  } else {
    // TSX abort triggered!
    return rdtsc_end() - beg;
  }

  // should not reach here
  printf("Not triggered\n");
}

// Iteratively probe the address with TSX RTM, and get the minimum timing
uint64_t measure(void *addr, FILE *fp) {
  int i;
  uint64_t clk, min = (uint64_t) -1;

  for(i=0; i<ITERATION; i++) {
    clk = tsxMeasure(addr);
    if (clk < min)
      min = clk;
  }
  fprintf(fp, "%llx, %ld\n", addr, min);

  return  min;
}
```

Figure 6: Intel TSX PoC Snippet

The code snippet of the Intel TSX proof-of-concept code is in Figure 6, where the TSX portion begins at the XEBGIN portion, where it attempts to access a kernel address space. As the transactional execution fails, it will be aborted, and the processor will rollback to the XBEGIN instruction. The CPU will then go into the else condition and the time taken for the address access is being recorded and output to a csv file.

There are also codes that were written in Python 3.7 to define the GUI, invoking the attack code and plotting the results from the Double Page Fault Attack using in-built libraries.

```
temp = os.path.join(temp, "libASLRTimingAtk.so")
x = ctypes.CDLL(temp)
cFunc = x._Z8libAgentPim
cFunc.argtypes = ctypes.POINTER(ctypes.c_int), ctypes.c_size_t
cFunc.restype = None

def runAttack() :
    ptrRetBuff = (ctypes.c_int * 0x80000)()
    cFunc(ptrRetBuff, len(ptrRetBuff))
    return list(ptrRetBuff)
```

Figure 7: Invoke Attack Code Snippet

```
def updPlot(xData, yData):
    global frmCounter
    subp.clear()
    axesSetup(subp)
    subp.scatter(xData, yData, c='b', marker='.')
    canvas.draw()
    frmCounter += 1

def updPlotMin():
    numRounds = 3
    xData = np.linspace(0x80000, 0xfffff, 0x80000)
    yData = runAttack()
    updPlot(xData, yData)
    for _ in range(numRounds - 1):
        yTmp = runAttack()
        for i in range(len(yData)) :
            if yData[i] > yTmp[i] :
                yData[i] = yTmp[i]
        del yTmp
        updPlot(xData, yData)
    btnPlotText.set("Attack Again")

btnPlot = tkinter.Button(master=root, textvariable=btnPlotText,
  command=updPlotMin)
btnPlot.pack(side=tkinter.BOTTOM)
```

Figure 8: Graph Plot Code Snippet

Figures 7 and 8 are snippets from GUI.py that can be found in our GitHub. Their purpose is to translate the attack into visible results for this experiment.

In the demonstration program, the attack C program (for Linux) is compiled into a dynamic library, which probes the access timing of double page fault. Multiple trials are to be attempted, after which the minimum timing results will be returned. The user interface, written in Python, can then launch the attack function and visualize the result in the plot. The Python program can also load the real kernel-space address layout, obtained from /proc/kallsyms with root privileges, to help to evaluate the attack outcome.

3. EVALUATION

In this section, we will discuss about our attack experiment, with both the original double page fault attack and its TSX variant and perform an evaluation of our experiment from the results observed from the experiment.

3.1 Double Page Fault Experiment

CPU & OS	Average Kernel Address Timing (cycles)	Average Non-kernel Address Timing (cycles)
Ubuntu 18.04 Intel i3-7100H	5290	5373
Ubuntu 18.04 VMWare Intel i3-7100H	7833	7996
Ubuntu 18.04 VirtualBox Intel i7-7700HQ	4127	3680
Ubuntu 18.04 VMWare Intel i5-8600	3525	3437
Fedora 30 Intel	4914	4898

Table 1: Double Page Fault Experimental Results

On Linux, we conducted the double page fault attack on several Linux OS, mainly on Ubuntu 18.04 and Fedora 30, on both main operating systems and virtual machines such as VMWare and VirtualBox. In order to effectively perform the attack, we turned off the Kernel Page Table Isolation (KPTI) on the modern Linux kernels, as they will prevent the attack from happening. The results from the experiment is represented in Table 1.

From our observations, the number of clock cycles for kernel and non-kernel address have very small differences, and in a few cases, the kernel address takes an even longer time than the non-kernel address, which contradicts our theoretical results.

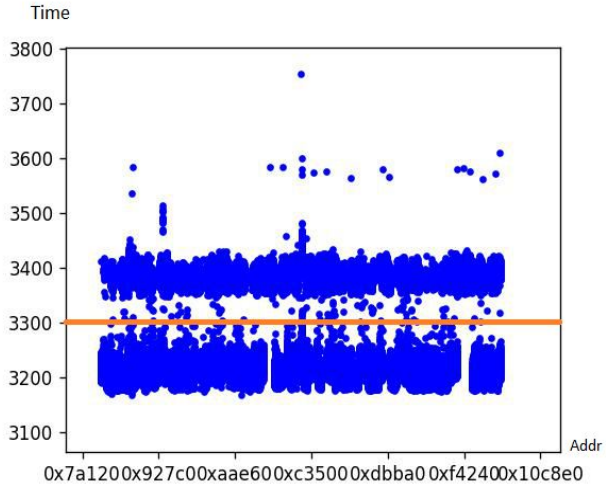


Figure 9: Graph of Ubuntu 18.04 <sample>

From the graph in Figure 9, we observe that there is a mixture of kernel and non-kernel addresses in between 3150 to 3450 clock cycles. There are two distinct groups from the results collected, where one group belongs in the 3200 clock cycle range while the other group belongs in the 3400 clock cycle range. As the kernel and non-kernel address appears equally on both groups, we are unable to successfully differentiate the kernel and non-kernel address space.

3.2 Double Page Fault Evaluation

We repeated the attack multiple times on several devices and obtained different results each time. The differences between the clock cycles for both kernel and non-kernel address are too small in order to distinguish the two. As the results we have obtained contradicts with the theoretical results that we expected, we looked at some of the reasons why we did not get the expected outcome we wanted.

Noise Generated by OS Handler. There is a large standard deviation in the timings generated from our experiment, largely due to the OS Handler. Although the total time measured for the Page Fault is several thousand clock cycles, the actual Page Fault takes less than 100 cycles, and the remaining time is caused by the OS exception handling of the segmentation fault. During the OS exception handling phase, it introduces noise to our data collected due to OS interrupts or other exceptions. This causes the number of clock cycles we measured to vary, causing a large standard deviation.

Small differences in kernel and non-kernel address space. The difference between a kernel address and a non-kernel address is only about 20 clock cycles. Comparing this difference to the noise generated by the OS Handler, this makes it challenging for us to be able to deduce the correct kernel address accurately.

3.3 TSX Double Page Fault Variant

Amongst all the devices that we have experimented with the Double Page Fault attack, only 1 of the devices supports the Intel TSX instructions. We were able to run the TSX Variant of the double page fault attack and the results are recorded in Table 2.

	Average Kernel Address Timing (cycles)	Average Non-kernel Address Timing (cycles)
Ubuntu 18.04 Intel i5-8600	166.1	175.7

Table 2: TSX Double Page Fault Experimental Results

From the results, we can immediately see a huge decrease in the amount of clock cycles, getting 166.1 clock cycles and 175.7 clock cycles for kernel and non-kernel address respectively. Without the noise generated from the OS exception handler, we were able to see a more distinct difference between the kernel and non-kernel address. We repeated the attack and were able to consistently observe similar results every time.

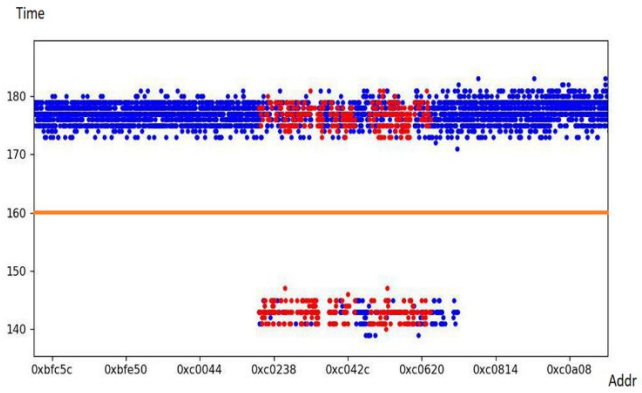


Figure 10: Graph of TSX Variant Attack

As seen in Figure 10, there is a distinct difference between the kernel and non-kernel address, which can be clearly seen once the noise from the OS Handler is removed. The red points denote kernel address while the blue points denote non-kernel address. However, it can be seen that there are some false positives where some non-kernel addresses take a shorter time, with similar time taken compared to kernel addresses, and some false negatives, where some kernel addresses take longer, with similar time taken compared to the non-kernel addresses.

3.4 TSX Attack Evaluation

Overall, the results obtained from the experiments tallies with our theoretical and expected outcome, where the time taken for a kernel address page fault is lesser than a non-kernel address page fault. As the measured timing in this case is purely the time taken for the TLB hit and TLB miss, it does not have as much variance in the result compared to the normal double page fault attack.

For the false positives, we propose that they might be actual kernel addresses that are not listed in the */proc/kallsyms* file. Since we are using the kernel addresses found in the */proc/kallsyms* file, these hidden entries that are not listed in the file are not recognised as kernel addresses, hence causing false positive entries.

For the false negatives, we propose that they might be caused due to system interrupts, causing the time taken to increase. Since we have only tested the TSX attack variant on one device only, more testing is required in order to determine the cause of the false positives and negatives.

Nevertheless, the TSX Double Page Fault attack variant is able to identify most kernel address space with a low percentage of false positives.

4. DISCUSSION

In this section, we will discuss possible mitigations to prevent a double page fault attack, and some limitations on the double page fault attack.

5.1 Mitigations

As the root cause of this side channel timing attack comes from the time difference between an allocated and unallocated kernel memory address, we will need to address this issue, such that there will be no time difference at all.

One suggestion to mitigate this attack is to modify the execution time of the OS execution handler such that the timing between an allocated and unallocated kernel space will be the same.

Another approach would be to perform the access permission check first before the MMU creates a page table entry. This will successfully mitigate the double page fault attack, as a TLB entry will not be created this way, and the page fault timing for allocated and unallocated kernel space would be the same.

An approach that is used in modern computers is the Kernel page-table isolation (KPTI). It isolates the kernel and user address space, preventing users from accessing the kernel memory to prevent side channel timing attacks such as the double page fault attack. A brief description of KPTI is available in Section 6.

5.2 Limitations

Despite the successful derandomization of kernel ASLR, the double page fault attack has several limitations.

Unable to work on AMD Processors. The double page fault attack does not work against AMD processors as the implementation of the MMU is different from the Intel processors. The AMD processors do not create a TLB entry for the valid kernel addresses, hence there will be no difference in the timing between a kernel and non-kernel address space for AMD processors.

Limited TSX Support. Even amongst the Intel processors, only a few high-end Intel processors support Intel TSX extensions. As such, it will be difficult in conducting a double page fault TSX attack in public.

Limited Sample Size. As we have only conducted our attack on 5 devices, Ubuntu 18.04, Fedora 30**,** and only 1 device with a CPU that supports Intel TSX architecture (i5-8600), our results may not accurately represent all computers.

6. RELATED WORK

In this section, we will discuss about the related work regarding kernel address space and side-channel attacks on ASLR.

DrK: Breaking KASLR using Intel TSX. *Jang et a*l. [2] presents a similar Double Page Fault attack using the Intel TSX variant. They were able to break KASLR on all major operating systems, successfully identifying mapped and unmapped memory pages. For the mapped memory pages, they were also able to identify executable and non-executable parts of the address.

Branch Target Buffer Side-Channel Attack on ASLR. *Gruss et al* [7] demonstrated a side-channel attack on the branch target buffer (BTB) to derandomize kernel and user space ASLR. Their attack exploited the fact that the BTB is shared among other applications for this side-channel attack to work. Their attack requires two processes, a victim process and a spy process. The victim process performs normal execution, and creates BTB entries, the spy process tries a list of random addresses and executes the branch instruction. The time taken for the branch instruction is measured, where a longer execution time means there the branch address has collided with the kernel branch address.

The attack was performed on an Intel Haswell Processor, and they were able to successfully uncover the kernel ASLR in 60 milliseconds.

Prefetch Side-Channel Attack on ASLR. *Etvyushkin et al* [8] explored another type of side-channel attack on ASLR called the Prefetch attack. The prefetch attack exploits two properties, where the execution time of the prefetch instruction depends on the state of CPU internal caches and prefetch instructions bypasses privilege checks. Using these properties, they were able to use prefetch instructions to differentiate the cache hit and misses for memory address and measure the timing between a cache hit or miss.

Their attack was able to uncover ASLR on Windows 10 and Linux OS, and leaked translation level for virtual addresses on both Intel x86 and ARMv8-A architectures.

Kernel page-table isolation (KPTI). KPTI (previously known as KAISER - Kernel Address Isolation to have Side channels Efficiently Removed) [5] is a feature in the Linux kernel that is used to defend against the *Meltdown* attack. It enforces a strict kernel and user space isolation, limiting the access to kernel space from user mode. They showed that there were able to successfully prevent double page fault attacks, TSX-based side-channel attacks and pre-fetch side-channel attacks.

KPTI uses shadow address spaces for its implementation of kernel address isolation. In this case, every process has two different address spaces. The first one maps the user space while the second one maps the kernel address space but has its user space protected with Supervisor Mode Access Prevention (SMAP) and Supervisor Mode Execution Prevention (SMEP). The protection prevents the kernel from executing instructions or data from the user space.

KPTI successfully prevents the Double Page Fault Attack, where it shows that the time taken to access both mapped and unmapped kernel address space are now the same, making it impossible to differentiate a mapped and unmapped kernel page. KPTI is able to prevent the attack due to the page table isolation, and the kernel addresses are not valid in the user space and are not mapped, and the MMU will not create a TLB entry for it.

7. CONCLUSION

In this paper we discussed a side-channel timing attack, the Double Page Fault attack and the Intel TSX variant, and provided a proof-of-concept using the attack to derandomize Linux kernel ASLR. We performed an experiment on Ubuntu 18.04 OS and discussed some observations from the experiment. We also explored other works ASLR side-channel attacks and mitigations to prevent these attacks.

8. ACKNOWLEDGMENTS

Our thanks to Prof. Hugh Anderson for his advice and helpful feedback.

9. REFERENCES

[1] Hund, R., Willems, C. and Holz, T. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. IEEE Symposium on Security and Privacy, 2013. DOI=http://doi.org/10.1109/SP.2013.23

[2] Jang, Y., Lee, S. and Kim, T. 2016. DrK: Breaking Kernel Address Space Layout Randomization with Intel TSX. In 23rd ACM Conference on Computer and Communications Security. DOI=https://doi.org/10.1145/2976749.2978321

[3] Linux Inside: https://legacy.gitbook.com/book/0xax/linux-insides/details

[4] Lipp M., et al. Meltdown: Reading Kernel Memory from User Space, https://meltdownattack.com/meltdown.pdf

[5] Gruss D., Lipp M., Schwarz M., Fellner R., Maurice C. and Mangard S., KASLR is Dead: Long Live KASLR, https://gruss.cc/files/kaiser.pdf

[6] Intel TSX Overview (2019) https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview

[7] Gruss D., Maurice C., Fogh A., Lipp M. and Mangard S., Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR https://gruss.cc/files/prefetch.pdf

[8] Evtyushkin D., Ponomarev D. and Abu-Ghazaleh N., Jump Over ASLR: Attacking Branch Predictors to Bypass ASLR http://www.cs.ucr.edu/~nael/pubs/micro16.pdf