electronics

Article

Breaking KASLR Using Memory Deduplication in Virtualized Environments

Taehun Kim [1], Taehyun Kim [2] and Youngjoo Shin [1,*]

[1] School of Cybersecurity, Korea University, Seoul 02841, Korea; taehunk.syoungjoo@korea.ac.kr
[2] School of Computer and Information Engineering, Kwangwoon University, Seoul 01897, Korea; taehun9203@gmail.com
* Correspondence: syoungjoo@korea.ac.kr

Abstract: Recent operating systems (OSs) have adopted a defense mechanism called kernel page table isolation (KPTI) for protecting the kernel from all attacks that break the kernel address space layout randomization (KASLR) using various side-channel analysis techniques. In this paper, we demonstrate that KASLR can still be broken, even with the latest OSs where KPTI is applied. In particular, we present a novel memory-sharing-based side-channel attack that breaks the KASLR on KPTI-enabled Linux virtual machines. The proposed attack leverages the memory deduplication feature on a hypervisor, which provides a timing channel for inferring secret information regarding the victim. By conducting experiments on KVM and VMware ESXi, we show that the proposed attack can obtain the kernel address within a short amount of time. We also present several countermeasures that can prevent such an attack.

Keywords: KASLR; side-channel attack; memory deduplication

check for updates

Citation: Kim, T.; Kim, T.; Shin, Y. Breaking KASLR Using Memory Deduplication in Virtualized Environments. *Electronics* **2021**, *10*, 2174. https://doi.org/10.3390/electronics10172174

Academic Editor: Stefanos Kollias

Received: 6 August 2021
Accepted: 3 September 2021
Published: 6 September 2021

1. Introduction

Operating systems protect their kernel from code reuse [1,2] attacks such as return-oriented programming (ROP) [3–6] by using kernel address space layout randomization (KASLR). However, KASLR is vulnerable to CPU side-channel attacks such as cache timing attacks [7,8], transient execution attacks [9], and other attacks exploiting a processor optimization technique (i.e., Intel TSX). The common principle behind these side-channel attacks is to break the KASLR using a feature in which user and kernel address spaces are mapped to the same page table. The Linux and Windows OSs adopted KPTI [10] as a defense mechanism against CPU side-channel attacks. KPTI protects the KASLR-enabled kernel from CPU side-channel attacks by separately allocating a page table of the user and kernel address spaces.

In this paper, we demonstrate that the KPTI defense mechanism does not guarantee full protection to KALSR. More specifically, the KASLR of the victim VM running the latest KPTI-enabled OS is still vulnerable to a memory deduplication attack. The memory deduplication attack was first introduced by Suzaki et al. [11]. It exploits a page sharing feature of hypervisors to infer the page content of other VMs. Our attack further extends the memory deduplication attack to break the KASLR for state-of-the-art secure kernels in a virtualized environment.

We present the details of our attack that breaks KASLR in the latest versions of Linux kernels. As KASLR randomizes the memory mappings of kernel pages, kernel symbols, such as functions and global variables, have to be relocated accordingly. This means that the content of the kernel pages should differ for every instance of the memory mapping. If an attacker might be able to know the page content, he/she can infer the base address determined by KASLR. The basic idea of the proposed attack is to find out the page content by leveraging the memory deduplication attack.

In order to successfully achieve the proposed attack, we have to address two challenging problems. The first one is to find the CHECKDEDUP function, which allows us to check whether a given candidate page p is the right one or not. The second challenging problem is to deal with the complexity of guessing in the case where certain kernel pages have a large entropy (i.e., the amount of uncertainty in the page content). We present our solutions for those problems in detail.

We also present a performance evaluation of the proposed attack under various experimental settings. Specifically, the experiments were conducted under KVM [12] and VMware ESXi [13,14] environments, both of which are hypervisors that support memory deduplication. The evaluation results show that a spy can infer information of the kernel address of the victim's VM within at least 12 min on a KVM hypervisor. To mitigate the proposed attack, we present some possible countermeasures, such as disabling the memory deduplication on a hypervisor.

The contributions of this paper are as follows:

1. We present a novel VM-based side-channel attack that exploits a deduplication technique of hypervisors. The main idea of the proposed attack is to infer the sensitive information from a timing difference of write access latency between a duplicate and deduplicated page. The proposed attack breaks KALSR in the latest versions of the Linux kernel equipped with the state-of-the-art kernel defense mechanism.
2. We evaluate the proposed attack by conducting extensive experiments under real virtualization environments where practical hypervisors such as KVM and VMware ESXi are used. The evaluation results support the feasibility and effectiveness of the proposed attack.

The remainder of this paper is organized as follows: In Section 2, we introduce some background knowledge of KASLR and memory deduplication attacks. In Section 3, we present the proposed KASLR-breaking attack in detail. In Sections 4 and 5, we present several countermeasures against the attack and previous related studies, respectively. Finally, we provide some concluding remarks in Section 6.

2. Background

In this section, we present some background knowledge regarding KASLR and a memory deduplication attack.

2.1. Kernel Address Space Layout Randomization (KASLR)

Address space layout randomization (ASLR) is a defense mechanism that protects the memory from code reuse attacks such as ROP in a user process. More specifically, whenever programs are loaded in memory, the ASLR generates a newly randomized base address in various sections of the process, such as a stack, heap, and shared library. This makes delivering code reuse attacks more challenging because the addresses of the ROP devices are randomized.

Meanwhile, recent code reuse attacks against kernel memory also introduce the necessity of KASLR, which is a kernel protection technique that applies ASLR to the kernel memory. In general, a Linux kernel consists of a kernel text and kernel modules. A main executable of the kernel is located at the kernel text, whereas loadable kernel modules, such as device drivers, are located at kernel modules. In Linux, KASLR is available to both the kernel text and kernel modules. Their base addresses are randomly generated with different levels of entropy. For instance, in recent versions of Linux (See Table 1), the kernel text and kernel module are randomized with 9-bit and 10-bit entropy, respectively. With such entropy, their base addresses are determined using the following equation:

$$\textit{base address} = \textit{start address} + s \times \textit{Page granularity}\ (0 \le s < \#\textit{slots}), \tag{1}$$

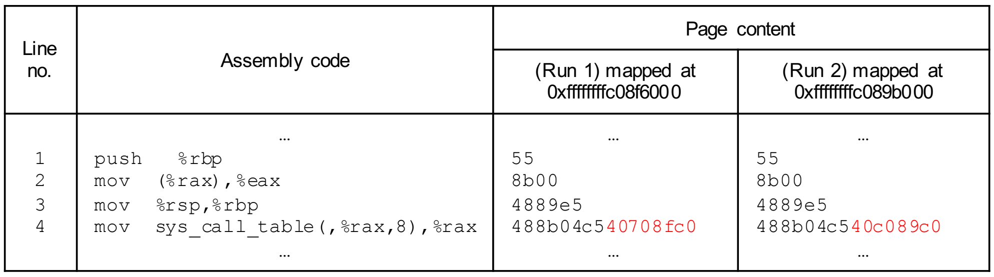
where the *start address* is the start address within the address range where the kernel text (and kernel module) can be placed.

Table 1. KASLR implementation using Linux.

OS	Types	Entropy	#slot	Address Range	Granularity
Ubuntu	Kernel Text	9 bit	512	0xffffffff80000000 ∼0xffffffffc0000000	2 MB
18.04	Kernel Module	10 bit	1024	0xffffffffc0000000 ∼0xffffffffc0400000	4 KB

For a page in the kernel text section, it is mapped into a kernel address aligned to a 2 MB boundary (i.e., *Page granularity* = 2 MB), while a page in the kernel module is mapped into an address aligned to a 4 KB boundary (i.e., *Page granularity* = 4 KB).

Once the base address has been determined by KASLR, the kernel text and kernel modules have to be relocated in the memory according to the base address. In particular, kernel symbols, such as functions and global variables, should be relocated with respect to the base address. This leads to the variance in the content of kernel pages belonging to the kernel text (or kernel modules). Figure 1 demonstrates an example of the variation in the content of a kernel page. An assembly instruction in the kernel text at Line 4 contains a kernel symbol `sys_call_table`. Two KASLR instances (i.e., Run 1 and Run 2 in the figure) result in two different addresses for the symbol, which makes a different machine code for the assembly instruction at Line 4. In short, KASLR introduces the variance in the content of kernel pages, and the content actually depends on the base address. This is the key idea of the proposed attack; if an attacker might be able to know the content of the kernel page, he/she can infer the base address determined by KASLR.



Line no.	Assembly code	Page content	
		(Run 1) mapped at 0xffffffffc08f6000	(Run 2) mapped at 0xffffffffc089b000
	...	...	...
1	`push %rbp`	55	55
2	`mov (%rax),%eax`	8b00	8b00
3	`mov %rsp,%rbp`	4889e5	4889e5
4	`mov sys_call_table(,%rax,8),%rax`	488b04c540708fc0	488b04c540c089c0
	...	...	...

Figure 1. An example of the content of a kernel page.

2.2. Memory Deduplication Attack

A memory deduplication attack [11,15,16] is a type of memory disclosure attack that allows an attacker to infer the content in the victim's memory. This attack exploits a Copy-on-Write (CoW) mechanism employed in hypervisors, such as KVM and VMWare ESXi, to enable memory deduplication (Figure 2). That is, to achieve memory saving, the hypervisor repeatedly looks for identical memory pages and merges them into a single page. The merged page is set to read-only, such that when a write occurs to the page, it is then duplicated back into separate pages. This may provide timing information to an attacker. Specifically, the attacker is able to infer whether the target page has been merged by observing the write latency. A memory deduplication attack leverages this timing information to infer the content of the target page.

3. Proposed Attack Technique

In this section, we present our proposed attack. Our attack attempts to break the KASLR of other VMs by reconstructing a memory deduplication attack in a virtualization environment.

In our attack model, we suppose that the spy and victim VM are co-located on the same host. Both VMs run the latest version of Linux, where the KPTI is enabled by default. We also suppose that the host runs a hypervisor that provides a memory deduplication technique.

As Linux applies KASLR to its kernel base and kernel modules independently, we demonstrate two attacks: one for breaking the KASLR in the kernel base and the other for breaking the KASLR in the kernel module. Figure 3 illustrates an overview of the proposed attack technique.

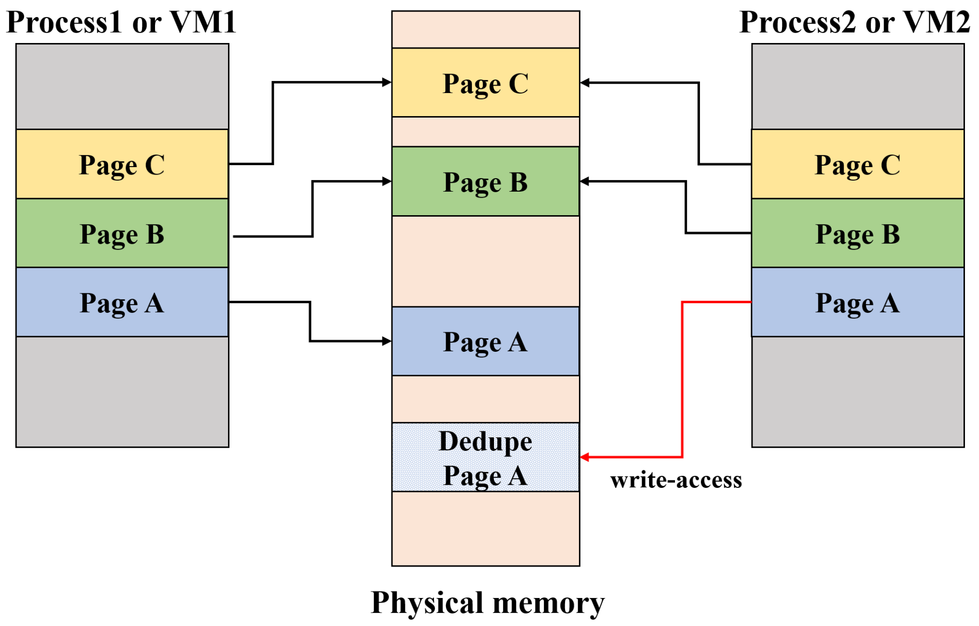


Figure 2. Copy-on-Write (CoW) mechanism.

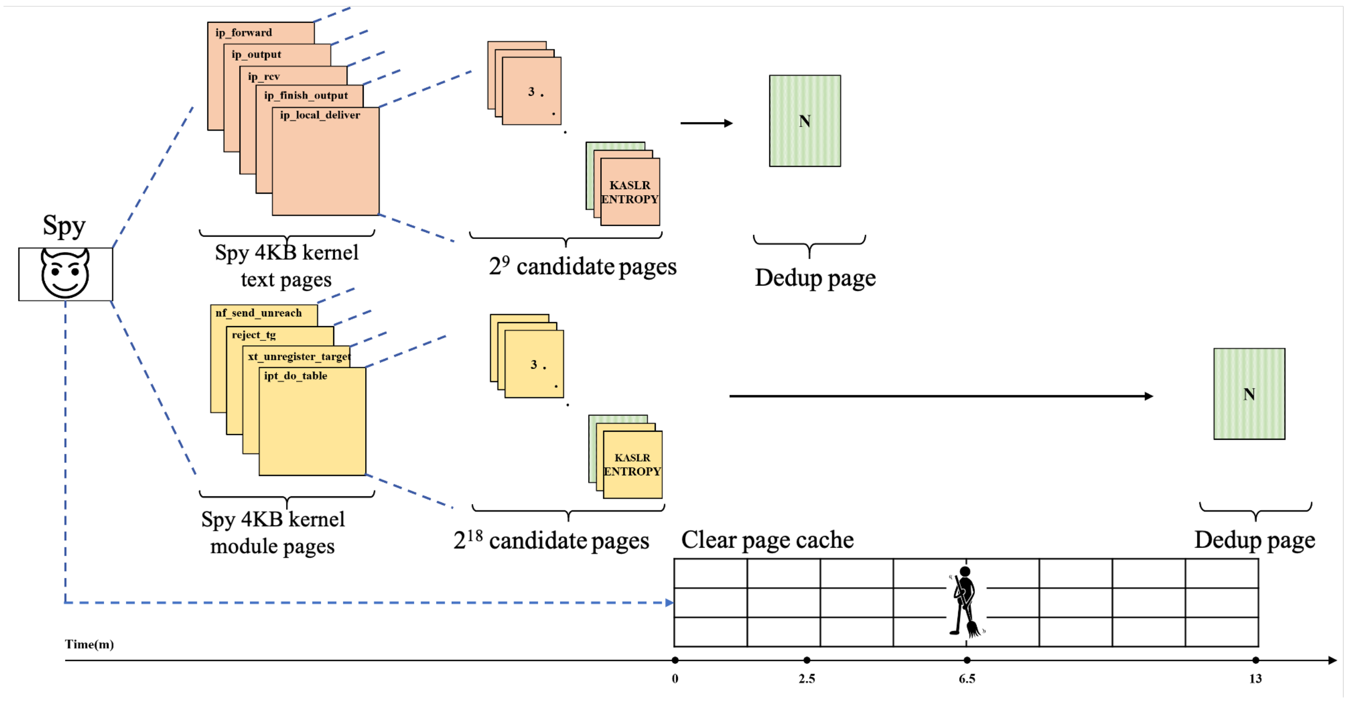


Figure 3. Proposed attack technique.

Because KASLR randomizes the mappings of kernel pages into a virtual address space, kernel symbols, such as functions and global variables, in the pages have different addresses. This means that whenever the kernel is being loaded to memory, its text segment has to be relocated accordingly to ensure that the kernel symbols are correctly referenced. Thus, the contents of the kernel pages differ for every instance of the memory mapping.

The basic idea of the proposed technique is to exploit the uncertainty contained in the victim's kernel page, which is caused by KASLR. That is, we infer the randomized base address by guessing the unknown part of the page with the memory deduplication attack.

The proposed attack technique is as follows: Suppose that the entropy (i.e., the amount of the unknown part) of the target kernel page is e, and we have a function CHECKDEDUP(p) that allows us to know whether a given memory page p has been merged with pages on the other VM. We map all 2^e candidate pages into memory, then look for the shared page by querying CHECKDEDUP for all candidates. The challenges for the proposed technique are two-fold: first, we have to find the function CHECKDEDUP, and we then have to deal with the complexity of guessing due to a large entropy in some kernel pages.

The memory deduplication on a hypervisor is based on a CoW mechanism. That is, the hypervisor finds duplicate pages among the VMs and merges them into a single physical page. The permission bit of the merged page is then set to read-only such that a page fault will be triggered when a write operation at the address of the page later occurs. The OS (i.e., a page fault handler) then makes a duplicate of the merged page back again to properly process the write operation. This makes the write accesses to the merged page take longer.

The write access latency can be measured by using `rdtsc`, an x86 instruction to read the current time stamp counter in CPU cycles. The following code snippet shows an example of measuring the write access latency.

```
uint64_t s, latency;
s=rdtsc();
// make write access to the page
write_access (page_addr);
latency=rdtsc() - s;
```

Figure 4 shows the traces of the measured latency of the write access for two different pages: one page has duplicates in memory and is therefore extremely likely to be deduplicated, whereas the other does not. As shown in Figure 4, a number of peaks occur only in duplicate pages owing to the repetition of merging and splitting through the CoW mechanism.

The function CHECKDEDUP can be devised by exploiting this difference in the write access times. That is, if a page p follows a pattern of the write access time, such as the duplicate page of Figure 4, CHECKDEDUP(p) outputs true; otherwise, it outputs false.

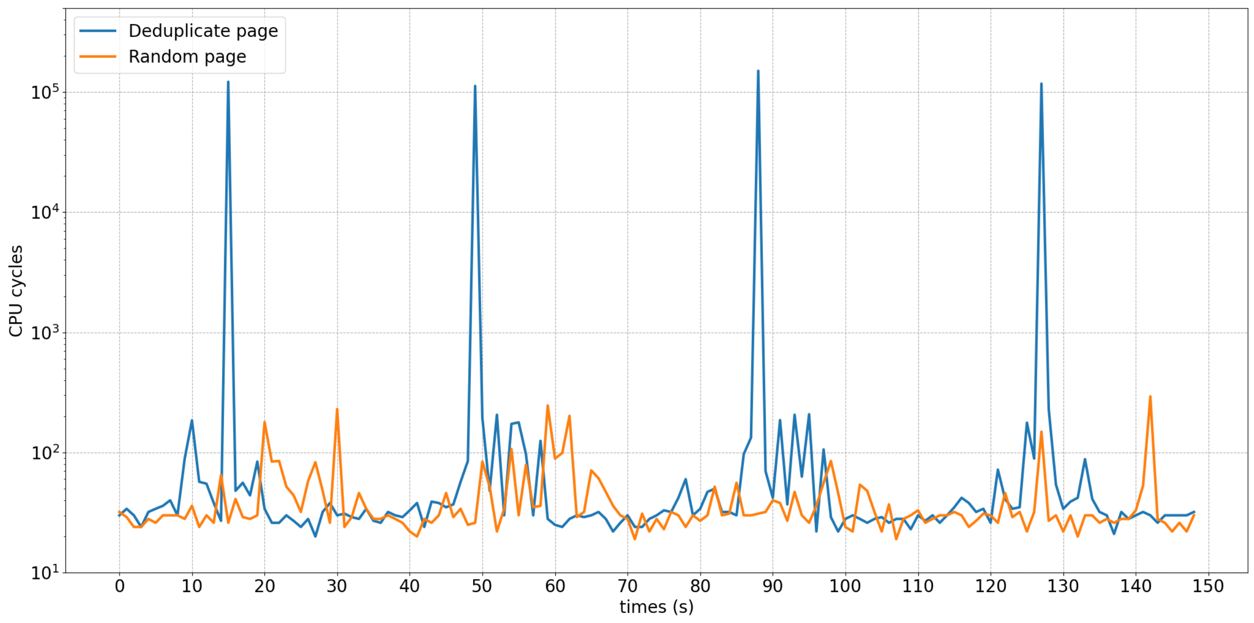


Figure 4. The time difference between deduplicated and random pages.

The complexity depends entirely on the uncertainty (i.e., entropy) of the kernel page. KASLR is the source of the uncertainty about the page content. On 64-bit Linux, KASLR provides 9-bit entropy for a kernel base (i.e., a kernel text) and 10-bit for kernel modules [17]. For some pages belonging to the kernel module, dependencies on external components (i.e., a kernel text or other kernel modules) are another source of the increased entropy. Figure 5 illustrates an example of dependencies among the kernel modules.

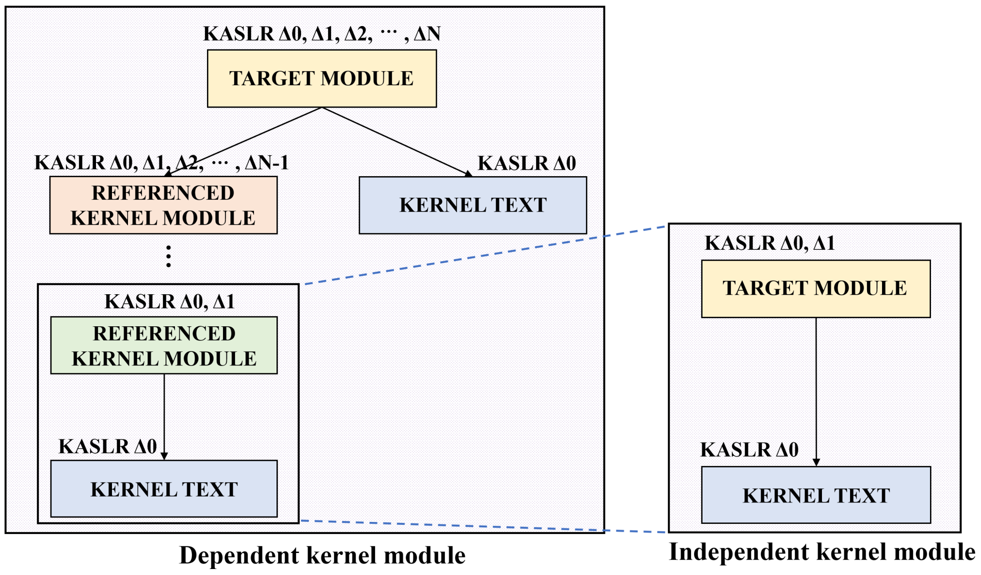


Figure 5. Dependency of the kernel module.

In the figure, $\Delta 0$ represents the entropy of the kernel text, and $\Delta 1$ represents the entropy of the kernel module, which has a reference to the kernel text, and so on. Suppose that a kernel page p belongs to the kernel module whose entropy is $\Delta 1$. As the kernel module has a reference to the kernel text, the entropy that p contains is $\Delta 0 + \Delta 1$ in total. Hence, a naive approach to find out the content of the page p is to guess all the $2^{\Delta 0 + \Delta 1}$ candidate pages. For certain modules like the target module in Figure 5 that has references to N other kernel modules, the guessing complexity will dramatically increase to $2^{\Delta 0 + \Delta 1 + \cdots + \Delta N}$.

Our method for reducing the complexity basically follows a divide-and-conquer approach. We break the problem of solving, that is, successfully guessing, a target page that has high entropy, into several smaller problems to guess each module's base address.

For instance, for the kernel page p that has entropy $\Delta 0 + \Delta 1$, we break the problem of finding out the content of p into two smaller problems; one that finds out the content of the kernel text, which has $2^{\Delta 0}$ complexity, and the other that finds out the content of the page p itself, which has $2^{\Delta 1}$ complexity. This will give us the complexity $2^{\Delta 0} + 2^{\Delta 1}$ in total, which is significantly less than $2^{\Delta 0 + \Delta 1}$. By guessing each module (i.e., solving each problem) individually, we can successfully solve the page of high entropy with reduced complexity.

Case Study and Evaluation

In this section, we present a case study on utilizing the proposed attack and its evaluation. To show the practicality and effectiveness of our attack, we target *iptables*, which is a Linux kernel-based packet filtering framework. For the evaluation, we conducted some experiments under hypervisor-based virtualization environments.

The experimental environment consists of two hypervisors, KVM in Ubuntu 18.04 LTS and VMware ESXi 7.0, both of which support memory deduplication. In the experimental setting, the VMs run a guest OS of Ubuntu 18.04 LTS with KPTI enabled. During the experiments, an attacker running in a spy VM tries to identify the kernel base address of the victim VM by using the attack technique proposed in the previous section.

In Linux, *iptables* consists of multiple kernel modules. In our experiment, we focus on the ipt_REJECT module because it has the highest dependency, as shown in Figure 6.

The module ipt_REJECT is responsible for handling packets that are subject to rejection according to the filtering rules. As shown in Figure 6, the module ipt_REJECT has dependencies on the kernel text as well as on the other kernel modules. To infer the base address of ipt_REJECT, the spy has two tasks; the first is to determine the base address of the kernel text and the second is to find all base addresses of the other modules referenced by ipt_REJECT.

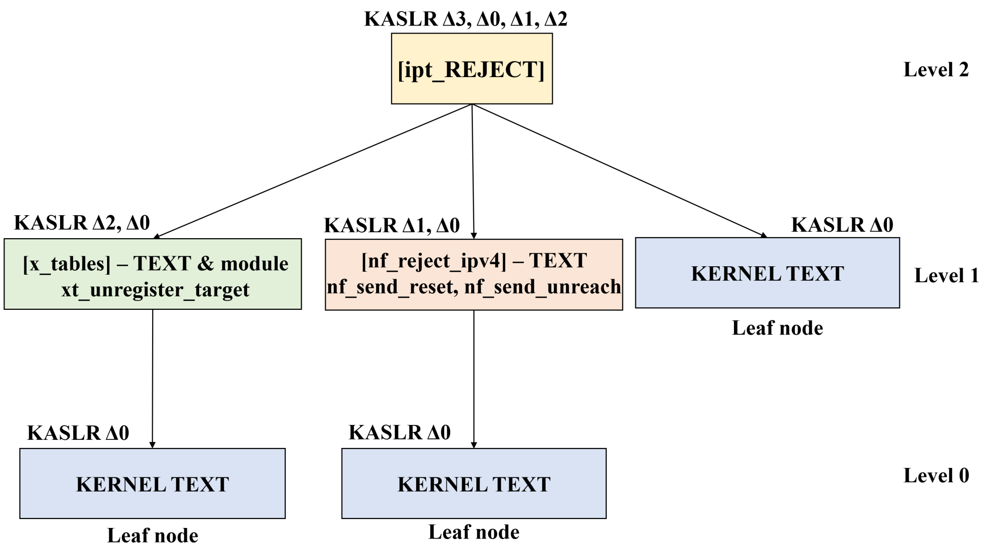


Figure 6. Dependency structure of ipt_REJECT module.

Because the base address of the kernel text is randomized through KASLR with 9-bit entropy, the spy needs to prepare 512 (=2^9) candidate pages in total before mounting the attack. After candidate pages on the kernel text have been generated and memory-mapped, the spy begins to apply write access to all pages and measure the write latency. In particular, the spy executes a write operation with a memory address of the first byte in each candidate page. For a deduplicated page among the candidates, a write will cause a CoW, which will incur relatively long latency until completion. Otherwise, a write will be completed with short latency. The number of cycles that have elapsed during a write operation is measured simultaneously while the write access occurs in the candidate page. Based on our experiment, a page with more than 10,000 cycles of write latency (cf. Figure 4) is considered a deduplicated page. Once a deduplicated page is found, the spy can determine the information of the randomized address on the kernel text.

Because ipt_REJECT has references to two other kernel modules (i.e., nf_reject_ipv4 and x_tables), the spy first has to figure out the base addresses of these kernel modules. According to our divide-and-conquer approach, the spy tries to solve the base addresses of those modules individually. As shown in the dependency graph, these modules depend on the kernel text as well as on the module itself. Because the base addresses in the kernel text have already been determined, we only need to find out the base addresses of the modules with up to 2^{10} guess pages for each. All dependencies are now solved except for the one on the last module (i.e., ipt_REJECT). This can also be solved using the 2^{10} page guessing. Thus, a total of $2^9 + 3 \times 2^{10} < 2^{12}$ page guesses are required as the maximum amount of resources.

The experimental results are presented in Table 2. For instance, it only took 12 min and 2 h to break KASLR for the kernel text in KVM and VMWare ESXi, respectively. The results show that our attack is able to successfully bypass KASLR under various settings including KPTI-enabled OS.

Table 2. Summary of attack environment and performance.

Hypervisor	CPU	Guset OS/ Kernel Version	Target	RAM			Times (hr)		Attack
				Host	Spy	Victim	Text	Module	
Ubuntu KVM	Intel Core i5-7400	Ubuntu 18.04 /5.3.7	Kernel Text/ Module	16 GB	4 GB	4 GB	0.05	0.2	✓
	Intel Core i7-6700			16 GB	4 GB	8 GB	0.05	0.2	✓
	Intel Core i7-8700			16 GB	8 GB	4 GB	0.05	0.2	✓
	Intel Core i9-9900KF			16 GB	4 GB	4 GB	0.05	0.2	✓
	AMD Ryzen7 2700x			16 GB	4 GB	4 GB	0.05	0.2	✓
VMWare ESXi	Intel Core i5-7400	Ubuntu 18.04 /5.3.7	Kernel Text/ Module	16 GB	4 GB	4 GB	2	8	✓

4. Countermeasure

In this section, we present some countermeasures for defending against this attack. The first method is to disable memory deduplication. Page sharing (i.e., memory deduplication), supported by the hypervisor, is the root cause of the proposed attack. Hence, disabling the memory deduplication in the hypervisor can be the fundamental countermeasure against such an attack. Fortunately, certain hypervisors, such as VMware ESXi, have already turned off the deduplication feature in the default configuration because of security issues.

The second is to mitigate the attack by intrusion detection. Owing to the nature of this type of attack, significant resource contention occurs during the attack. This contention can be exploited for devising intrusion detection systems that detect such attacks. There are several detection schemes for resource contention-based attacks including cache side-channel attacks [18,19]. These detection techniques make use of special hardware that provide various counters related to the CPU performance. With the help of such counters, we are able to implement an anomaly based detection system for the proposed attack.

Finally, we can mitigate the attack by introducing noise to a timer. The timing difference of write access latency, which is measurable with an `rdtsc` instruction, enables the proposed attack. Hence, we can mitigate the attack by introducing some noise to the timing source. For instance, it can be achieved by masking certain bits of a time stamp counter that `rdtsc` returns. However, making a noisy timer is not able to completely mitigate the attack, as attackers can bypass this by utilizing other timing sources (e.g., counting the iteration of a loop in a concurrent thread).

5. Related Work

In this section, we present studies related to CPU side-channel and memory deduplication attacks.

5.1. CPU Side-Channel Attack against KASLR

Hund et al. [8] determined a physically backed kernel address using a double-page fault, which literally occurs as two page faults. More specifically, it exploits the translation lookaside buffer (TLB) behavior, where a virtual address resolved to a physical address is cached into the TLB regardless of whether the virtual address has user access permission in the page table entry (PTE). The first page fault occurs when a spy accesses a kernel virtual address space in user mode. If the address fails to page table walk (i.e., it is not allocated a page), the TLB does not update any of its entries. However, if the address succeeds in the page table walk (i.e., it is allocated a page), despite the page having no access permission in user mode, the resolved address mapping is updated into the TLB entry. The spy accesses the same virtual address space occurring on the second page fault, where the spy measures the time required to handle it. If the address is physically backed, the time needed to handle the second page fault is shorter than that of the non-physically backed address owing to the TLB hit. Thus, Hund et al. [8] distinguished a valid kernel address using the time difference of the page fault handling and the behavior of the TLB.

Gruss et al. [7] broke the KASLR by leveraging the software-based memory prefetching technique provided for cache optimization. Because the prefetch instructions can be used regardless of privileged/unprivileged users, it is possible to fetch the kernel memory into the cache in user mode. Moreover, the instructions have different execution times according to the status of the CPU caches and the depth of the page table walk, where the spy can distinguish valid from invalid addresses as well as which translation-level is held by the virtual address. Thus, if a spy in user mode prefetches the kernel address space, a physically backed kernel address has a shorter prefetch time than an invalid kernel address. Based on this time difference in the prefetching, Gruss et al. [7] broke the KASLR. In contrast to these two attacks [7,8], the proposed attack does not depend on a micro-architectural feature such as the TLB but breaks the KASLR using a memory deduplication supported by a hypervisor.

Jang et al. [17] first introduced the characteristics of Intel transactional synchronization extensions (TSX) on side-channel attacks, where they de-randomized the kernel address space. Because the TSX suppresses exceptions, any exceptions that occur inside a transaction are aborted and a user-defined TSX abort handler is executed without the intervention of an OS. When conducting two types of operations (i.e., *read* and *execute*) within the transaction and measuring the execution time of the transaction, the spy can differentiate between mapped and unmapped pages (i.e., *a dTLB hit versus a dTLB miss and a page table walk*) as well as executable and non-executable pages (i.e., *decoded icache versus a page table walk*). Because no OS intervention takes place when a page fault occurs inside a transaction, fewer cycles and a higher accuracy are required than in the double page fault attack developed by Hund et al. [8], which processes a page fault by calling a signal handler.

Meltdown, proposed by Lipp et al. [9], de-randomizes KASLR by exploiting the side effects of an out-of-order execution, where the transient introductions are executed before the exception is raised (i.e., before the corresponding uOPs are retired). In a Meltdown attack, the spy loads a byte of a non-accessible kernel address, increasing the exception. Because the *load* instruction, in practice, takes over many complicated tasks (i.e., resolves the virtual-to-physical address, checks the page table entry permission, and loads the value), subsequent instructions are executed out-of-order in a transient window. More specifically, because the load instruction is issued before the properties of the page table entry are checked, if the already loaded data leave their mark in a microarchitectural state (i.e., L1 cache) over the transient windows before the exception is confirmed, the roll-back cannot clear the putrid state. Subsequently, the encoding value on the L1 cache is retrieved using a Flush+Reload [20] cache side-channel attack. If it is a valid kernel address, the Flush+Reload attack is succeeded by a cached putrid state; otherwise, it fails to decode it, and they [9] can successfully break the KASLR.

There are other side-channel attacks that break KASLR by exploiting microarchitectural vulnerabilities. Schwarz et al. [21] discovered that a store-to-load forwarding unit has a side effect in speculative execution, which can be exploited to infer whether a specific kernel address is present or not. Canella et al. [22] evaluated several Intel processors, which have hardware patches to Meltdown vulnerabilities, and found that the hardware-based mitigations are not sufficient to prevent Meltdown attacks. They demonstrated a new KASLR-breaking attack by exploiting the timing difference between valid and invalid kernel addresses in these processors.

The previous approaches are dependent on processor-specific features, such as TSX and an out-of-order execution, and are based on the fact that the kernel address space is mapped in the user address and shares a page table. However, with the introduction of kernel page table isolation (KPTI), these attacks cannot compromise the kernel security. By contrast, the attack presented in this paper is still feasible for breaking the KASLR even in the presence of the KPTI. Because our method depends solely on the memory deduplication feature of the hypervisor, the KPTI does not mitigate our attack in any way. Table 3 summarizes the differences between the previous and proposed attacks.

Table 3. Summary of the attack environment and performance.

	Attack Techniques	Prerequisite	Configuration	Breaking KASLR with KPTI
Hund et al. [8]	Double Page Fault	Kernel address space is mapped user address space	Host machine	✗
Gruss et al. [7]	Software prefetch			✗
Jang et al. [17]	Intel TSX			✗
Lipp et al. [9]	Out-of-order execution			✗
Schwarz et al. [21]	Store-to-load forwarding			✗
Canella et al. [22]	Out-of-order execution			✗
Kim et al. [23]	Return Stack Buffer			✗
Our attack	Memory Deduplication	Memory deduplication is enabled	Cross-VM	✓

5.2. Memory Deduplication Attack

Memory deduplication attacks identify pages used in a victim VM or on a website, exploiting the memory deduplication feature and measuring the execution time of the write access to distinguish merged from normal pages. Suzaki et al. [11] detected a downloaded file from the browser of a victim VM by mounting the memory deduplication attack. Gruss et al. [24] also conducted a memory deduplication attack against the page used in a sandboxed browser, in which the sandbox is broken when a victim opens a website or checks the application program in use. Barresi et al. [25] determined the user address space layout breaking the ASLR on a victim VM, applying a similar approach as our attack. However, there are some clear differences between them. First, our target is the KASLR, and we accurately analyzed its effect on the kernel address space, where kernel symbols are relocated through a static relocation (i.e., `R_X86_64_PC32` and `R_X86_64_32S`). Second, our study can also determine the based addresses of the kernel modules that have a longer attack time and greater space complexity. Using a divide-and-conquer approach, we effectively overcome such challenges.

6. Conclusions

In this paper, we proposed an attack that breaks the KASLR of another VM using the memory deduplication technique. Our attack exploits the timing difference in a write access as a side-channel to distinguish a merged kernel page from a normal page. More seriously, the proposed attack demonstrates that KASLR can be broken, even in the latest versions of a Linux kernel employing KPTI. To the best of our knowledge, this is the first study on accurately extracting the address information of the kernel modules by overcoming the high guessing complexity. One of the possible countermeasures against the proposed attack is to disable the memory deduplication feature in the hypervisor.

We emphasize that the KPTI mechanism is unable to completely protect KASLR from side-channel attacks, and believe that further research on increasing the kernel security is required. Hence, our future work will be to devise a solution that reinforces the current kernel security mechanism to defend the state-of-the-art side-channel attacks, including the proposed attack.

Author Contributions: Resources, T.K. (Taehun Kim); Writing—original draft, T.K. (Taehyun Kim); Writing—review & editing, Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported as part of Military Crypto Research Center funded by Defense Acquisition Program Administration(DAPA) and Agency for Defense Development(ADD). This work was supported by an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korean government (MSIT) (No. 2019-0-00533, Research on CPU vulnerability detection and validation). We received support in the form of a Korea University Grant in 2021.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Snow, K.Z.; Monrose, F.; Davi, L.; Dmitrienko, A.; Liebchen, C.; Sadeghi, A.R. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 574–588. [CrossRef]
2. Bletsch, T.; Jiang, X.; Freeh, V.W.; Liang, Z. Jump-oriented programming: A new class of code-reuse attack. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security, Hong Kong, China, 22–24 March 2011; pp. 30–40.
3. Dai Zovi, D. Practical return-oriented programming. *Source Boston* **2010**. Available online: https://repository.root-me.org/Exploitation%20-%20Syst%C3%A8me/Microsoft/EN%20-%20Practical%20Return%20Oriented%20Programming.pdf (accessed on 5 May 2010).
4. Davi, L.; Dmitrienko, A.; Sadeghi, A.R.; Winandy, M. *Return-Oriented Programming without Returns on ARM*; Technical Report, Technical Report HGI-TR-2010-002; Ruhr-University Bochum: Bochum, Germany, 2010.
5. Roemer, R.; Buchanan, E.; Shacham, H.; Savage, S. Return-oriented programming: Systems, languages, and applications. *ACM Trans. Inf. Syst. Secur. (TISSEC)* **2012**, *15*, 1–34. [CrossRef]
6. Checkoway, S.; Davi, L.; Dmitrienko, A.; Sadeghi, A.R.; Shacham, H.; Winandy, M. Return-oriented programming without returns. In Proceedings of the 17th ACM Conference on Computer and Communications Security, Chicago, IL, USA, 4–8 October 2010; pp. 559–572.
7. Gruss, D.; Maurice, C.; Fogh, A.; Lipp, M.; Mangard, S. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 368–379.
8. Hund, R.; Willems, C.; Holz, T. Practical timing side channel attacks against kernel space ASLR. In Proceedings of the 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 19–22 May 2013; pp. 191–205.
9. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Fogh, A.; Horn, J.; Mangard, S.; Kocher, P.; Genkin, D.; et al. Meltdown: Reading Kernel Memory from User Space. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018.
10. Gregg, B. KPTI/KAISER Meltdown Initial Performance Regressions. 2018. Available online: https://www.linux.com/news/kptikaiser-meltdown-initial-performance-regressions/ (accessed on 12 February 2018).
11. Suzaki, K.; Iijima, K.; Yagi, T.; Artho, C. Memory deduplication as a threat to the guest OS. In Proceedings of the Fourth European Workshop on System Security—EUROSEC'11, Salzburg, Austria, 10 April 2011; pp. 1–6. [CrossRef]
12. Arcangeli, A.; Eidus, I.; Wright, C. Increasing memory density by using KSM. In Proceedings of the Linux Symposium. Citeseer, 2009; pp. 19–28. Available online: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.454.5113 (accessed on 1 January 2009).
13. Waldspurger, C.A. Memory resource management in VMware ESX server. *ACM Sigops Oper. Syst. Rev.* **2002**, *36*, 181–194. [CrossRef]
14. Venkitachalam, G.; Cohen, M. Transparent Page Sharing on Commodity Operating Systems. 2009. Available online: https://patents.google.com/patent/US7500048B1/en (accessed on 3 March 2009).
15. Suzaki, K.; Iijima, K.; Yagi, T.; Artho, C. Software side channel attack on memory deduplication. In Proceedings of the ACM Symposium on Operating Systems Principles (SOSP 2011), Cascais, Portugal, 23–26 October 2011; pp. 2–3.
16. Suzaki, K.; Iijima, K.; Yagi, T.; Artho, C. Implementation of a memory disclosure attack on memory deduplication of virtual machines. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.* **2013**, *96*, 215–224. [CrossRef]
17. Jang, Y.; Lee, S.; Kim, T. Breaking Kernel Address Space Layout Randomization with Intel TSX. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 380–392.
18. Chiappetta, M.; Savas, E.; Yilmaz, C. Real time detection of cache-based side-channel attacks using Hardware Performance Counters. *Appl. Soft Comput.* **2016**, *49*, 1162–1174. [CrossRef]
19. Payer, M. HexPADS: A platform to detect stealth attacks. In *International Symposium on Engineering Secure Software and Systems*; Springer: Cham, Switzerland, 2016; Volume 9639, pp. 138–154._9. [CrossRef]
20. Yarom, Y.; Falkner, K. Flush + Reload: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In Proceedings of the 23th USENIX Security Symposium, San Diego, CA, USA, 20–22 August 2014; pp. 719–732.
21. Schwarz, M.; Canella, C.; Giner, L.; Gruss, D. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv* **2019**, arXiv:1905.05725
22. Canella, C.; Schwarz, M.; Haubenwallner, M.; Schwarzl, M.; Gruss, D. KASLR: Break It, Fix It, Repeat. In Proceedings of the 15th ACM Asia Conference on Computer and Communications Security, Taipei, China, 5–9 October 2020; pp. 481–493.
23. Kim, T.; Shin, Y. Reinforcing Meltdown Attack by using a Return Stack Buffer. *IEEE Access* **2019**, *7*, 186065–186077. [CrossRef]
24. Gruss, D.; Bidner, D.; Mangard, S. Practical memory deduplication attacks in sandboxed javascript. In Proceedings of the European Symposium on Research in Computer Security, Vienna, Austria, 21–25 September 2015; pp. 108–122.
25. Barresi, A.; Razavi, K.; Payer, M.; Gross, T.R. CAIN: Silently Breaking ASLR in the Cloud. In Proceedings of the 9th USENIX Workshop on Offensive Technologies (WOOT 15), Washington, DC, USA, 10–11 August 2015.