Breaking Kernel Address Space Layout Randomization with Intel TSX

Yeongjin Jang, Sangho Lee, and Taesoo Kim
School of Computer Science, Georgia Institute of Technology
{yeongjin.jang, sangho, taesoo}@gatech.edu

ABSTRACT

Kernel hardening has been an important topic since many applications and security mechanisms often consider the kernel as part of their Trusted Computing Base (TCB). Among various hardening techniques, Kernel Address Space Layout Randomization (KASLR) is the most effective and widely adopted defense mechanism that can practically mitigate various memory corruption vulnerabilities, such as buffer overflow and use-after-free. In principle, KASLR is secure as long as no memory leak vulnerability exists and high entropy is ensured.

In this paper, we introduce a highly stable timing attack against KASLR, called DrK, that can precisely de-randomize the memory layout of the kernel without violating any such assumptions. DrK exploits a hardware feature called Intel Transactional Synchronization Extension (TSX) that is readily available in most modern commodity CPUs. One surprising behavior of TSX, which is essentially the root cause of this security loophole, is that it aborts a transaction without notifying the underlying kernel even when the transaction fails due to a critical error, such as a page fault or an access violation, which traditionally requires kernel intervention. DrK turned this property into a precise timing channel that can determine the mapping status (i.e., mapped versus unmapped) and execution status (i.e., executable versus non-executable) of the privileged kernel address space. In addition to its surprising accuracy and precision, DrK is universally applicable to all OSes, even in virtualized environments, and generates no visible footprint, making it difficult to detect in practice. We demonstrated that DrK can break the KASLR of all major OSes (i.e., Windows, Linux, and OS X) with near-perfect accuracy in under a second. Finally, we propose potential countermeasures that can effectively prevent or mitigate the DrK attack.

We urge our community to be aware of the potential threat of having Intel TSX, which is present in most recent Intel CPUs—100% in workstation and 60% in high-end Intel CPUs since Skylake—and is even available on Amazon EC2 (X1).

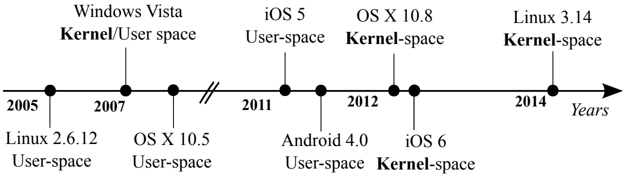


Figure 1: The adoption status of both user-space and kernel-space ASLR in popular operating systems, ordered by year [62].

1. INTRODUCTION

Enhancing the security of operating systems (OSes) has been an active and important research topic since the underlying OS is commonly considered to be the Trusted Computing Base (TCB) for user applications and their security mechanisms. Among various hardening techniques, Kernel Address Space Layout Randomization (KASLR) is the most comprehensive and effective security mechanism and raises a practical hurdle for exploiting memory corruption vulnerabilities [31, 32], such as buffer overflow and use-after-free. In this regard, today's major commodity OSes (*e.g.*, Windows, Linux and OS X) as well as mobile OSes (*e.g.*, Android and iOS) have implemented and deployed KASLR to protect the core kernel image and device drivers from exploitation (see Figure 1).

In principle, KASLR can effectively (*i.e.*, statistically) mitigate exploitation, as long as two assumptions hold: 1) no memory disclosure vulnerabilities exist and 2) enough randomization entropy is guaranteed. Therefore, typical attacks against the kernel require a preceding attack, which focuses either on leaking code or data pointers [17, 40, 42] to directly figure out the memory layout, or on exploiting implementation caveats to indirectly break the imperfect randomness [44], as a stepping stone for the ultimate control-hijacking attack.

To the best of our knowledge, exploiting the cache-based timing channel [27] is the first attempt to *universally* break KASLR without violating these two fundamental assumptions. The key idea of the cache-based timing attack is to exploit a timing difference (*i.e.*, cache miss and hit) for accessing mapped (*i.e.*, cached) and unmapped (*i.e.*, not-cached) pages to determine page mapping status. More precisely, it deliberately fills or evicts certain cache lines to indirectly affect the execution time in the kernel space. Such timing differences can be observed by measuring how quickly a system call returns from the kernel space, or how quickly a faulty access to the kernel space gets to the OS page fault handler. Under a typical threat model—local privilege escalation, this attack can break KASLR (*i.e.*, leaking the partial bits of a randomized address) in theory, but it barely works in practice for three reasons. First, it generates strong signals (*e.g.*, segmentation faults and lots of system calls) that typical system monitoring tools (*e.g.*, `fail2ban` and `sysdig`) consider to be abnormal behavior, thereby resulting in prompt mitigation.

CCS'16, October 24 - 28, 2016, Vienna, Austria

ISBN 978-1-4503-4139-4/16/10. . . $15.00

DOI: http://dx.doi.org/10.1145/2976749.2978321

Grade/Generation	Skylake	Broadwell	Haswell
Server/Workstation	17/17 (100.0%)	19/19 (100.0%)	37/85 (43.5%)
High-end Consumer	23/38 (60.1%)	11/22 (50.0%)	2/92 (2.2%)
Low-end Consumer	4/32 (12.5%)	2/16 (12.5%)	0/79 (0.0%)

Table 1: Commodity Intel CPUs supporting TSX, varying CPU grades and generations (February 2016). Server CPUs include Xeon and Pentium D server, high-end consumer CPUs include i5 and i7, and low-end consumer CPUs include i3, m, and others. All recent CPUs for server/workstation and more than half of high-end consumer CPUs support TSX [28].

Second, it requires a large page (2 MB) to accurately locate the virtual address regions to intentionally generate cache conflicts to the targeted physical pages, which unfortunately requires higher privileges than a normal user in most commodity OSes: `hugetlbfs` in Linux [57], and `SeLockMemoryPrivilege` in Windows [13, 63].

Lastly, the attack is neither accurate (*i.e.*, reversely mapping a conflicted cache line to its preimage set) nor fast enough (*e.g.*, their double page fault attack took 17.3–72.9 s to probe the entire kernel space of 32-bit Windows 7 in a carefully controlled environment) for practical use. In fact, these practical hurdles are the essential foundation of currently proposed software-based mitigation schemes [27].

In this paper, we introduce a highly stable timing attack against KASLR, called DrK ([dɪræk] **De-Randomize Kernel** address space), which is similar in spirit to the previous universal attack [27], but with higher accuracy and better performance. To break KASLR in an OS-agnostic way, DrK exploits a timing side-channel in a new hardware feature, called Intel Transactional Synchronization Extension (Intel TSX), that is widely deployed in modern Intel CPUs—in our survey, 100% of CPUs for server/workstation and 60% in high-end computers since Skylake have Intel TSX (see Table 1). Our attack has higher precision (*e.g.*, executable bits of pages), higher accuracy (*e.g.*, near-prefect de-randomization of memory layout), and is faster (*e.g.*, under a second) than the state-of-the-art cache-based attack. More importantly, DrK does not generate distinctive footprints that look abnormal to system monitoring tools, and is universally applicable to all OSes, even under a virtualized environment (*e.g.*, Amazon EC2).

The key idea of DrK is to exploit an unusual behavior of TSX in handling erroneous situations inside a transaction. When a transaction aborts (typically due to read or write conflicts), the CPU directly invokes an abort handler (specified by a user) to resolve it without interrupting the underlying OS. However, even when an unrecoverable error happens inside a transaction (*e.g.*, an access violation or a page fault), the CPU similarly aborts the transaction without informing the underlying OS, although these errors traditionally require the intervention of the underlying OS.

In DrK, we turned this property into a better timing channel, enabling us to precisely determine the mapping status (*i.e.*, mapped versus unmapped) and executable status (*i.e.*, executable versus non-executable) of the privileged address space, by intentionally generating an access violation inside a transaction (*e.g.*, accessing or jumping into kernel address regions).

In this paper, we make three significant contributions:

- **A practical attack.** We demonstrate that DrK can break the KASLR of popular OSes, including the latest Windows, Linux and OS X, with near-perfect accuracy and high precision with sub-second execution time.
- **Analysis.** We provide an in-depth analysis of the DrK attack with our hypothesis and experiment designs. We show our results in three major OSes to understand the root cause (*i.e.*, CPU internal architecture) of the timing differences.
- **Countermeasures.** Although we believe it could be hard to have a practical software-based mitigation, we propose several countermeasures that can effectively prevent a DrK attack.

The remainder of this paper is organized as follows. §2 introduces KASLR and Intel TSX. §3 explains how our attack works. §4 shows our evaluation results. §5 provides an in-depth analysis of DrK to understand the hardware characteristics. §6 proposes possible countermeasures. §7 discusses the limitations of DrK. §8 compares it with other projects, and §9 concludes this paper.

2. BACKGROUND

In this section, we provide a technical overview of KASLR and Intel TSX as a basis for understanding the technical details of the DrK attack.

2.1 Kernel ASLR

ASLR is a comprehensive and popular defense mechanism that mitigates memory corruption attacks in a probabilistic manner. To exploit a memory corruption vulnerability, such as use-after-free, attackers need to figure out the memory layout of a target process or the system ahead of time. ASLR mitigates such attacks by incorporating a non-deterministic behavior in laying out the program's or system's address space. More specifically, whenever a program is loaded or a system is booted, the ASLR mechanism randomizes their address spaces, including code and data pages.

Since ASLR is highly effective in practice, most real exploits first have to bypass ASLR (or KASLR) before attempting to launch a real control-hijacking attack, such as return-oriented programming (ROP). For example, most web browser exploits demonstrated in the latest Pwn2Own competition [41, 43] include one or more information leak vulnerabilities to bypass KASLR, thereby escaping a user-level sandbox. For this reason, all major commodity OSes, including Windows, Linux, and OS X, as well as mobile OSes, including Android and iOS, have deployed ASLR in user space and recently applied it to kernel space.

Adoption. Figure 1 shows the timeline of the ASLR deployment in popular OSes. Microsoft started supporting KASLR in Windows Vista (2007) and Apple started its support with iOS 5 and OS X 10.8 (2012). Linux has provided KASLR as an option (*i.e.*, `CONFIG_RANDOMIZE_BASE=y`) since kernel version 3.14 (2014) and, recently, popular distributions (*e.g.*, Ubuntu 15.04) have enabled KASLR by default.

Implementation. Table 2 summarizes how 64-bit commodity OSes implement KASLR for kernel text and modules in terms of entropy (*i.e.*, amount of randomness) and granularity (*i.e.*, unit of randomization). The entropy of KASLR is determined by the kernel address range (*e.g.*, 1 GB–16 GB) and the size of alignment, which is usually a multiple of the page size (*e.g.*, 4 KB–16 MB) for better performance and memory utilization. For example, Linux's kernel address range is 1 GB (30 bits) and its alignment size for kernel text is 16 MB (24 bits), so that its KASLR entropy is only 6 bits (*i.e.*, 64 slots for the location). In contrast, Windows 10's kernel address range is 16 GB (34 bits) and its alignment size for kernel text is 2 MB (21 bits), so that its KASLR entropy is 13 bits (*i.e.*, 8,192 slots for the location). Thus, a set of all possible randomized base addresses is

$$\{\texttt{base\_address} + s \times \texttt{align\_size} : 0 \le s < \#slots\}.$$

2.2 Intel TSX

In this section, we explain the basic concept of Intel TSX to help understand the DrK attack. Intel TSX is Intel's implementation of hardware transactional memory (HTM) [23, 36, 37, 58, 61]. HTM

OS	Types	Entropy	#Slots	Address Range	Align Base	Align Size	Broken by DrK?
Linux	Kernel	6 bits	64	`0xffffffff80000000` – `0xffffffffc0000000`	0x1000000	16 MB	✓ (100.00%)
	Modules	10 bits	1,024	`0xffffffffc0000000` – `0xffffffffc0400000`	0x1000	4 KB	✓ (100.0%)
Windows	Kernel	*13 bits	8,192	`0xfffff80000000000` – `0xfffff80400000000`	0x200000	2 MB	✓ (100.0%)
	Modules	*13 bits	8,192	`0xfffff80000000000` – `0xfffff80400000000`	0x200000	2 MB	✓ (99.98%)
OS X	Kernel	8 bits	256	`0xffffff8000000000` – `0xffffff8020000000`	0x200000	2 MB	✓ (100.0%)

Table 2: Summary of KASLR implementations in popular OSes. According to our experiment, all KASLR implementations we tested generate a random address by adding a random offset to the fixed base address (*i.e.*, kernel or module base) either at the booting time or when loading modules. The numbers marked in blue color indicate varying, randomized ranges, so called entropy. (*) Johnson and Miller [31] reported that Windows has 17-bit worth of entropy for the kernel and 19-bit for modules, but we have only observed 13-bit of entropy during our experiments.

```
// begin a transaction
if (_xbegin() == _XBEGIN_STARTED) {
  // the transaction starts
  ...
  // this transaction successfully terminated
  _xend();
} else {
  // the transaction is aborted
  abort_handler();
}
```

Figure 2: A minimal code snippet that derives TSX: this example executes the code block in the `if`-statement transactionally, meaning that any error inside the code block makes it roll back.

provides lock-less synchronization among threads by ensuring transactional execution at the hardware level; *i.e.*, it enables concurrent access to shared memory by multiple threads and discards changes if a read-write conflict, write-write conflict, or other error happens during the transaction.

Note that this paper's main concern is not how we effectively use TSX to process transactions, but how we exploit the way it handles an *exception*, which accidentally exposes a clear, stable timing channel.

Example code. To explain our attack, we first introduce how a transaction can be implemented by using TSX, which we will use as a template for the actual attack. Figure 2 shows a minimal source snippet to run a transaction. A transaction region starts with `_xbegin()` and terminates when `_xend()` is invoked (*i.e.*, committed). Then, all instructions (*e.g.*, `if`-statement in Figure 2) in the transactional region are guaranteed to be *atomically* executed. However, a transaction might fail (*i.e.*, abort) as well: for example, when two or more concurrent transactions affect each other during the execution—a read-write or a write-write conflict, depending on how they affect each other. In such a case, it automatically rolls back the aborted transactions (*e.g.*, cleaning up the overwritten memory space) and invokes an abort handler specified by a user (*e.g.*, `else`-branch in Figure 2). The more *interesting* situation, in terms of security, is when erroneous situations occur during the execution: for example, segmentation faults, page faults, or even interrupts.

Suppressing exceptions. According to Intel's manual ([30, §15.3.7]), a transaction aborts when such a hardware exception occurs during the execution of the transaction. However, unlike normal situations where the OS intervenes and handles these exceptions gracefully, TSX instead invokes a user-specified abort handler without informing the underlying OS. More precisely, TSX treats these exceptions in a *synchronous* manner—immediately executing an abort handler while suppressing the exception itself. In other words, *the exception inside the transaction will not be communicated to the underlying OS*. This allows us to engage in abnormal behavior (*e.g.*, attempting to access privileged, *i.e.*, kernel, memory regions) without worrying about crashing the program. In DrK, we break KASLR by turning this surprising behavior into a timing channel that leaks the status (*e.g.*, mapped or unmapped) of all kernel pages.

3. THE DrK ATTACK

In this section, we provide a high-level description of the DrK attack, which breaks KASLR by exploiting a timing channel in TSX. As explained in §2.2, when an exception occurs inside a transaction, TSX aborts its execution and, importantly, suppresses the exception (*i.e.*, no OS intervention). The key idea of DrK is to measure the *timing difference* in handling a transaction abort when attempting to access mapped kernel memory regions compared to unmapped regions. Accessing the kernel space from a user process incurs an access violation (*i.e.*, a page fault), but TSX suppresses this exception and immediately invokes its abort handler. The mapping status of the targeted kernel address results in a time difference (an order of a few hundred cycles) in invoking the abort handler due to the subtleties in TSX's micro-architecture (see §5). More importantly, this attack is not observable to the OS, as these exceptions are all suppressed. Furthermore, unlike prior attacks that only try to distinguish between mapped and unmapped pages, the DrK attack can even extract the executable and non-executable bit of every kernel page.

3.1 Threat Model

The DrK attack is built on four realistic assumptions:

1. The attacker has unrestricted access to the local, user-level, and non-root privilege execution environment of the target system.
2. The attacker knows a memory corruption vulnerability in the kernel space, but needs to bypass the KASLR deployed in the target system in order to exploit this vulnerability.
3. The attacker does not have any explicit way to figure out the kernel memory layout.
4. The attacker can gather the information of the target system: for example, the OS version or CPU information.

This threat model is very realistic. For example, the platform-as-a-service (PaaS) cloud services such as Heroku [25] provide a local execution environment that satisfies all of the assumptions above. Similarly, these assumptions hold true in exploiting the vulnerabilities in modern web browsers due to their user-level sandbox; real exploits demonstrated in the Pwn2Own competition [41, 43] are performed under the same threat model as the DrK attack. Moreover, the operating systems disallow user-level code to access to their kernel address space information. In Ubuntu, access to `/proc/kallsyms`, which shows all mappings of kernel space, is prohibited to non-root users. In Windows 10, there is a system call `NtQuerySystemInformation()` which allows a normal user to see the current mapping of the kernel. However, in the `LOW` or `UNTRUSTED` integrity level that is generally set as the running level of sandboxed applications (*e.g.*, the renderer process of Google Chrome), it is not allowed to access the system call to get the address layout of the

```
 1 // The given argument addr is an address for a kernel page.
 2 uint64_t do_probe_memory(void *addr, mode_fn fn)
 3 {
 4   // Timer starts
 5   uint64_t beg = rdtsc_beg();
 6   // initiate TSX region
 7   if (_xbegin() == _XBEGIN_STARTED) {
 8     // fn() performs either 1) or 2).
 9     // 1) execute   : mov rax, addr; jmp rax
10     // 2) read      : mov rax, [addr]
11     fn(addr);
12     // commit TSX, which will never take place.
13     _xend();
14   } else {
15     // TSX aborted; end timer, return the timing.
16     return rdtsc_end() - beg;
17   }
18   errx(1, "Not reachable");
19 }
20 // probe the address for multiple times
21 uint64_t probe_memory(int ntimes, void *addr, mode_fn fn)
22 {
23   uint64_t min = (uint64_t)-1; // UINT64_MAX
24   while (ntimes --) {
25     uint64_t clk = do_probe_memory(addr, fn);
26     // Only record the minimum timing observed.
27     if (clk < min)
28       min = clk;
29   }
30   return min;
31 }
```

Figure 3: Code snippet that probes timing for a kernel address access. The access on address through `fn(addr)` always raise an exception (*i.e.*, access violation) which makes the transaction abort. In the DrK attack, we measure the minimum timing value from multiple trials of probing for determining page mapping status (*e.g.*, 215 cycles for mapped region and 245 cycles for unmapped region).

kernel. We confirmed that DrK works in such restricted integrity levels.

3.2 Overview

Figure 3 shows a code snippet that we ran to probe a kernel address for its mapping status. We perform two types of access to a kernel address `addr` inside a TSX region (`if`-statement, from `_xbegin()` to `_xend()`): (1) try to execute on `addr` by running `mov rax, addr; jmp rax` (*exec*), and (2) try to read a value from `addr` by running `mov rax, [addr]` (*read*). Note that since `addr` is a kernel address, the access is not performed, instead generating an exception, which makes the transaction abort.

We measure the timing between the initialization of the TSX region (`_xbegin()`) and the abort handler (line 15-16). Since the DrK attack relies on the timing difference on the hardware critical path (see §5), we measure the minimum timing for a memory page.

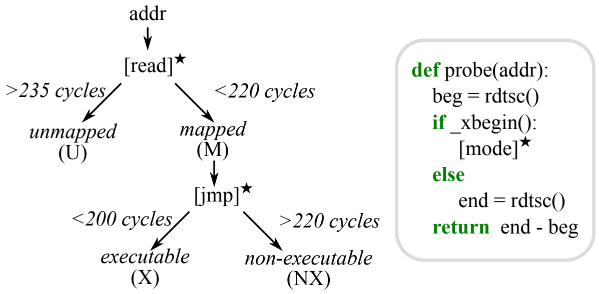
Figure 4 shows an example of how timing can determine the mapping status of a kernel page. First, we attempted to read a given kernel address. It took more than 235 cycles for unmapped pages and less than 220 cycles for mapped pages. (see Figure 6a).

Next, we attempted to execute (*i.e.*, `mov rax, addr; jmp rax`) on a given kernel address. It took less than 200 cycles for the executable pages and more than 220 cycles if the page is non-executable (see Figure 6b).

We observed a significant timing difference (±10%) that can be used for a timing channel to *precisely* identify the mapping and executable permission status for a given kernel address.

Probing strategy. Figure 5 shows how DrK works. Basically, it consists of two steps: (1) collect timing information for the memory (probing step) and (2) determine the kernel map using the timing information.

In (1) we first analyze the target OSes to determine some invariant of the kernel address space layout. This step is essential to avoid searching the entire 64-bit address space. For example, we can use the information in Table 2 to largely reduce the random search space.



```
def probe(addr):
  beg = rdtsc()
  if _xbegin():
     [mode]★
  else
     end = rdtsc()
  return  end - beg
```

Figure 4: An overview of the timing attack in DrK. From the timing differences in calling TSX abort handler for read access (U/M) and execution access (X/NX) inside a transaction, the DrK attack can infer the memory layout of the OS, as well as their permission status (*i.e.*, executable or non-executable). The numbers are collected on a system running Ubuntu 16.04 LTS (kernel 4.4.0), on Intel Core i7-6700K (Skylake) 4.0 GHz processor.

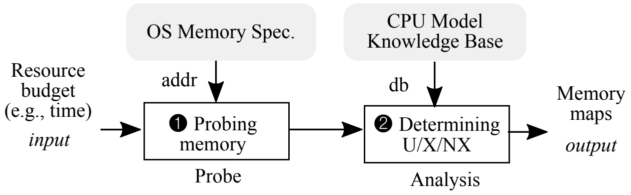


Figure 5: Our attack consists of two steps: ❶ Probing and ❷ Analysis. ❶ Given resource budgets provided by users (*e.g.*, time or iteration) and memory specification of the target OS (*e.g.*, maps, types, and page size), we first probe the timing of the TSX aborts in each memory region. ❷ Then, with the pre-measured knowledge base on the timing characteristics of each CPU model, we determine the memory layout (*e.g.*, mapped regions) and status (*e.g.*, executable regions).

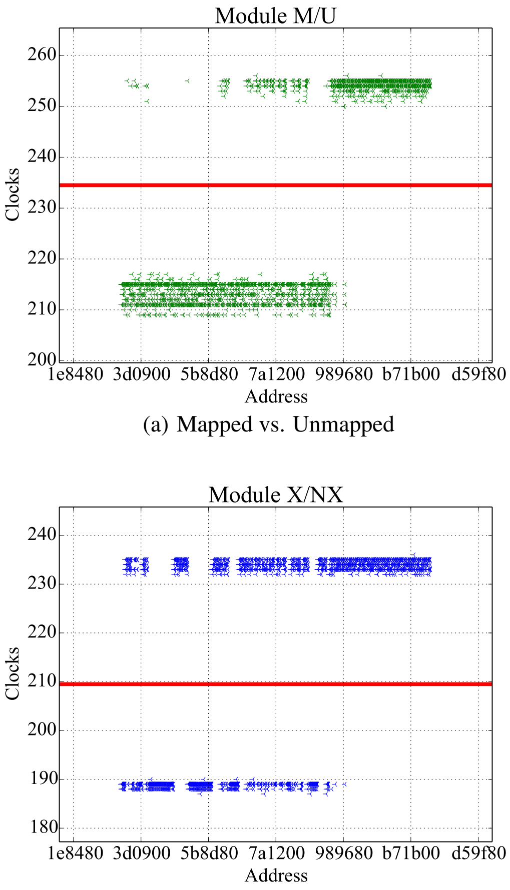
In the analysis step (2), we determine the mapping status of the kernel address using the timing information gathered in step (1). Note that the timing information largely depends on the hardware characteristics (*e.g.*, CPU generation and clock frequency, see Table 3), so we need to know such information for the CPU. We can collect such information during an offline analysis and store it in a database.

Summary. The benefits of DrK, *accuracy*, *covertness*, and *OS-independence*, come from the characteristics of TSX. On accessing a kernel address, the occurrence of a page fault inside the TSX region will directly call an abort-handling procedure in the user-space process without notifying the operating system. This *shortcut* not only minimizes measuring noise but also probes any kernel address without the OS being aware of it and does not rely on how an OS interrupt handler is implemented. This is a clear advantage over Hund *et al.*'s attack against KASLR [27], which relies on an OS interrupt handler, leading to high noise, OS-awareness, and OS-dependency.

4. ATTACK EVALUATION

We attempt to answer the following questions to evaluate DrK:

- How different are the TSX timing characteristics across various CPU and OSes? (§4.1)
- How effectively can DrK break KASLR of the popular OSes? (§4.2, §4.2.3)
- How does DrK work in a virtualized environment (§4.3)?
- With what configuration does DrK give high precision (§4.4)?
- How much better is DrK than prior attacks on KASLR (§4.5)?



(a) Mapped vs. Unmapped

(b) Executable vs. Non-executable

Figure 6: Two timing graphs for measuring timings on Linux kernel modules area, running on a Skylake (Core i7-6700K) processor. The graph shows that the difference in the minimum timing (see Table 3) is sufficient to set the threshold (the red line in each graph) for determining page mappings. In the upper graph, the red line placed at 235 cycles clearly distinguishes the mapped from the unmapped pages of the modules. Similarly, the red line placed at 210 cycles in the lower graph clearly separates the executable pages from the non-executable ones (including unmapped pages). We probed the timing on accessing each page in the Linux kernel module area 100 times, while running kernel version `4.4.0`. Modules are loaded from `0xffffffffc0347000` to `0xffffffffc0bf9000`, and the DrK attack breaks KASLR with perfect (100%) accuracy.

4.1 Characteristics of the Timing Channel

The DrK attack uses timing information as an oracle for determining the mapping status of kernel memory pages. We rely on the timing difference on accessing each type of kernel page mappings: unmapped, mapped, and executable. A necessary condition that enables such a distinction is that there should be a prominent timing gap to determine different mappings.

We observed that the timing channel measured with a TSX abort handler has a significant timing gap between the different mappings. Table 3 shows the *minimum* timing that we could observe on accessing each type of page mapping, across 4 types of processors over 1,000 times. There are significant timing differences between mapped (fast) versus unmapped (slow) pages, and executable (fast) versus non-executable (slow, including unmapped) pages. When we attempted to access to arbitrary kernel memory pages using a `mov` instruction, the timing differences between unmapped and mapped pages were 18–31 cycles[1]. Similarly, when we attempted to execute arbitrary kernel memory pages using a `jmp` instruction, the timing differences between executable and non-executable pages were 30–44 cycles.

CPU/Types	**READ** (`mov rax,[addr]`)			**JMP** (`mov rax,addr; jmp rax`)		
	X	NX	<U	X<	NX	U
i7-6700K (4.0G, Skylake)	209	209	240	181	226	226
i5-6300HQ (2.3G, Skylake)	164	164	188	142	178	178
i7-5600U (2.6G, Broadwell)	149	149	173	134	164	164
E3-1271v3 (3.6G, Haswell)	177	177	195	159	189	189

Table 3: The minimum observed timings over 1,000 iterations of probing for the known kernel mappings, for each CPU. In the DrK attack, we use timing differences to determine the mapping status of the page. We measured the timings using `mov` and `jmp` instructions, to observe the minimum timings for unmapped (**U**), non-executable (**NX**), and executable (**X**) pages. The value indicates that having access with a `mov` instruction on unmapped pages takes 18–31 more cycles (marked in red color) than the mapped pages. Likewise, the timing for executable pages on accessing with a `jmp` instruction takes 30–44 fewer cycles (marked in blue color) than non-executable or unmapped pages. Note that the values are observed minimums, namely, we cannot observe the timings below the values per each mapping status during the experiment.

While the timings depend on the architecture or clock rate of the processor, the common characteristics of the timing is that there is always a timing gap for the page mapping types across different types of processors. Furthermore, the minimum timing does not depend on OS settings. By running the experiment over multiple environments (different OSes, or under the Xen hypervisor), we discovered that the timing is characteristic of the processor and did not depend on the software settings (see §4.4).

The difference between minimum timings for each page mapping type can be exploited to determine the memory mapping status. For example, on an Intel Core i7-6700K Skylake processor, probing unmapped pages always took more than 240 cycles. In contrast, probing on mapped pages took less than that, with a minimum of 209 cycles. If a page was probed in 230 cycles, then it is a mapped page because for unmapped pages, it is impossible for probing to take less than 240 cycles. Thus, we set the threshold to use in determining page mapping status as a value less than the minimum timing of an unmapped page. We also took a value less than the minimum timing of non-executable page as the threshold for distinguishing executable and non-executable pages.

Figure 6 shows how the timing is measured for the actual memory pages, for 100 iterations of probing. On probing mapped and unmapped pages (Figure 6(a)), we set the horizontal (red) line at 235 cycles (less than the minimum of U), and used it as the threshold for mapped pages. The line clearly separates the unmapped pages (the upper half) from the mapped pages (the lower half), and there is a clear gap of about 30 cycles between the halves. On probing the executable permission status (Figure 6(b)), we set 210 cycles (less than the minimum of NX) as the threshold for executable pages. The red line on the threshold clearly separates the non-executable or unmapped pages (the upper half) from the executable pages (the lower half), and there is a clear gap of about 40 cycles between them.

As shown in the graph, the timing channel is highly consistent. Using this timing channel, the DrK attack can clearly determine the mapping status of a page and its executable permission by comparing measured timings to the minimum threshold for the types of pages. On evaluating DrK for breaking KASLR in the commodity OSes (in §4.2), we observed that the DrK attack can achieve 100% accuracy in determining page mapping and executable status across multiple runs of the attacks.

[1] There was no difference between read and write attempts.

4.2 Breaking KASLR in Popular OSes

To demonstrate the feasibility of the DrK attack in realistic settings, we evaluated the attack on commodity 64-bit OSes that use KASLR, namely, Linux kernel `4.4.0`, Windows 10 `10.0.10586`, and Mac OS X El Capitan `10.11.4` [2]. Furthermore, we also mounted the DrK attack on a Linux virtual machine (VM) running on a Xen hypervisor, and a cloud environment (an X1 instance of Amazon EC2), to test DrK against cloud environment settings.

Table 4 summarizes the result of the DrK attack on various hardware and software configurations. In short, the DrK attack demonstrates around 99%–100% accuracy—not just mapped and unmapped pages, but also executable pages, independent to the OS—for determining kernel address mappings in all major OSes and even in a virtualized environment, in few seconds with near-perfect accuracy. To the best of our knowledge, this level of accuracy, speed, and generality in a cache timing side-channel attack has never been demonstrated before.

1. **Accuracy.** The DrK attack is highly accurate. It can identify the mapping status of a kernel address at the page-level granularity. In comparing the result from the DrK attack with the ground truth page table mappings, we achieved 100% accuracy in detecting the correct page mapping across the OSes and CPUs.

   The high accuracy of DrK lets the attacker infer more information about the kernel; for example, the mapping addresses can be used for detecting the exact location of some kernel modules. The DrK attack can accurately identify the location of the driver code in Windows, by correctly determining the base address of 97 specific drivers among a total 141 loaded drivers using the unique signature of X/NX/U mapping size information.

2. **Speed.** DrK can scan the entire possible kernel allocation space of 64-bit OSes very quickly. For attacking the start address of the kernel image, *i.e.*, getting ASLR slide, DrK is very fast: it only took 5 ms to successfully identify the base address of the Linux Kernel. For the full scanning of the Linux kernel and modules pages (more than 6,000 pages), it took less than a second while achieving 100% accuracy.

3. **Generality.** DrK does not depend on software settings. The attack works well over the latest version of all three commodity OSes (Windows, Linux, and OS X), and even works on in virtualized environments (Linux guest under Xen HVM). The root cause of the timing channel in the DrK attack bounds to the hardware specifications; therefore, if the processor supports TSX, then the system is vulnerable to the attack. This also indicates that the attack would be very difficult to defeat using software-level countermeasures.

Attack strategy. The DrK attack consists of two stages. First, we scan all possible slots of kernel pages mentioned in Table 2 to find the base and the end address of both kernel and drivers (modules). In this step, the base address (*ASLR slide*) is found. The second step is to obtain more accurate mapping information at a page-level granularity. We try to measure the permission of each page starting from the base address to the end address. Then, we compare the result with the ground truth mapping information extracted from page table entries to evaluate the accuracy of the DrK attack.

In the following, we describe in detail the experiment configuration, settings, and interesting issues and tricks specific to each OS and virtualized environment.

4.2.1 *Finding the Base and End Address*

We used the following OS-specific information to get the current mapping range of the running OS (see Table 2).

Linux. In Linux, kernel and modules addresses are mapped in different regions. The base address of the kernel is in the `0xffffffff80000000`–`0xffffffffc0000000` address range, and there are only 64 slots where the kernel can start (aligned with 16 MB mask). For the module addresses, the base address can start in the range of `0xffffffffc0000000`–`0xffffffffc0400000`, and 1,024 slots are available for base addresses for the modules (aligned with 4 KB mask). To find the base address of each region, we sequentially scan those slots from start to end.

After successfully determining the base address of the region, we find the end address of each region as follows. For the kernel, since it is always mapped as a whole chunk in Linux, we set the end address as the first unmapped page that can be found starting from the kernel base address. For the modules, while each module is mapped as a chunk, there is a region of unmapped pages between the modules. Thus, finding unmapped pages in the region can only tell the number of modules loaded in the area. To identify the end of module mapping area from this information, we use the number of modules that is currently loaded in the OS. This is available through a user level program `lsmod`, which shows the list of currently loaded modules. If the number of detected unmapped region matches the total number of modules, we set the end address at that point.

Windows. Unlike Linux, Windows does not have separated areas for kernel and driver mappings. Specifically, kernel pages and driver pages can both lie in the same address range of `0xfffff80000000000`–`0xfffff80400000000`. To distinguish between kernel pages and drivers, we use the following facts: 1) all kernel pages in Windows are mapped with executable permission, 2) the kernel uses a page size of 2 MB, and its total size is at most 6 pages (12 MB in size for the tested version of Windows 10), and 3) the kernel can either come in front of drivers or come after.

Knowing this, we scan the whole available address space from one end to the other, to find the first and the last mapped pages in the area. If first and the last pages are found, we look for a consecutive 12 MB area mapped with executable permissions at either end. This area contains kernel pages. After the kernel page range is determined, we know the remaining space that is mapped to be the module area.

OS X. For OS X, we only launched the attack to find the base address (*i.e.*, *ASLR slide*) of the kernel image. Since the kernel can be mapped in the range of `0xffffff8000000000`–`0xffffff8020000000` over 256 slots, we scan them to find the first mapped address.

On all tested operating systems, finding the kernel base and end address can be done very quickly. All scans were completed within several milliseconds (5 ms, 31 ms, and 797 ms in Linux, OS X and Windows, respectively).

4.2.2 *Accuracy: Detecting Page Mappings*

After discovering the base and the end address of kernel mappings, we performed accuracy testing of the DrK attack. Basically, we probed all mapped areas with DrK, then compared the result from the attack to the ground truth mappings extracted from the page tables.

Table 4 shows the result of our accuracy testing.

Linux. For Linux, we ran the attack on three different CPUs (i7-6700K, i5-6300HQ, and E3-1271 v3), and on the kernel version 4.4.0 running Ubuntu 16.04 LTS. DrK measured 6,147 pages in total (3,075 for kernel and 3,072 pages for the modules) and it was

[2] All tested operating systems are the latest version as of May 2016.

				Accuracy								
OS	CPU	Type	# Pages	M/U	M/U	X/NX/U	X/NX/U	Max M	Max X	# Iter	Time (s)	Clock
				Kernel	Module	Kernel	Module					
Linux	Skylake (i7-6700K)	Kernel/Modules	6,147	100.00%	100.00%	100.00%	100.00%	235	210	100	0.16	3.9 GHz
	Skylake (i5-6300HQ)	Kernel/Modules	6,147	100.00%	100.00%	100.00%	100.00%	183	163	20	0.09	3.0 GHz
	Skylake (i5-6300HQ)	Kernel/Modules	6,147	100.00%	100.00%	100.00%	100.00%	183	163	100	0.21	3.0 GHz
	Haswell (E3-1271 v3)	Kernel/Modules	6,147	100.00%	100.00%	100.00%	100.00%	192	177	100	0.19	3.9 GHz
Windows	Skylake (i5-6300HQ)	Kernel/Driver	34,258	100.00%	100.00%	See (a)	90.45%	183	163	100	4.96	2.8 GHz
	Skylake (i5-6300HQ)	Kernel/Driver	34,258	100.00%	100.00%	See (a)	98.60%	183	163	500	22.5	2.8 GHz
	Skylake (i5-6300HQ)	Kernel/Driver	34,258	100.00%	100.00%	See (a)	99.28%	183	163	1,000	45.9	2.8 GHz
Linux Xen	Skylake (i5-6300HQ)	Kernel/Modules	5,633	100.00%	100.00%	99.98%	99.98%	580 (b)	530 (b)	100	0.65	2.3 GHz
	Skylake (i5-6300HQ)	Kernel/Modules	5,633	100.00%	100.00%	100.00%	100.00%	580 (b)	530 (b)	500	2.61	2.3 GHz
Amazon EC2	Haswell (E7-8880 v3)	Kernel/Modules	6,147	100.00%	100.00%	100.00%	100.00%	181	165	100	0.27	2.7 GHz
Linux	Skylake (i7-6700K)	Kernel Base Addr	64	100.00%	-	-	-	235	-	100	5ms	3.9 GHz
Windows	Skylake (i5-6300HQ)	Kernel Base Addr	8,192	100.00%	-	-	-	183	-	100	797ms	3.0 GHz
OS X	Skylake (i7-6700K)	Kernel Base Addr	256	100.00%	-	-	-	235	-	100	31ms	3.9 GHz

Table 4: Summary of the evaluation results of the DrK attack. To calculate the accuracy, we ran the full attack 10 times, then calculated the geometric mean to show the consistency of the result over multiple runs. To break KASLR in Linux, it took around 0.2 seconds with 100% accuracy in detecting the mapping status of each page. For Windows, while the attack on determining mapped / unmapped address resulted 100% accuracy over 100 probing iterations, the attack on the executable permission did not. Running more iterations gives better accuracy: 98.60% and 99.28% on 500, and 1,000 iterations respectively. The DrK attack also works well in a virtualized environment. We ran Linux over a Xen hypervisor and in cloud environment on an X1 instance of Amazon EC2, and both resulted 100% of accuracy. For the last three rows, we only ran the DrK attack to find the base address of the kernel. It only took 5 ms for Linux, 31 ms for OS X, and 797 ms for Windows to find the ASLR slide. (a) In Windows, all pages of the kernel (`HAL` and `ntkrnlmp.exe`) are mapped with executable permissions. Therefore, we did not run X/NX detection for the pages. (b) See §4.3 and §4.4 for the effect of frequency scaling on the timings.

able to identify their mappings with 100% accuracy. In total, the attack took around 0.2 seconds (from retrieving the base address to determining all page permissions).

Windows. For Windows, we ran the DrK attack on Skylake i5-6300HQ processor running Windows 10 version `10.0.10586`. Both kernel and drivers consisted of 34,258 pages (probing 8,192 of 2 MB pages for the base and end addresses, and 26,066 pages for measuring 4 KB pages for the module addresses) to be scanned. The total attack, including scanning slots for the base address and measuring each page, completed in under five seconds with 100 iterations of probing, which yields 100% (mapping) and 90.45% (executable) accuracy in detecting module mappings.

Running more iterations on Windows gives better results in finding executable pages. When running 1,000 iterations on each page, it yields 99.28% accuracy in detecting executable pages, while taking long (45.9 s) for probing.

4.2.3 Detecting Module Addresses

```
// BASE_ADDR     -      END_ADDR     PERM   NAME     SIZE
0xffffffffc035b000-0xffffffffc0360000 U
0xffffffffc0360000-0xffffffffc0364000 X  libahci    4000
0xffffffffc0364000-0xffffffffc0368000 NX libahci    4000
0xffffffffc0368000-0xffffffffc036c000 U
0xffffffffc036c000-0xffffffffc036e000 X  i2c_hid    2000
0xffffffffc036e000-0xffffffffc0371000 NX i2c_hid    3000
0xffffffffc0371000-0xffffffffc0376000 U
0xffffffffc0376000-0xffffffffc039a000 X  drm        24000
0xffffffffc039a000-0xffffffffc03cc000 NX drm        32000
0xffffffffc03cc000-0xffffffffc03cd000 U
```

Figure 7: List of module mappings in Linux kernel 4.4.0. Note that the mapping is always done in the following sequence: X, NX, U, and a chunk of unmapped pages separates the mappings of consecutive modules. The sizes of X and NX pages are diverse by the modules.

Fine-grained module detection. The DrK attack allows for a very accurate picture of the kernel address space layout. This can be further exploited to identify the exact location of a specific module (driver). For example, from the mapping information, we can infer the addresses of modules such as `raid`, `drm`, and `libahci` in Linux, and locating the drivers such as `NTFS`, `pci`, and `msrpc` in Windows.

The DrK attack uses the size information of executable and non-executable pages of the module as a signature. Figure 7 shows the list of module mappings in Linux. In Linux, module mapping always start with a code area (`.text`), which has executable permission. Subsequently, areas such as `.bss`, `.rodata` with **NX** permission are mapped. Note that the size of any single module is likely different from that of the others. We set the size of **X/NX** areas as the signature (*e.g.*, **X**:`0x2000` and **NX**:`0x3000` for `i2c_hid` module).

Among a total of 80 modules loaded in Ubuntu 16.04 LTS, we can determine the exact location of 29 modules that have a unique size signature. However, the method cannot detect modules with the same size. For example, for the worst case, there were 27 modules with minimal size that have one page (`0x1000`) of **X** area and three pages (`0x3000`) of **NX** area. Despite such a limitation, this module detection can still be useful in attacks because the attacker can reduce the uncertainty of the address (from targeting 80 modules to only targeting 27 modules in the worst case).

For Windows, we can uniquely detect 97 drivers among a total of 141 drivers using size-based driver signatures. Since the kernel drivers of Windows have discardable mappings after initialization, we cannot directly use the same signature for detecting Linux kernel modules. Instead, we use two sizes that are unchanging during the lifetime of the module to build the signature: 1) the total size of the driver memory region (from start of the driver, but before the start address of the next mapped driver), and 2) the size of first contiguous executable region. Via experimentation, we observed 97 unique size signatures for detecting the driver addresses.

A prior work [27] tried a similar method for detecting kernel drivers on Windows. However, since the work could not distinguish **X** pages from **NX** pages, their detection result was far smaller (21) than ours (97). Detection of **X/NX** gives much better precision for determining the location of the drivers since it provides unique size signatures for the drivers.

4.3 DrK in Virtualized Environment

To test the feasibility of the DrK attack in virtual and cloud environments, we carried out the attack on Linux running under Xen hypervisor 4.4, as well as an X1 instance of Amazon EC2 [2] to check if the attack can be launched in such an environment.

Iterations	2	5	20	50	100
# incorrect	239	70	18	8	0
Accuracy (%)	92.22	97.72	99.41	99.74	100.0

Table 5: Achieved accuracy over the number of iterations for the modules pages (3,072 pages in total) of Linux Kernel 4.4.0 on Haswell (Xeon E3-1271 v3) processor. Probing the address with more iterations gives better accuracy. For measuring with only 2 iterations (minimum iterations to measure TLB cache hit), 239 mapped pages were detected as unmapped pages, which renders 92.22% of accuracy. With more iterations, such as 20, 50, and 100, the number of incorrectly measured pages decreases as 18, 8, and 0, respectively. With 100 iterations, the DrK attack achieves perfect accuracy on breaking KASLR.

Results. The DrK attack can detect 99.99%–100% of kernel mappings in a virtualized environment. The accuracy is slightly lower than the bare-metal result. We believe that the difference in accuracy is caused by factors that can affect TSX execution, such as virtual interrupts generated by the hypervisor (*e.g.*, `VMEXIT`).

Interestingly, we observed that a Skylake laptop processor (Core i5-6300HQ) with Xen resulted in timings that were very different from other environments (Table 4). We believe that this was due to speed throttling, *i.e.*, Intel Speed Step and Turbo Boost, because its clock rate was lower than the same processor in a bare-metal environment (2.3 GHz versus 3.0 GHz). We cover more issues with clock speed in §4.4.

4.4 Controlling the Noise

Since DrK is a cache-based timing channel attack, it is not free from the noise of the channel. On determining page mapping status and executable status, we used the minimum cycle observed when probing the pages as the threshold. However, occasionally, we observed measurement errors, possibly due to hardware characteristics (*e.g.*, cache coherence traffic and cache conflict).

We minimized such measurement errors by probing a certain address multiple times. Table 5 shows how the accuracy changes by the number of iterations. With 2 iterations, 239 mapped pages were detected as unmapped, among 3,072 pages in the scanned area (92.22% of accuracy). However, as the number of iterations increased to 5, 20, and 50, the number of mis-detected pages was drastically reduced, 70, 18, and 8, respectively. Finally, probing with 100 iteration, the accuracy reached 100%.

Note that the number of iterations to achieve the perfect accuracy depends on the environment, such as processor generation or software settings for generating interrupts that is asynchronously handled in TSX. For example, unlike Haswell that needed 100 iterations to get the 100% accuracy on Linux, Skylake only required 20 iterations to achieve perfect accuracy (see the second row on Table 4). For the virtualized environment, the measured accuracy is slightly lower than running bare-metal with the same number (100) of iterations (100% vs 99.98%). However, still, we can manage the noise by increasing the number of iterations: *e.g.*, 500 iterations achieved 100% accuracy.

Dynamic frequency scaling (*i.e.*, Intel Speed Step or Intel Turbo Boost) also affects the timings. Since we use the number of clock cycles (using `rdtscp`) to measure timing, a change in the clock frequency would affect the timing measurements [30, §17.14].

We solved this problem by keeping the processor busy. We empirically observed that running two dummy loops consuming 100% CPU time was enough to maximize the processor clock rate, and achieve perfect accuracy. This also implies that the DrK attack works well with processors with high workload; *i.e.*, when the processor runs jobs other than the attacks.

	READ		**JUMP**		
Trace point	**M**	**U**	**X**	**NX**	**U**
dTLB-loads	3,021,847	3,020,043	3,018,191	3,018,857	3,025,769
dTLB-load-misses	84	**2,000,086**	64	91	109
iTLB-loads	267	425	590	**1,000,247**	272
iTLB-load-misses	6	6	31	12	**1,000,175**
L1-icache-load-misses	1,092	1,027	1,157	1,190	1,391
L1-dcache-loads	3,021,885	3,020,081	3,018,229	3,018,895	3,025,807
L1-dcache-load-misses	1,000,856	1,000,787	1,002,456	1,000,603	1,002,539

Table 6: The number of TLB and cache loads and misses when DrK probes single kernel memory page 1,000,000 times, measured by hardware performance counters. Probing methods are reading mapped (**M**) and unmapped (**U**) pages, and jumping into executable (**X**), non-executable (**NX**), and unmapped (**U**) pages. Tracepoints are data TLB loads/misses, instruction TLB loads/misses, L1 instruction cache misses, and L1 data cache load/misses. Reading of **U** page generated data TLB misses. In addition, while jumping into **X** pages does not generate hit/miss counts on an instruction TLB (iTLB), jumping into **NX** page incurs hit on iTLB, but jumping into **U** page generated an iTLB miss. The remaining tracepoints were similar to each other.

4.5 Comparison with the Prior Attack

We summarize the evaluation result of the DrK attack by comparing it with the prior attack presented by Hund *et al.* [27].

Speed and accuracy. The DrK attack took about 0.5 seconds in Linux and about 5 seconds in Windows to achieve 100% accuracy of detecting the full mapping information of kernel and driver space. In contrast, the prior attack requires 17 seconds to obtain 96% accuracy.

Noise of the channel. Compare to the prior attack, DrK is strong against measurement noise. As shown in Figure 6, the timing difference between mapped and unmapped pages was over 10% in DrK. In contrast, in the prior attack, the timing difference between mapped and unmapped pages was only around 1% (30–50 cycles from 5,000 cycle), which can easily fluctuate.

Executable page detection. Unlike the prior attack, DrK can distinguish executable pages from non-executable pages. Our new finding is to demonstrate an accurate way of distinguishing executable pages from the mapped pages, which allows an attacker to effectively break KASLR, especially when detecting the exact location of kernel drivers.

Module detection. As mentioned in §4.2.3, DrK detected a larger number of drivers (97) than the prior attack (5–21). This is because DrK has better accuracy than the prior attack and DrK is able to use not only size signatures but also executable mapping status.

5. IN-DEPTH ANALYSIS OF DrK

In this section we figure out what causes DrK observe timing differences. Since the internal characteristics of Intel processors are barely documented, we first take a look into detailed measurements on hardware events during the DrK attack using the *hardware performance counter (HPC)* (§5.1). Then, we analyzed the measurement results to figure out what causes the time difference between mapped and unmapped pages (§5.2), and between executable and non-executable pages (§5.3), in accordance with a simplified Intel CPU architecture diagram (Figure 8).

5.1 Measuring Hardware Events

We probed a memory page 1,000,000 times while measuring hardware events using the HPC. We tested the following five of memory probings:

1. read a mapped kernel page
2. read an unmapped kernel page
3. jump into an executable kernel page
4. jump into a non-executable kernel page
5. jump into an unmapped kernel page.

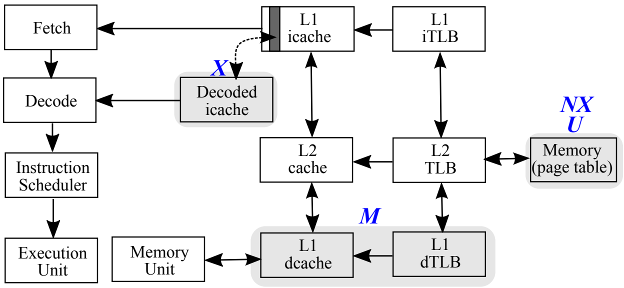


Figure 8: Simplified Intel Skylake architecture including pipeline and cache hierarchy. We omitted L2, last level, and any other page table caches for simplicity. Each **M**, **U**, **X**, and **NX** indicates the locations where page faults are checked according to our measurements. Note that decoded icache, which caches decoded micro-ops, is inside L1 icache [30, 49] and it is a virtually-indexed and virtually tagged (VIVT) cache, which does not requires an iTLB access for address translation.

On probing, we set tracepoints to measure:

1. `dTLB-loads`: the total number of attempts to load data translation lookaside buffer (dTLB), including cache hit on dTLB.
2. `dTLB-load-misses`: the number of failed attempts to load dTLB; cache miss on dTLB.
3. `iTLB-loads`: the total number of attempts to load instruction TLB (iTLB), including cache hit on iTLB.
4. `iTLB-load-misses`: the number of failed attempts to load iTLB; cache miss on iTLB.
5. `L1-icache-load-misses`: the number of failed attempts to load L1 instruction cache (icache), cache miss on icache.
6. `L1-dcache-loads`: the total number of attempts to load L1 data cache (dcache), including dcache hit.
7. `L1-dcache-load-misses`: the number of failed attempts to load L1 dcache; dcache miss.

Note that we were unable to measure `L1-icache-loads` because all tested CPUs that support TSX (listed in Table 3) did not support the counter for it. Table 6 summarizes the results measured in the Linux machine with Intel Core i7-6700K (Skylake) 4.0 GHz CPU.

5.2 Mapped versus Unmapped Pages

We conjecture that *dTLB* makes timings for mapped and unmapped kernel pages differently, by its different caching behavior on page mappings. As introduced in §3 and §4, a TSX abort handler for a violation of reading mapped kernel memory (**M**) was called faster than that for a violation of reading unmapped kernel memory (**U**). We believe that the timing difference is originated by dTLB hit on accessing a mapped memory (fast), and dTLB miss for an unmapped memory (slow, because it requires a page table walk).

Table 6 supports this hypothesis: while reading an unmapped (**U**) page generated a lot of (over two millions) dTLB misses (`dTLB-load-misses`, marked in red), accessing a mapped (**M**) page did not. The counter shows that the access to a mapped kernel memory caches its page table entry in dTLB. On the subsequent probing, the page table entry can be retrieved by just accessing the dTLB. Then, the page privilege is checked without having page table walk, thus it results in faster determination of the page fault exception. In contrast, the access to an unmapped kernel memory cannot be cached into dTLB, because it has no corresponding physical address. After suffering dTLB-miss, the probing should wait until the processor seek the page table entry then figuring out the page fault. Thus, a read attempt to an unmapped page always requires a page table walk, so it takes longer to raise the page fault exception (and calling TSX abort handler).

We note that this explanation corresponded to Hund *et al.* [27]'s finding.

5.3 Executable versus Non-executable Pages

We conjecture that *decoded icache* is the hardware components that creates the side channel by handling executable and non-executable kernel pages differently, as shown in timing differences. In the DrK attack, timing of probing executable (**X**) pages was measured as faster than that of non-executable (**NX**) and unmapped (**U**) pages. While it seems caching in iTLB would generate such timing difference, nonetheless, it is not true.

Table 6 supports the hypothesis as follows.

iTLB is not the origin. Unlike the result on mapped/unmapped pages, jumping into an **X** page did not generate any additional iTLB-loads (590 in iTLB-loads on 1 M accesses). This means that iTLB hit did not even happen on probing. Moreover, although iTLB actually hits on probing **NX** pages (only 12 misses on 1 M accesses), the timing of **NX** pages is not as fast as **X**. Further, although probing on **U** pages generates many iTLB misses, the timing of **NX** pages and **U** was the same (see Table 3). This proves that iTLB is not the origin of the timing channel.

Decoded icache for faster timing on X. Table 6 also implies that, unlike executing on **NX** and **U** pages, probing **X** pages does not need to translate the virtual address to the physical address to fetch instructions. From this observation, we came up with a hypothesis that decoded icache is the origin of the timing channel; decoded icache is the place in which fetched and decoded micro-ops are cached (Figure 8). And it is virtually-indexed, and virtually-tagged (VIVT) cache, which does not require address translation on its access according to a patent document [49] published by Intel Corporation.

On probing an executable memory page, once the instructions in the address are decoded and cached, the processor no longer needs to translate the address. Instead, it directly fetches the instructions from the decoded icache, however, the actual execution fails due to the access violation. Since the probing does not go through iTLB nor page table walk, the exception is generated relatively faster than probing other page mappings.

Page table walk is required for both NX and U. We conjecture that page faults for both **NX** and **U** pages are generated after the processor finished the page table walk, since their timings are the same (Table 3). And, we hypothesize missing of coherency mechanism in TLB in Intel processor makes those two access must go through the page table walk.

On probing non-executable (**NX**) pages, the processor will cache the corresponding page table entry into iTLB, and the entry is marked as non-executable. However, the processor cannot determine the page fault by just inspecting the NX bit on the page table entry from iTLB since TLB is not a coherent cache in Intel architecture [24]. That is, there could be a case that other cores have updated the permission on the page table as executable, but the current core has a mismatched page table entry in its TLB. In such a case, on (trying) execution on the page, the processor is required to resolve the latest page table entry from the page table to check if there has been any update. Therefore, on the subsequent probing, despite iTLB hits, but the entry set with the NX bit; then, the processor walks through the page table, and get the results that latest entry: the entry is set with the NX bit. Finally, the processor can generates a page fault exception.

U pages take the same path as the mapped versus unmapped case. After the iTLB miss, the processor walks through the page table, figures out the address is not mapped, and generates a page fault.

In summary, while executable pages (**X**) do not walk through the page table on the subsequent probing (fast), both mapped (**M**) and unmapped (**U**) pages always make processor walks through the page table (slow) to determine the page fault.

6. POTENTIAL COUNTERMEASURES

We discuss potential countermeasures against the DrK attack. In summary, we see there are no effective countermeasures that can prevent DrK without hurting usability and performance.

6.1 Eliminating Timing Channel

One of the most fundamental countermeasures against DrK is eliminating any timing differences when probing the kernel addresses at the user-level execution. When the Intel CPU handles exceptions for unprivileged accesses, it takes different hardware paths according to whether the accessed memory region is executable, mapped, or unmapped. §5 shows our measurements about the differences in terms of the loads and misses of cache and TLB.

Hardware modification. One possible way to flatten the timing differences is modifying the hardware. For example, we can change the CPU to not cache unprivileged accesses to kernel addresses, which makes every unprivileged memory access take the same hardware path. Any timing channel attacks, including DrK and Hund *et al.* [27], cannot observe timing differences and would therefore fail. However, since this solution demands hardware modifications, it cannot protect already deployed CPUs from DrK.

Kernel page table separation. Another countermeasure against DrK is separating the kernel page table from a user page table. If no kernel memory is mapped into a user page table, DrK would have no chance to break KASLR. However, the kernel address space is mapped into a user page table to minimize the execution overhead of privileged instructions (*e.g.*, system call). Without this, the system would need to flush TLBs whenever a user process invokes a system call, which would significantly reduce the overall system performance [6, 27]. For example, Xen uses separated page tables for kernel and user processes when a guest machine is a 64-bit para-virtualized machine (PVM), but this degrades system performance such that the 64-bit PVM is not recommended [46] for use. Therefore, It is impractical to adopt this countermeasure due to performance degradation.

6.2 Monitoring Hardware Events

Modern CPUs provide HPC interfaces to monitor hardware events that occur inside a CPU, such as the numbers of retired instructions, taken branches, and cache loads/misses. Although the OS cannot observe any page faults while being attacked by DrK, it can infer whether DrK is being performed by using the hardware event information. For example, we identified that during the DrK attack, a CPU generates lots of loads/misses on both TLBs and the L1 i-cache (see Table 6). Further, the HPC provides information about the number of transactional aborts (`tx-abort`) generated by a CPU. This number would be large when the system is under attack by DrK because each memory probing generates an abort.

However, monitoring hardware events has limitations to detect DrK in terms of accuracy and performance. First, a benign program can show a similar behavior to DrK when it randomly accesses different memory regions or heavily uses TSX. Further, DrK can cloak its behavior by decreasing the frequency of memory probing. Thus, the system cannot avoid false detection problems. Next, monitoring all processes of the OS is unrealistic due to the overhead of checking HPC: about 20% performance overhead [60]. System-wide monitoring could reduce the overhead, but, in that case, it is difficult to determine which process engages in suspicious behavior.

6.3 Live Re-randomization and Fine-grained KASLR

One possible approach to cope with DrK is to use fine-grained ASLR and live re-randomization [21]. They not only adjust the base addresses of kernel code and modules, but also randomize all kernel code, static data, stack, dynamic data, and modules while re-randomizing them periodically.

However, as shown in Table 4, DrK took less than 0.2 seconds in Linux to detect full page mappings with 100% accuracy. Moreover, by detecting a single module rather than the whole space, the attack can be even faster (proportional to the scan size). This implies that the OS needs to re-randomize its address space more than once per second, which is problematic due to performance overhead.

In case of fine-grained KASLR, adding base offset within a page can defeat DrK since the finest granularity that DrK can detect the mapping is a page. However, this would have implementation challenges, e.g., a compatibility issue on misaligned pages.

6.4 Other Countermeasures

Lastly, we introduce a few mechanisms that can prevent or mitigate DrK, but these are less practical.

Disabling TSX. The strongest countermeasure against DrK, though a naïve one, is disabling TSX. Unlike other instructions that can be disabled through model-specific registers (MSRs) or BIOS settings (e.g. VM extensions), TSX cannot be disabled in such manner. However, as a CPU manufacturer, Intel can disable the feature via a microcode update or product line change. In fact, Intel had already disabled TSX in the Haswell CPU due to a hardware bug (Erratum HSD136 in [29]). Nonetheless, this solution is problematic because TSX is already widely used; *e.g.*, `glibc` uses TSX in its `pthread` library for synchronization and Java uses TSX for thread scheduling [34, 45].

Different caching policy. TSX only works with memory regions configured as *write-back* ([30, §15.3.8.2]), which is a default configuration due to its efficiency. In our experiment, we observed a few (288 pages among 26,066 pages in Windows) memory pages in driver area configured as write-through or uncacheable and DrK misjudged them as unmapped pages. Although those pages do not belong to any kernel drivers, (*i.e.*, it does not affect the accuracy evaluation) this implies that if an OS makes the entire kernel memory either write-through or uncachable, it can be secure against DrK. However, this configuration is impractical, as it results in huge performance degradation [6].

Noisy timer. Finally, a noisy timer or coarse-grained timer is a well-known countermeasure against timing attacks. The system can add some noise when returning a timer value or prevent a user program from using a fine-grained timer (*e.g.*, `rdtsc`). However, the noisy timer is just a work-around such that it cannot completely prevent DrK. Also, many benign programs need to use the fine-grained timer, *e.g.*, to precisely measure performance.

7. DISCUSSIONS

Limitations. DrK has some limitations. First, DrK always treats uncachable, write-through, and paged-out memory regions as unmapped. DrK relies on Intel TSX, which only works with memory pages configured as write-back ([30, §15.3.8.2]). Thus, it cannot probe memory pages configured as uncachable or write-through; its access to such memory pages always aborts. DrK also treats swapped-out pages as unmapped because access to such pages generates a page fault which aborts the transaction ([30, §15.3.8.2]).

However, we did not observe write-through pages for the code and data areas of the kernel and drivers, in both Linux and Windows, because write-through pages are slower than write-back pages. Due to the performance issues, kernel developers do not use write-through pages in general. Further, most of the important kernel pages, such as kernel text, data, and drivers, are frequently used, so they are usually kept in memory (*i.e.*, not paged out). Therefore, these limitations are negligible.

Breaking "security by memory obscurity". DrK can also be applied to launch an undetectable, crash-resistant memory mapping probing [19]. Some system protection mechanisms, such as CPI [35], ASLR-Guard [39], and Kenali [54], assume a secret memory location that the attacker cannot know to store sensitive information for integrity protection. However, using DrK, such an assumption could be broken if such a secret address is not selected carefully, because the DrK attack can fully search for the address space without any crash. We plan to figure out how we can use DrK to break such protection mechanisms.

8. RELATED WORK

In this section, we provide a comprehensive landscape of past research related to the DrK attack.

ASLR: Attacks and defenses. Since the most modern OSes adapt W⊕X and ASLR to prevent code injection and code reuse attacks [12, 48, 51, 52], attackers and defenders continuously find new attacks to break ASLR and develop countermeasures against them. Researchers find that many ASLR implementations are insecure because they do not fully randomize address spaces (*e.g.*, shared libraries without ASLR and fixed memory allocation) and do not provide enough entropy (*e.g.*, limited mapping range and large alignment size) to avoid performance degradation. These make the ASLR implementations vulnerable to prediction and brute-force attacks [9, 12, 18, 51, 52]. To prevent such attacks, researchers propose fine-grained ASLR technologies that randomize the location of functions [7, 33], basic blocks [59], and even instructions and registers [26, 47].

On the other hand, researchers also discover that even fine-grained ASLR can be broken by information leak vulnerabilities [50, 55], since they let attackers know de-randomized addresses. By using it, Snow *et al.* [53] break the fine-grained ASLR. To mitigate such an attack, three kinds of schemes have been proposed: (1) dynamic (re-)randomization [4, 8, 16, 20, 21, 38] to make leaked information useless, (2) execution-only memory [3, 11, 15] and destructive code read [56] to prevent attackers from reading any code gadgets, and (3) pointer integrity [14, 35, 39] to prevent code pointer manipulation.

In addition, researchers recently found that memory deduplication can be used to break ASLR without information leak vulnerabilities [5, 10].

Timing attacks against KASLR. Hund *et al.* [27] present a timing side channel attack against kernel space ASLR, which is the work most relevant to DrK. The main advantage of these timing attacks over the previous ASLR attacks is that they neither relies on weak implementations of ASLR nor information leak vulnerabilities.

Hund *et al.*'s attack and DrK focus on similar timing differences caused by how a processor handles a page fault for mapped compared to unmapped kernel memory pages. However, unlike DrK, Hund *et al.*'s attack should call the OS page fault handler whenever probing each kernel memory page, which suffers from high noise due to a long execution path inside the OS. Furthermore, this lets the OS know which user process frequently accesses kernel memory pages, so that the OS can easily detect the attack.

In contrast, DrK uses a TSX abort handler to probe a kernel memory page, whose execution path is shorter than that of OS page fault handler, making it less error-prone and even able to recognize the small difference between accessing executable and non-executable pages. Also, it is difficult for the OS to detect DrK because it cannot directly observe the behavior of DrK (§4).

Recently, Gruss *et al.* [22] exploit the prefetch instruction on the processor, which loads a specific address into a certain cache level, to probe mapping information without causing exceptions. However, since the prefetch instruction targets data, this attack cannot identify whether an address is executable or non-executable, unlike DrK.

Crash-resistant memory probing. One of the advantages of DrK is that it does not generate a crash when probing the kernel's address space. Recently, Gawlik *et al.* [19] have shown a similar web attack for crash-resistant memory probing. They found that memory access violations by some JavaScript methods do not crash modern web browsers having fault-tolerant functionality, which allows for memory probing without a browser crash. However, OSes can identify whether such an attack is performed because it cannot suppress the exception, unlike DrK. Also, as the authors mention, this attack can be mitigated by limiting the number of faults that can be caused, checking the exception information, using guard pages, and using memory safety solutions. However, none of these approaches can mitigate DrK.

TSX timing channel. We found two blog articles [1, 64] that depicted kernel timing attacks using TSX while we conducted this research. Note that although this paper and the two blog articles are based on similar observations, this work makes the following important contributions, unlike the blog articles which only conjecture that such an attack is possible. First, we did comprehensive evaluations. We demonstrated and analyzed DrK with three different Intel CPU generations (§4.1) in all major OSes (§4.2 and §4.2.3). Moreover, we give instructions on controlling the noise of timing channel to get the best precision (§4.4). Second, we showed what causes this timing channel through experiments. We studied the architecture of the modern Intel CPU in depth and discovered which execution paths lead to such a timing channel (§5). We monitored the behavior of the Intel CPU in detail using the HPC and checked its architectural details to figure out the root cause. Lastly, we discovered that the TSX timing channel can be used to determine whether a memory page is executable or non-executable (§3 and §4). Note that neither the two blog articles nor Hund *et al.* [27]'s work discovered this timing channel.

9. CONCLUSION

To protect the kernel memory from attacks in the wild, commodity OSes have adopted KASLR, which is proven to be a practical defense mechanism against many memory corruption attacks. In this paper, we introduced DrK, a timing side channel attack that almost perfectly de-randomizes KASLR using the Intel CPU's new instruction set, TSX. Our evaluation shows DrK is much better than the prior side channel attack in terms of precision, platform independence, covertness, and speed. We further analyzed which architectural characteristics exposed such timing differences and proposed some countermeasures to eliminate it.

Acknowledgments

We thank the anonymous reviewers, for their helpful feedback, as well as GTISC lab members for their proofreading efforts. This research was supported by the NSF award DGE-1500084, CNS-1563848, CRI-1629851 ONR under grant N000141512162, DARPA TC program under contract No. DARPA FA8650-15-C-7556, and DARPA XD3 program under contract No. DARPA HR0011-16-C-0059, and ETRI MSIP/IITP[B0101-15-0644].

Responsible vulnerability disclosure. Following the guidance of responsible vulnerability disclosure, We confidentially reported the vulnerability to through US-CERT (VU#954695) and Microsoft Security Response Center (MSRC, Case 32737, TRK:0001003139), and shared this manuscript with affected vendors to resolve the newly discovered security threat. After the public disclosure, we will release the source code of the DrK attack available to the public.

References

[1] Anababa. What Does Transactional Synchronization Extensions (TSX) Processor Technology Mean to Vulnerability Exploits (e.g. Brute Forcing)?,. http://hypervsir.blogspot.com/2014/11/what-does-transactional-synchronization.html.

[2] AWS Blog. Amazon EC2 X1 Instances. https://aws.amazon.com/ec2/instance-types/x1/.

[3] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny. You Can Run but You Can't Read: Preventing Disclosure Exploits in Executable Code. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, Nov. 2014.

[4] M. Backes and S. Nürnberger. Oxymoron: Making Fine-Grained Memory Randomization Practical by Allowing Code Sharing. In *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[5] A. Barresi, K. Razavi, M. Payer, and T. R. Gross. CAIN: Silently breaking ASLR in the cloud. In *9th USENIX Workshop on Offensive Technologies (WOOT)*, Washington, D.C., Aug. 2015.

[6] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU caching on byte-addressable non-volatile memory programming, 2012.

[7] S. Bhatkar, R. Sekar, and D. C. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2005.

[8] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okhravi. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[9] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh. Hacking Blind. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[10] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[11] K. Braden, S. Crane, L. Davi, M. Franz, P. Larsen, C. Liebchen, and A.-R. Sadeghi. Leakage-Resilient Layout Randomization for Mobile Devices. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[12] S. Checkoway, L. Davi, A. Dmitrienko, A.-R. Sadeghi, H. Shacham, and M. Winandy. Return-Oriented Programming without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, IL, Oct. 2010.

[13] R. Chen. Some remarks on VirtualAlloc and MEM_LARGE_PAGES. https://blogs.msdn.microsoft.com/oldnewthing/20110128-00/?p=11643.

[14] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard^{TM}: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[15] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz. Readactor: Practical Code Randomization Resilient to Memory Disclosure. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[16] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z. Snow, and F. Monrose. Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[17] S. Esser. mach_port_kobject() and the Kernel Address Obfuscation. https://sektioneins.de/en/blog/14-12-23-mach_port_kobject.html.

[18] I. Evans, S. Fingeret, J. Gonzalez, U. Otgonbaatar, T. Tang, H. Shrobe, S. Sidiroglou-Douskos, M. Rinard, and H. Okhravi. Missing the Point(er): On the Effectiveness of Code Pointer Integrity. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[19] R. Gawlik, B. Kollenda, P. Koppe, B. Garmany, and T. Holz. Enabling Client-Side Crash-Resistance to Overcome Diversification and Information Hiding. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[20] J. Gionta, W. Enck, and P. Ning. HideM: Protecting the Contents of Userspace Memory in the Face of Disclosure Vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[21] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[22] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[23] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang. Protecting Private Keys against Memory Disclosure Attacks using Hardware Transactional Memory. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[24] Henry. TLB and Pagewalk Coherence in x86 Processors. http://blog.stuffedcow.net/2015/08/pagewalk-coherence/.

[25] Heroku. Heroku: Cloud Application Platform. https://www.heroku.com/.

[26] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd My Gadgets Go? In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[27] R. Hund, C. Willems, and T. Holz. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[28] Intel. ARK | Your Source for Intel Protect Specifications. http://ark.intel.com.

[29] Intel Corporation. *Desktop 4th Generation Intel CoreTM Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family*, 2015.

[30] Intel Corporation. *Intel 64 and IA-32 Architectures Developer's Manual*, 2015.

[31] K. Johnson and M. Miller. Exploit Mitigation Improvements in Windows 8. In *Black Hat USA*, 2012.

[32] D. Keuper. XNU: a security evaluation. 2012.

[33] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, Chicago, IL, Dec. 2006.

[34] A. Kleen. Lock elision in the GNU C library, 2013. https://lwn.net/Articles/534758/.

[35] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proceedings of the 11th Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[36] V. Leis, A. Kemper, and T. Neumann. Exploiting Hardware Transactional Memory in Main-Memory Databases. In *Proceedings of the 30th IEEE International Conference on Data Engineering Workshop*, Chicago, IL, Mar.–Apr. 2014.

[37] Y. Liu, Y. Xia, H. Guan, B. Zang, and H. Chen. Concurrent and Consistent Virtual Machine Introspection with Hardware Transactional Memory. In *Proceedings of the 20th IEEE Symposium on High Performance Computer Architecture (HPCA)*, Orlando, FL, USA, Feb. 2014.

[38] K. Lu, S. Nurnberger, M. Backes, and W. Lee. How to Make ASLR Win the Clone Wars: Runtime Re-Randomization. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[39] K. Lu, C. Song, B. Lee, S. P. Chung, T. Kim, and W. Lee. ASLR-Guard: Stopping Address Space Leakage for Code Reuse Attacks. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[40] MITRE Corporation. CVE-2015-1097. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1097.

[41] MITRE Corporation. CVE-2015-1674. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-1674.

[42] MITRE Corporation. CVE-2015-8569. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-8569.

[43] MITRE Corporation. CVE-2016-0175. http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-0175.

[44] NES CONSEIL. Bypassing Windows 7 Kernel ASLR. https://dl.packetstormsecurity.net/papers/bypass/NES-BypassWin7KernelAslr.pdf.

[45] Oracle. Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/8/docs/technotes/tools/unix/java.html.

[46] Oracle. Oracle VM Performance and Tuning - Part 5. https://blogs.oracle.com/jsavit/entry/oracle_vm_performance_and_tuning4.

[47] V. Pappas, M. Polychronakis, and A. D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-Place Code Randomization. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[48] PaX Team. PaX address space layout randomization (ASLR), 2003. https://pax.grsecurity.net/docs/aslr.txt.

[49] L. Rappoport, C. Koren, F. Sala, O. Lempel, I. Ouziel, I. Kim, R. Gabor, L. Libis, and G. Pribush. Method and Apparatus for Pipeline Inclusion and Instruction Restarts in a Micro-op Cache of a Processor, June 2010. US Patent App. 12/317,959.

[50] F. J. Serna. The info leak era on software exploitation. In *Blackhat USA*, 2012.

[51] H. Shacham. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct.–Nov. 2007.

[52] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS)*, Washington, DC, Oct. 2004.

[53] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[54] C. Song, B. Lee, K. Lu, W. R. Harris, T. Kim, and W. Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[55] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter. Breaking the Memory Secrecy Assumption. In *Proceedings of the Second European Workshop on System Security (EUROSEC)*, 2009.

[56] A. Tang, S. Sethumadhavan, and S. Stolfo. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[57] The Linux Kernel Archives. Huge Pages. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt.

[58] Z. Wang, H. Qian, J. Li, and H. Chen. Using Restricted Transactional Memory to Build a Scalable In-Memory Database. In *Proceedings of the ACM EuroSys Conference*, Amsterdam, The Netherlands, Apr. 2014.

[59] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[60] V. Weaver. Linux perf event Features and Overhead, 2013. http://researcher.watson.ibm.com/researcher/files/us-ajvega/FastPath_Weaver_Talk.pdf.

[61] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast In-memory Transaction Processing using RDMA and HTM. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, Oct. 2015.

[62] Wikiwand. Address space layout randomization. http://www.wikiwand.com/en/Address_space_layout_randomization.

[63] Windows Dev Center. Creating a File Mapping Using Large Pages. https://msdn.microsoft.com/en-us/library/windows/desktop/aa366543(v=vs.85).aspx.

[64] R. Wojtczuk. TSX Improves Timing Attacks Against KASLR. https://labs.bromium.com/2014/10/27/tsx-improves-timing-attacks-against-kaslr/.