KASLR-MT: Kernel Address Space Layout Randomization for Multi-Tenant Cloud Systems

Fernando Vano-Garcia, Hector Marco-Gisbert

School of Computing, Engineering and Physical Sciences, University of the West of Scotland, High St, Paisley PA1 2BE, UK

ARTICLE INFO

Keywords:
Cloud
Virtualization
Security
KASLR
Memory Deduplication
Memory Management
Operating Systems

ABSTRACT

Cloud computing has completely changed our lives. This technology dramatically impacted on how we play, work and live. It has been widely adopted in many sectors mainly because it reduces the cost of performing tasks in a flexible, scalable and reliable way. To provide a secure cloud computing architecture, the highest possible level of protection must be applied. Unfortunately, the cloud computing paradigm introduces new scenarios where security protection techniques are weakened or disabled to obtain a better performance and resources exploitation. Kernel ASLR (KASLR) is a widely adopted protection technique present in all modern operating systems. KASLR is a very effective technique that thwarts unknown attacks but unfortunately its randomness have a significant impact on memory deduplication savings. Both techniques are very desired by the industry, the first one because of the high level of security that it provides and the latter to obtain better performance and resources exploitation. In this paper, we propose KASLR-MT, a new Linux kernel randomization approach compatible with memory deduplication. We identify why the most widely and effective technique used to mitigate attacks at kernel level, KASLR, fails to provide protection and shareability at the same time. We analyze the current Linux kernel randomization and how it affects to the shared memory of each kernel region. Then, based on the analysis, we propose KASLR-MT, the first effective and practical Kernel ASLR memory protection that maximizes the memory deduplication savings rate while providing a strong security. Our tests reveal that KASLR-MT is not intrusive, very scalable and provides strong protection without sacrificing the shareability.

1. Introduction

Cloud computing has become a significant aspect of our lives. It allows a provider to share pools of configurable resources (hardware/software) through virtualization, yielding new complex business models that were unpredictable some years ago. Cloud computing has been widely adopted in many sectors, mainly because it reduces the cost of performing tasks in a flexible, scalable and reliable way. From the user's point of view, they can benefit from vast computing power and storage without the need to possess the necessary hardware resources.

Infrastructure as a Service (IaaS) [11] is considered one of the fundamental building blocks for cloud services because at this level, a client is able to configure virtualized environments with high flexibility without having to concern about deploying large rooms of physical computers. Thence the service provider supplies the storage, networking and virtualization so that the client has full control over the system from OS layer upwards. Efficient resource management is fundamental to deal with a proper cloud infrastructure [22, 29, 8]. Hardware resources are a critical asset in the business and it must be managed and utilized adequately. A cloud service provider will obtain more benefits if he/she is able to operate more virtual machines with the same resources [18, 27].

Given the significance that cloud computing has in people's lives, it is imperative to offer confidentiality, integrity, and availability in any cloud computing architecture. Furthermore, there is a stack of services relying on IaaS for example *Platform as a Service* (PaaS), *Software as a Service* (SaaS) or *Function as a Service* (FaaS) to name a few. Any security issue affecting the base will affect the upper layers as well. For that reason, IaaS service providers must ensure to reach the highest possible level of security in order to guarantee a suitable quality to their clients.

Since the finding of the first computer bug back in 1947 [2] the sphere has been changing swiftly. Attackers are aware of the fact that computers are the building blocks of our society. For this reason, bugs with security implications are being abused in order to make profit of those vulnerabilities. Given the asymmetric nature of the confrontation between attackers and defenders, the former enjoy their tactical advantage while the latter design and develop defense mechanisms to prevent the successful exploitation of the most complex attacks. In recent years, there has been a transition in the *battlefield* from userland to kernel since the userland exploits complexity has overtaken the kernel ones [17]. Furthermore, a successful exploitation of a kernel vulnerability is much more dainty for an attacker.

Although cloud service providers desire to yield as much security as possible in their infrastructure, it is not always possible. Current security protection mechanisms are far from perfect and, in some cases, they introduce prohibitive overheads. Kernel randomization is a widely adopted security mechanism that introduces a prohibitive overhead [23] to the memory savings of the memory deduplication. Be-

Fernando.Vano-Garcia@uws.ac.uk (F. Vano-Garcia); Hector.Marco@uws.ac.uk (H. Marco-Gisbert)
www.fervagar.com (F. Vano-Garcia); www.hmarco.org (H. Marco-Gisbert)
ORCID(s): 0000-0001-6158-8694 (F. Vano-Garcia); 0000-0001-6976-5763 (H. Marco-Gisbert)

cause disabling the kernel randomization is not a option for security, cloud providers will sustain a forfeit of memory resources.

The rest of the paper is organized as follows. Section 2 provides a detailed background on memory deduplication and kernel randomization. Section 4 points out the reasons why the kernel randomization penalizes the memory deduplication. In section 5 the Linux kernel randomized zones are identified and a comprehensively detailed analysis on the deduplication effect of randomizing them is presented. Based on the analysis, section 6 describes the proposed solution, while the implementation of the *KASLR-MT* technique in Linux is presented in section 7. Section 8 provides an evaluation of the proposed implementation. The paper finishes in section 9 with a discussion on Linux kernel modules and how position-independent code could alleviate the issue, followed by some conclusions and future work in section 10.

2. Background

In this section we explain the memory deduplication technique used to save physical memory. Then, we explain the kernel randomization concept and the two main approaches followed to randomize code. Finally the attacker model is presented.

2.1. Memory Deduplication

Memory deduplication is a physical memory saving technique. Given the noteworthy importance of efficient memory resources utilization on behalf of cloud computing providers, deduplication is a desired feature. It is able to reduce the memory footprint across virtual machines [7, 1], decreasing the total cost of managing and ownership.

Although in the first instance deduplication was designed to be used in hypervisors [19, 1, 12, 26], it was gently applied for memory contents of non-virtualized environments as well. Then, it was widely adopted by most of the operating systems. For example the Linux kernel included Kernel Samepage Merging (KSM) in the version 2.6.32 and Windows introduced Memory Page Combining in Windows 8 [20]. Furthermore, data deduplication is commonly used in other areas such as databases or web contents [3, 13, 25].

When used with virtualization technologies, memory deduplication is usually operating at the hypervisor layer, along with the memory manager of the physical host machine. In almost all modern operating systems, memory is organized in pages. Typically, each memory page consumes 4096 bytes of physical memory. This strategy facilitates an efficient management of memory resources and enables virtual memory, which is a fundamental feature for virtualization. Thanks to this memory organization, memory deduplication can merge all pages with identical content into only one. Note that swapped pages can not be deduplicated but only the ones that reside in physical memory.

Different architectures support different page sizes. A greater pagesize implies less page table entries and less TLB faults, resulting in higher performance. However, this reduces the chances of finding two pages with equal content, which reduces the memory saving rate. Thanks to the memory management virtualization support, it is possible for hypervisors to implement a memory deduplication using pages of 4096 bytes independently of the page size guest view.

In virtualized environments, deduplication is commonly applied to the entire guest memory region corresponding to the virtual machine (often called guest physical memory). Hence, those pages belonging to that memory region will be candidates for being shared across all virtual machines. On the one hand, this increases the chances of finding matching pages to share. On the other hand, it enables different types of side-channels [21, 10] that might compromise the confidentiality. Different solutions have been researched [24, 14, 9, 16] so, depending on the adversary model of a cloud provider, memory deduplication can be used without the need to sacrifice security.

2.2. Address Space Layout Randomization

Address Space Layout Randomization (ASLR) [15] is a security technique that consists in placing the memory regions of a program in random locations. The objective of this technique is to hinder the successful exploitation of vulnerabilities that relies on the knowing program addresses. As a consequence, an attacker must obtain an addresses where code or data is located in memory in order to trigger a malicious payload. This in part is because the *Data Execution Prevention / No-Execute* (DEP/NX) prevents to execute injected code.

ASLR embraces some requirements in order to be applied correctly. It needs a high-quality entropy source for generating cryptographically secure random numbers to determine the addresses where the program will be loaded. If this requirement is not fulfilled, the predictability of the generated addresses will be high and then it will be easier for an attacker to guess correctly a valid address [5].

Kernel ASLR (KASLR) or *kernel randomization* is the applying of this technique to the kernel. Locations of kernel memory regions are determined at boot time and they are not changed until next shutdown/reboot. Each implementation has its particularities, but code and data regions are commonly randomized. It is currently being used by all the main operating systems: Apple introduced it into MacOS X Mountain Lion (Mac OS X 10.8) [28], Microsoft introduced it in Windows Vista and Linux in the kernel version 3.14.

Since the randomization is only applied at boot time, any *information leak* revealing a kernel address will derandomize the kernel, and the bypass will be effective until next reboot.

2.3. Randomizing Code

In order to randomize code, the loader must be able to choose an arbitrary address and then load the code at that address. To achieve this, the code being randomized must be compiled and linked to allow loaders to put them at random memory locations. There are mainly two different approaches to enable code to be randomized, the Position-Independent Code (PIC) and the relocations. Both approaches can load and run code at random virtual addresses but they have implementations differences that have an impact in the performance and also in the shared memory.

Position-independent code is a piece of machine code that can be executed regardless of where it is loaded in memory without any code modification. This feature is crucial for shared libraries, to keep the code segment as non-writable and to allow memory sharing by several processes using the `Copy-On-Write` (COW) mechanism [4]. Instead of referencing symbols with absolute addresses, PIC uses indexed addressing using relative offsets. For example, a data variable can be referenced relatively using the program's instruction pointer. For efficiently accessing symbols and data beyond the addressable limits of an architecture, userland programs use a look-up table called Global Offset Table (GOT). This table is placed in a data segment and contains absolute addresses of global exported symbols. Hence, PIC allows to randomize code and at the same time to it.

On the other hand, relocatable code is a piece of machine code that also can be executed regardless of where it is loaded in memory, but the executable segment needs to be patched when it is loaded. Relocation information is consulted by the loader in order to adjust symbol references through different parts of the program with final absolute addresses in-place. Although the final program with the absolute references can be more efficient than position-independent code, the load time work of relocatable code is considerably heavier, as every reference in code must be fixed-up. In contrast to PIC, relocatable code is not suitable for shared libraries. The relocations in code pages trigger the COW mechanism, generating private copies of these pages for each process using the same library loaded at different virtual addresses. The result is a decrement of memory sharing among processes, which is the main purpose of shared libraries.

2.4. Attacker Model

Disabling the kernel randomization in favour of having the highest possible memory deduplication saving rate introduces serious weaknesses. Vulnerabilities that rely on knowing where the kernel has been mapped will have 100% of success, since those addresses are not longer a secret that attackers need to obtain.

In a cloud environment, where virtual machines can interact each other, this is even more risky. A kernel vulnerability could compromise the entire physical cloud and all its virtual machines, even if they are running in different tenants and physical machines. Local, remote, inter-VM, intra-VM, inter-Tenant and intra-Tenant attackers do not need to perform a prior kernel attack to know where the kernel resides in memory, they already know, and their attacks will always success.

Our goal is to provide kernel randomization while introducing a negligible impact in the memory savings to enable cloud providers to use the KASLR protection along with the memory deduplication benefits. We assume that the kernel contains a software vulnerability that requires to bypass the kernel randomization to successfully be exploited. That is, our goal is that attackers exploiting a kernel vulnerability from local, remote, inter-VM, intra-VM, inter-Tenant and intra-Tenant must face the full kernel randomization protection with almost no effect in the memory deduplication.

3. Linux Kernel Randomization

This section describes how kernel randomization is performed in the Linux kernel v5.0.6 for the x86_64 architecture. We describe how the bootloader and Linux kernel randomizes five `memory areas`.

In Section 2.3, we described two different approaches to randomize code: position-independent code and relocations. Current Linux kernel implementation is not PIC compliant. It uses text relocations, dynamically patching all the position-dependent references after the final address of the code memory region is randomly calculated.

At the early stages of the boot process, the Linux kernel is decompressed in memory by the bootloader. One of its purposes, along with other system initialization operations, is to place the regular kernel into a random location.

The random numbers used by the KASLR are requested at boot time and, at this early stage there is no much entropy available. Having quality random numbers is key to have an effective KASLR, otherwise the addresses will be predictable and the protection useless. In order to obtain random numbers, Linux uses different sources of entropy such as rdrand, rdtsc, system timers, etc. The outcome of all the available sources is combined by using the `XOR` operation, and the result value is diffused by using circular multiplication.

The decompressor uses two random numbers to randomize both the physical address and the virtual address where the kernel will be loaded and mapped respectively. The physical address determines where the regular kernel image is going to be decompressed in memory. This image is an Executable and Linkable Format (ELF) file with a relocation table appended to it. Relocation information is generated by the static linker and it is needed for patching the virtual addresses of every position-dependent reference. Each virtual address that needs to be updated has its corresponding entry in this relocation table.

Once the kernel is decompressed into the randomly chosen physical address, the kernel image is parsed to extract the information about the segments that compose the kernel virtual memory. All the loadable segments are placed into their corresponding location. Then, relocations are processed, the kernel needs to be patched taking into account the new base address that differs from the base address that it was linked to. Listing 1 shows the pseudo-code of the loop processing and fixing relocations.

After the relocations have been applied by the bootloader, the execution control is transferred to the kernel. At this point, some system initializations operations are performed including the randomization of the remaining four `memory regions`. First the physical direct mapping (`physmap`), the dynamic memory region (`vmalloc`) and the virtual mem-

```
for each relocation do
        reloc_addr = <read relocation entry>
        // Calculate its current physical location
        ptr = (reloc_addr + phys_map)
        check_memory_bounds(ptr)

        // Update virtual address
        *ptr += kaslr_virt_offset
```

Listing 1: Pseudo-code of relocations update

ory map (`vmemmap`) are randomized, later when the first module is loaded the module base address is randomly calculated (`modules start`).

Although each region is randomized independently, there is a position order. The physical direct mapping will always be placed at a lower virtual address than the `vmalloc` region, which also will be placed at a lower virtual address than the `vmemmap` region.

The randomization algorithm is trying to use as efficient as possible the entropy. The first region is bounded to the lower third of the entire virtual space available for these three regions. A random address within the bounds is selected to map the memory region. The remaining space, subtracting a padding to avoid region overlapping, is then divided by two in order to place the second region, following the same approach. Similarly, the last region is placed at a random address within the remaining space after subtracting the padding corresponding to the second one.

The final memory region to be randomized is the zone where modules are loaded. It uses a different logic from the previous memory regions, since it is not calculated until the first module is loaded in the system. When the first module is loaded, a random number of pages between 1 and 1024 is determined and it is added to a static base address, establishing the virtual base address where the allocation of all the loadable modules starts. From that point, subsequent modules are sequentially allocated in the order as they are loaded. Once a module is loaded into its final location, relocations need to be done to fix-up their references. These relocations include absolute addresses and relative offsets. It is worth noting that even relative offsets depend on address randomization if they refer to memory regions being also randomized. In that case, the relative offset may differ, since the distance between referee and referrer is not constant. In addition, the module load order is not deterministic, so the final order and therefore the relocations applied to modules can be different in different kernel boots.

To summarize, the Linux kernel randomizes a total of six different addresses:

- `Kernel Base Physical Address`: Physical address where the kernel is decompressed.
- `Kernel Base Virtual Address`: Virtual address of the kernel text mapping, containing the code and data segments.
- `Physmap`: Direct mapping of all physical memory. Also used to dynamically allocate physically contiguous memory.
- `Vmalloc`: Memory region to dynamically allocate virtually contiguous memory.
- `Vmemmap`: Kernel virtual memory map, containing metadata of physical page frames.
- `Modules`: Virtual base address where modules are loaded.

4. The Problem: KASLR vs Deduplication

In this section, we present how kernel randomization reduces the effectiveness of memory sharing by deduplication in virtualized systems.

As detailed in section 2.1, the memory deduplication mechanism merges pages with equal content. Regardless of whether they comprise code or data and independently of their virtual address, two or more pages will be merged if their contents are identical. On the other hand, as described in section 2.3 kernel randomization could affect the host's memory sharing effectiveness if the relocation randomization approach is used. Unfortunately, as we present in section 3 we found that current Linux versions uses the *relocations* approach for randomizing the kernel.

The main issue is that current Linux kernel randomization is following a relocation approach that modifies code at boot time depending on random addresses, but memory deduplication requires to have equal content to merge pages. This conflict results on either having the KASLR enabled but reducing the shared memory or disable the KASLR to have the maximum deduplicated memory.

4.1. Breaking Shared Code

The memory deduplication fails to merge kernel code because of the randomization. Figure 1 exemplifies this concept. It shows the same hypothetical `printk()` kernel call from 2 kernels that were mapped at different base addresses. Kernel 2 base address is `0x40000` and kernel 3 base address is `0x80000`. The resulting `mov $0xaddr, %rdi` is patched by the kernel loader accordingly to the random kernel base address at boot time.

In more detail, the instruction is placing the virtual address of the string into a register and passing it to the `printk()` function, following the *System V AMD64 ABI* convention. The absolute reference in the `mov` instruction differs because the data region was loaded at different addresses. These absolute addresses need to be fixed-up at load time, before even running the kernel, altering the page content. As a consequence, those pages cannot be merged by deduplication because their content after the relocation will differ. This is just an example to illustrate the issue that prevents memory

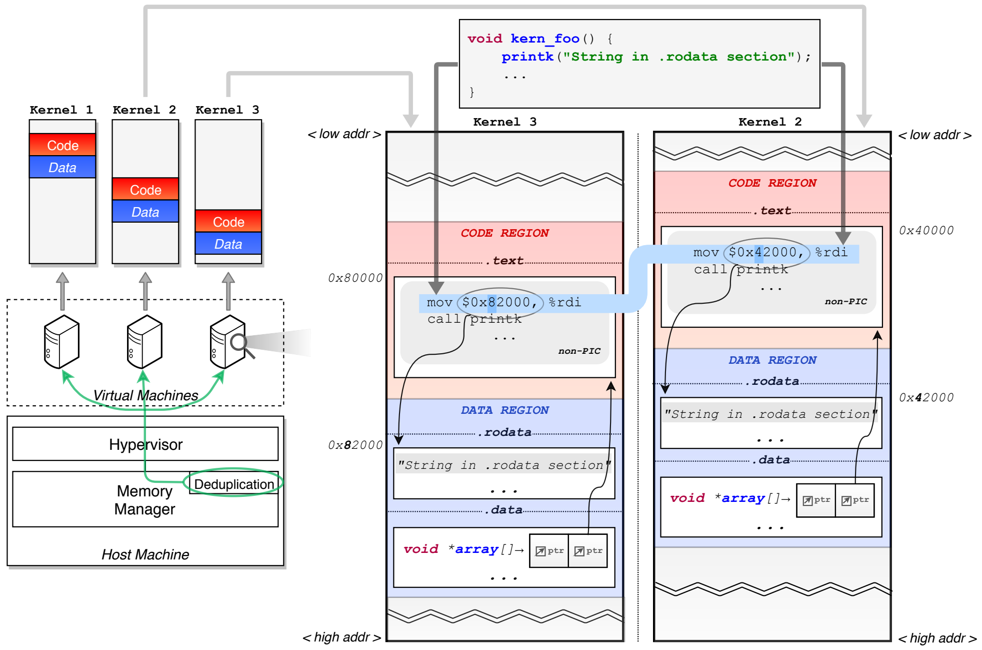

Figure 1: Overview scenario, with an hypervisor using memory deduplication and running three virtual machines. Kernel memories of two virtual machines are detailed, showing their load address along with some contents of the code and data regions.

deduplication from merging Linux code, but the Linux kernel is a modern and advanced operating system containing many coding tricks to improve both the readability and efficiency of the code.

Therefore, a full analysis is necessary, presented in section 5, to determine, on the one hand, to what extent the code is being modified because of the randomization and, on the other hand, which randomization zones produce most kernel and modules code modifications.

4.2. Breaking Shared Data

Relocations do not only affect to code but also to data variables whose contents depend on the virtual address of the memory location being referenced. For example, a data pointer containing the address of a dynamically allocated object. Therefore, the problem occurs when memory contents depend on the position of the memory location being referenced and this location is randomized. A very known example in userland applications is the usage of a `GOT/PLT` to jump to libraries. The `GOT` contains pointers to where library functions reside in memory. Therefore, the fact of randomizing libraries prevents the merging of the page holding the `GOT`.

Similarly, the Linux kernel contains structures that hold pointers to functions. Those pointers depend on the kernel base address. As a consequence, all pages containing one single pointer referencing to a randomized address will not be merged by memory deduplication. This can be extended to any memory in Linux that holds data, such us `vmalloc`, `vmemmap` and `modules data`.

Returning to figure 1, we can observe an array of pointers in the data section of both kernels, where the second element points to an address of its corresponding code section. Similar to what occurs with the relocation in the `mov` instruction, given that the base address of these two kernels differ, the pointer will differ as well. As a consequence, the host cannot share these memory pages.

5. KASLR Influence on Deduplication

As discussed in section 4, randomizing the memory layout of guest kernels reduces the effectiveness of the memory deduplication. In this section, in order to design a new KASLR, we present a comprehensive analysis of the impact of each randomized area to the memory deduplication.

Our goal is to precisely measure to what extent a particular *memory region* (kernel code, kernel data, modules code, modules data, vmalloc and vmemmap) differs when randomizing virtual *memory base addresses* (kernel vaddr., kernel paddr., vmalloc, vmemmap, modules and physical mapping). Since there are six *memory base addresses* to be randomized, our test probes a total of $2^6 = 64$ combinations per each *memory region*. The result of each one is the number of pages that are different between a particular *memory region*. For example, enabling randomization for all *memory base addresses* but one, enabling randomization for all *memory base address* but two, etc. By doing this, we can precisely achieve the following:

1. **The KASLR influence**: of each combination to identify which randomized *memory base address* have more impact on memory deduplication.
2. **The best combination**: for our purpose. In our case, we aim to fully randomize as many *memory base addresses* as possible. To achieve that we need to obtain those that have low or zero impact on memory deduplication. As detailed in the solution section 6, those

Linux Code

Randomized Base Addresses						
modules	vmemmap	vmalloc/ ioremap	physical mapping	kernel vaddr.	kernel paddr.	% equal pages
●	●	●	●	○	●	100
●	●	●	●	○	○	100
●	●	●	○	○	●	100
●	●	○	●	○	●	100
●	○	●	●	○	●	100
○	●	●	●	○	●	100
●	●	●	○	○	○	100
●	●	○	●	○	○	100
●	●	○	○	○	●	100
●	○	●	●	○	○	100
●	○	●	○	○	●	100
●	○	○	●	○	●	100
○	●	●	●	○	○	100
○	●	●	○	○	●	100
○	●	○	●	○	●	100
○	○	●	●	○	●	100
●	●	○	○	○	○	100
●	○	●	○	○	○	100
●	○	○	●	○	○	100
●	○	○	○	○	●	100
○	●	●	○	○	○	100
○	●	○	●	○	○	100
○	●	○	○	○	●	100
○	○	●	●	○	○	100
○	○	●	○	○	●	100
○	○	○	●	○	●	100
●	○	○	○	○	○	100
○	●	○	○	○	○	100
○	○	●	○	○	○	100
○	○	○	●	○	○	100
○	○	○	○	○	●	100
○	○	○	○	○	○	100
●	●	●	●	●	●	2.6
●	●	●	●	●	○	2.6
●	●	●	○	●	●	2.6
●	●	○	●	●	●	2.6
●	○	●	●	●	●	2.6
○	●	●	●	●	●	2.6
●	●	●	○	●	○	2.6
●	●	○	●	●	○	2.6
●	●	○	○	●	●	2.6
●	○	●	●	●	○	2.6
●	○	●	○	●	●	2.6
●	○	○	●	●	●	2.6
○	●	●	●	●	○	2.6
○	●	●	○	●	●	2.6
○	●	○	●	●	●	2.6
○	○	●	●	●	●	2.6
●	●	○	○	●	○	2.6
●	○	●	○	●	○	2.6
●	○	○	●	●	○	2.6
●	○	○	○	●	●	2.6
○	●	●	○	●	○	2.6
○	●	○	●	●	○	2.6
○	●	○	○	●	●	2.6
○	○	●	●	●	○	2.6
○	○	●	○	●	●	2.6
○	○	○	●	●	●	2.6
●	○	○	○	●	○	2.6
○	●	○	○	●	○	2.6
○	○	●	○	●	○	2.6
○	○	○	●	●	○	2.6
○	○	○	○	●	●	2.6
○	○	○	○	●	○	2.6

Table 1
Analysis results of the Linux code memory region.

Linux Data

Randomized Base Addresses						
modules	vmemmap	vmalloc/ ioremap	physical mapping	kernel vaddr.	kernel paddr.	% equal pages
●	●	○	○	○	●	80.9
●	○	●	○	○	○	80.8
○	●	○	○	○	○	80.8
○	○	○	○	○	○	80.8
●	○	○	○	○	○	80.7
○	○	●	○	○	○	80.7
●	●	●	○	○	○	80.6
●	○	●	○	○	●	80.5
●	●	○	○	○	○	80.4
○	●	●	○	○	○	80.4
○	○	●	●	○	○	80.4
●	○	●	●	○	○	80.3
○	●	○	○	○	●	80.3
○	○	○	●	○	○	80.3
○	●	●	●	○	●	80.2
●	○	○	●	○	○	80.2
○	○	●	○	○	●	80.2
●	○	●	●	○	●	80.1
○	●	●	●	○	○	80.1
○	●	○	●	○	○	80.1
○	○	○	●	○	●	80.1
●	●	●	●	○	●	80.0
●	●	●	●	○	○	80.0
●	○	○	●	○	●	80.0
○	○	●	●	○	●	80.0
●	○	○	○	○	●	80.0
●	●	●	○	○	●	79.8
●	●	○	●	○	●	79.7
●	●	○	●	○	○	79.7
○	●	○	●	○	●	79.7
○	●	●	○	○	●	71.5
○	○	○	○	○	●	71.2
○	○	○	○	●	○	64.7
●	○	○	○	●	●	64.6
○	●	●	○	●	○	64.6
●	●	○	●	●	○	64.4
●	○	○	●	●	●	64.4
●	●	●	●	●	○	64.3
●	●	●	○	●	○	64.3
○	○	●	○	●	○	64.3
○	○	●	●	●	●	64.2
●	○	●	○	●	○	64.2
○	○	○	●	●	●	64.2
●	○	○	○	●	○	64.2
●	●	○	●	●	●	64.1
●	●	○	○	●	●	64.1
○	●	●	●	●	○	64.1
○	●	○	○	●	●	64.1
○	○	○	●	●	○	64.1
●	○	●	●	●	○	64.0
○	●	●	○	●	●	64.0
○	●	○	○	●	○	64.0
○	○	○	○	●	●	64.0
●	●	●	○	●	●	63.9
●	●	○	○	●	○	63.9
○	●	○	●	●	○	63.9
●	○	●	●	●	●	63.8
●	○	○	●	●	○	63.8
○	○	●	●	●	○	63.8
○	○	●	○	●	●	63.8
○	●	●	●	●	●	63.7
●	○	●	○	●	●	63.6
○	●	○	●	●	●	63.5
●	●	●	●	●	●	58.4

Table 2
Analysis results of the Linux data memory region.

memory base address will be randomized as usual and we are not modifying the kernel code that randomizes them.

3. **Not biased and Independent Results**: of the number of virtual machines. Instead of calculating the memory deduplication differences before and after randomization, we are calculating how different or equal is a *memory region* after applying randomization. The first approach can be tuned to obtain almost any number by choosing a high number of virtual machines.

Modules Code

Randomized Base Addresses						
modules	vmemmap	vmalloc/ ioremap	physical mapping	kernel vaddr.	kernel paddr.	% equal pages
○	○	●	●	○	○	30.4
○	○	○	●	○	○	29.4
○	●	●	●	○	○	27.2
○	●	●	●	○	●	26.8
○	●	●	○	○	●	25.9
○	○	○	●	○	●	25.6
○	●	●	○	○	○	25.5
○	●	○	○	○	○	24.9
○	●	○	●	○	●	24.7
○	○	○	○	○	○	24.5
○	○	●	○	○	○	24.4
○	○	●	●	○	●	24.3
○	●	○	○	○	●	23.8
○	○	●	○	○	●	23.7
○	○	○	○	○	●	23.7
○	●	○	●	○	○	23.4
●	○	●	●	○	●	12.2
○	●	●	●	●	○	12.1
●	●	●	○	○	○	12.1
●	●	○	○	○	●	12.1
●	○	●	●	○	○	12.1
●	●	○	○	○	○	12.1
●	●	●	●	○	●	12.0
●	●	●	●	○	○	12.0
●	●	●	○	○	●	12.0
●	○	●	○	○	●	12.0
●	○	○	●	○	●	12.0
○	●	●	○	●	○	12.0
○	○	●	○	●	●	12.0
●	○	○	●	○	○	12.0
●	○	○	○	○	●	12.0
○	○	○	●	●	○	12.0
●	○	○	○	○	○	12.0
○	●	●	●	●	●	11.9
●	●	●	○	●	○	11.9
●	●	○	●	●	○	11.9
●	●	○	●	○	●	11.9
●	●	○	○	●	●	11.9
●	○	●	○	●	●	11.9
●	○	○	●	●	●	11.9
○	●	○	●	●	●	11.9
●	●	○	●	○	○	11.9
●	●	○	○	●	○	11.9
●	○	○	●	●	○	11.9
○	○	●	●	●	○	11.9
●	○	●	○	○	○	11.9
○	●	○	○	●	○	11.9
○	○	●	○	●	○	11.9
●	●	●	●	●	●	11.8
●	●	●	●	●	○	11.8
●	●	●	○	●	●	11.8
●	●	○	●	●	●	11.8
●	○	●	●	●	●	11.8
●	○	●	●	●	○	11.8
○	●	●	○	●	●	11.8
○	○	●	●	●	●	11.8
●	○	●	○	●	○	11.8
●	○	○	○	●	●	11.8
○	●	○	●	●	○	11.8
○	●	○	○	●	●	11.8
○	○	○	●	●	●	11.8
●	○	○	○	●	○	11.8
○	○	○	○	●	●	11.8
○	○	○	○	●	○	11.8

Table 3
Analysis results of the modules code memory region.

Modules Data

Randomized Base Addresses						
modules	vmemmap	vmalloc/ ioremap	physical mapping	kernel vaddr.	kernel paddr.	% equal pages
○	○	●	●	○	○	48.6
○	○	○	●	○	○	47.8
○	●	●	●	○	○	46.1
○	●	●	●	○	●	45.8
○	●	●	●	●	○	45.3
○	●	●	○	○	●	44.9
○	●	○	○	○	○	44.7
○	○	●	○	○	○	44.7
○	○	○	●	○	●	44.6
○	●	○	●	○	●	44.1
○	●	●	○	○	○	44.0
○	○	●	●	○	●	43.5
○	○	○	○	○	○	43.2
○	●	○	○	○	●	43.1
○	○	●	○	○	●	43.1
○	○	○	○	○	●	43.1
○	●	○	●	○	○	41.8
○	●	●	●	●	●	41.4
○	○	○	●	●	○	41.3
○	○	○	○	●	○	41.2
○	○	●	○	●	○	40.9
○	●	○	○	●	○	40.6
○	●	○	●	●	●	40.1
○	●	○	○	●	●	40.0
○	●	○	●	●	○	39.6
○	○	●	○	●	●	39.6
○	○	●	●	●	○	39.5
○	○	○	●	●	●	39.3
○	○	●	●	●	●	39.1
○	●	●	○	●	○	39.0
○	●	●	○	●	●	38.9
○	○	○	○	●	●	38.4
●	●	●	●	●	○	30.3
●	●	●	●	○	●	30.3
●	●	●	○	●	●	30.3
●	○	●	●	●	●	30.3
●	●	●	○	●	○	30.3
●	●	●	○	○	●	30.3
●	●	○	●	●	○	30.3
●	●	○	○	●	●	30.3
●	○	●	●	●	○	30.3
●	○	●	○	●	●	30.3
●	○	○	●	●	●	30.3
●	●	●	○	○	○	30.3
●	●	○	●	○	○	30.3
●	●	○	○	●	○	30.3
●	●	○	○	○	●	30.3
●	○	●	●	○	○	30.3
●	○	●	○	●	○	30.3
●	○	●	○	○	●	30.3
●	○	○	●	○	●	30.3
●	○	○	○	●	●	30.3
●	○	○	○	●	○	30.3
●	○	○	○	○	●	30.3
●	●	○	●	●	●	30.2
●	●	●	●	○	○	30.2
●	●	○	●	○	●	30.2
●	○	●	●	○	●	30.2
●	○	○	●	●	○	30.2
●	●	○	○	○	○	30.2
●	○	●	○	○	○	30.2
●	○	○	●	○	○	30.2
●	○	○	○	○	○	30.2
●	●	●	●	●	●	30.1

Table 4
Analysis results of the modules data memory region.

For example, if half of the memory of a virtual machine can be deduplicated, then if we have two identical machines, the memory deduplication will save 50% of the memory but if we have three, the deduplication will report %66 of memory saved. Therefore, although this is reflecting the actual savings, it is not appropriate to know the real impact of the KASLR because of the noise that is being introduced in the measure.

Therefore our analysis will focus on obtaining the num-

Vmalloc Space

Randomized Base Addresses						
modules	vmemmap	vmalloc/ ioremap	physical mapping	kernel vaddr.	kernel paddr.	% equal pages
●	○	●	●	○	●	3.0
●	○	●	●	○	○	3.0
●	●	●	●	●	●	2.9
●	●	●	○	●	○	2.9
●	○	●	○	●	●	2.9
○	●	●	●	●	○	2.9
○	●	○	●	●	●	2.9
●	●	○	●	○	○	2.9
●	●	○	○	●	○	2.9
●	●	○	○	○	●	2.9
●	○	●	○	○	●	2.9
●	○	○	●	○	●	2.9
○	●	●	○	●	○	2.9
○	●	●	○	○	●	2.9
○	○	●	●	○	●	2.9
○	○	●	○	●	●	2.9
●	○	●	○	○	○	2.9
●	○	○	●	○	○	2.9
●	○	○	○	○	●	2.9
○	●	●	○	○	○	2.9
○	●	○	●	○	○	2.9
○	●	○	○	○	●	2.9
○	○	●	○	●	○	2.9
○	○	○	●	●	○	2.9
○	●	○	○	○	○	2.9
○	○	○	●	○	○	2.9
○	○	○	○	●	○	2.9
○	○	○	○	○	○	2.9
●	●	●	●	●	○	2.8
●	●	●	●	○	●	2.8
●	●	●	○	●	●	2.8
●	●	○	●	●	●	2.8
●	○	●	●	●	●	2.8
○	●	●	●	●	●	2.8
●	●	●	●	○	○	2.8
●	●	●	○	○	●	2.8
●	●	○	●	●	○	2.8
●	●	○	●	○	●	2.8
●	●	○	○	●	●	2.8
●	○	●	●	●	○	2.8
●	○	○	●	●	●	2.8
○	●	●	●	○	●	2.8
○	●	●	○	●	●	2.8
○	○	●	●	●	●	2.8
●	○	●	○	●	○	2.8
●	○	○	●	●	○	2.8
○	●	●	●	○	○	2.8
○	●	○	●	●	○	2.8
○	●	○	●	○	●	2.8
○	●	○	○	●	●	2.8
○	○	●	●	●	○	2.8
○	○	○	●	●	●	2.8
●	●	○	○	○	○	2.8
●	○	○	○	●	○	2.8
○	●	○	○	●	○	2.8
○	○	●	●	○	○	2.8
○	○	●	○	○	●	2.8
○	○	○	●	○	●	2.8
○	○	○	○	●	●	2.8
●	○	○	○	○	○	2.8
○	○	●	○	○	○	2.8
○	○	○	○	○	●	2.8
●	●	●	○	○	○	2.7
●	○	○	○	●	●	2.7

Table 5
Analysis results of the `vmalloc` memory region.

Virtual Memory Map

Randomized Base Addresses						
modules	vmemmap	vmalloc/ ioremap	physical mapping	kernel vaddr.	kernel paddr.	% equal pages
○	○	○	●	○	○	42.5
●	○	●	○	○	○	41.8
○	○	●	○	○	○	41.8
○	○	●	○	●	○	41.7
●	○	○	●	●	○	41.1
○	○	○	○	●	○	40.9
●	○	○	○	●	○	40.8
●	○	●	●	●	○	40.7
○	○	○	○	○	○	40.7
○	○	○	●	●	○	40.6
○	○	●	●	●	○	40.5
○	○	●	●	○	○	40.4
●	○	○	○	○	○	40.4
●	○	○	●	○	○	40.3
●	○	●	○	●	○	40.2
●	○	●	●	○	○	39.1
●	○	○	●	○	●	38.0
●	○	○	○	○	●	37.9
●	○	●	○	○	●	37.5
○	○	●	●	●	●	37.4
○	○	●	○	○	●	35.8
○	○	○	○	●	●	35.6
●	○	●	●	●	●	35.5
●	○	○	●	●	●	35.3
○	○	○	○	○	●	35.2
○	○	○	●	○	●	35.1
●	○	○	○	●	●	34.8
○	○	○	●	●	●	34.8
○	○	●	●	○	●	34.7
●	○	●	●	○	●	34.6
○	○	●	○	●	●	34.0
●	○	●	○	●	●	33.6
●	●	●	●	●	○	0.1
●	●	●	●	○	●	0.1
●	●	●	○	●	●	0.1
●	●	○	●	●	●	0.1
●	●	●	●	○	○	0.1
●	●	●	○	●	○	0.1
●	●	●	○	○	●	0.1
●	●	○	●	●	○	0.1
●	●	○	○	●	●	0.1
○	●	●	●	●	○	0.1
○	●	●	●	○	●	0.1
○	●	○	●	●	●	0.1
●	●	●	○	○	○	0.1
●	●	○	○	●	○	0.1
●	●	○	○	○	●	0.1
○	●	●	●	○	○	0.1
○	●	●	○	○	●	0.1
○	●	○	●	●	○	0.1
○	●	○	○	●	●	0.1
●	●	○	○	○	○	0.1
○	●	●	○	○	○	0.1
○	●	○	●	○	○	0.1
○	●	○	○	●	○	0.1
●	●	●	●	●	●	0.0
○	●	●	●	●	●	0.0
●	●	○	●	○	●	0.0
○	●	●	○	●	●	0.0
●	●	○	●	○	○	0.0
○	●	●	○	●	○	0.0
○	●	○	●	○	●	0.0
○	●	○	○	○	●	0.0
○	●	○	○	○	○	0.0

Table 6
Analysis results of the `vmemmap` memory region.

ber of pages that are equal in a particular *memory region* after applying randomization. Note that our analysis obtains separately the randomization effect produced to kernel code and data, but it is not possible to randomize both separately. At the same time, the *physical mapping* can influence a particular *memory region*, but it is actually a virtual mapping to all physical memory and not actual memory, so their content must be ignored in the analysis.

Even though the Linux kernel guests uses pages of 2 MiB, the memory deduplication implementation in Linux

	Impact on Linux Kernel Memory Regions					
Randomized Addresses	Linux Code	Linux Data	Modules Code	Modules Data	vmalloc	vmemmap
Kernel Base (Physical)	none	none	none	none	none	low
Kernel Base (Virtual)	high	med	med	low	none	none
Physical Direct Mapping	none	none	none	none	none	none
Vmalloc/ioremap	none	none	none	none	none	none
Vmemmap	none	none	none	none	none	high
Modules	none	none	med	med	none	none

Table 7
Impact of randomizing *kernel addresses* on the different *memory regions* of the Linux kernel.

works with 4 KiB pages. Therefore, our analysis calculating how equal is a *memory region* after applying all randomization detects differences at page level, that is, a single bit difference in a page is reported as a full 4 KiB page mismatch. In our experiments, we used a generic GNU/Linux Ubuntu 19.04 with Linux 5, and Gnome desktop with the networking enabled.

Tables 1 and 2 shows the analysis of the Linux code and data respectively. Tables 3 and 4 do the same for the code and data of the modules, table 5 shows the analysis of `vmalloc` and finally table 6 presents the results of the Virtual Memory Map *memory region*. The column *% equal pages* on the tables indicates the percentage of similarity between a particular *memory region* across all *memory base address* randomization combinations.

The results of the randomization effects on the Linux code shown in table 1 point out that the code *memory region* is only and drastically affected by the randomization of the kernel virtual address. When the randomization is applied to the *kernel virtual address*, the % of equal pages is reduced from 100% to 2.6% regardless of whether we are randomizing any of the other areas or not. Although we discussed in section 3 that current Linux is using relocations to implement the Kernel ASLR, the results indicate that almost all kernel code pages are altered in this relocation process resulting in a non-shared Linux code.

Similarly, results for the data *memory region* shown in table 2 suggest that the kernel virtual address also affects the data region, although to a lesser extent. In our experiments, the percentage of equal pages for the case when randomizing all the *memory base address* but the *kernel virtual address* is 80%, which decreases to 64.7% when only that address is randomized. Unlike the code region, the best case is never 100% because of the presence of unique data contents that do not depend on randomization. An example of data information with low *shareability* is a variable storing timestamps.

Regarding the Linux loadable modules, as stated in section 3, they are a special case because they are randomized following a logic different from the other addresses. They are randomized at the moment the first module is loaded in the kernel, and the module loading order is not deterministic. Therefore, even if the load offset is not randomized, the non-deterministic load order can introduce other kind of differences in memory contents. It explains the low rates of equal pages shown in table 3. This circumstance will be discussed in more detail in section 9. Considering this and examining the results of the table, we observe that both *base addresses* of the kernel and the modules are affecting to the contents of the modules *memory region*. The common characteristic of the top results for this case is that both are not randomized (30.4% to 23.4%). The same issue with the non-deterministic loading order of modules affects the modules data *memory region*. Apart from that, we can see in table 4 two main weights sorting the results: the influence of randomizing the *modules base address* and, to a lesser extent, the randomization of the *kernel virtual address*.

The results of the `vmalloc` *memory region* shown in table 5 reveals a really low percentage of equal pages in all cases. This might be caused by the nature of its contents, given that it is a general-purpose data *memory region* with different kinds of information. In addition, this memory is dynamically allocated and some allocations may last just a few milliseconds. This volatility makes difficult to find a consolidated common state to compare with. However, unlike the other analysed *memory regions*, no pattern is observed in the table. Therefore, kernel randomization is not influencing its contents. Accordingly, we consider that the `vmalloc` region should be always randomized because little profit can be drawn from the memory deduplication viewpoint.

Lastly, the results for the `vmemmap` *memory region* shown in table 6 reveals other two main sorting weights. On the one hand, it is drastically affected by the randomization of its own *memory base address*, going from 30-40% when it is not randomized down to 0.1% when it is randomized. The reason is the presence of lists of objects with references to the virtual address of the previous and next objects, also present in the same *memory region*. The second factor influencing the percentage of equal pages is the *kernel physical address*, although it is softer compared with the impact of the `vmemmap` *base address*.

A summary of the analysis is presented in table 7, condensing the obtained results into a quick lookup table categorizing the influences that the *randomized address* have on the different *memory regions*. From it, we can extract three main conclusions. First, the Linux kernel *virtual address*

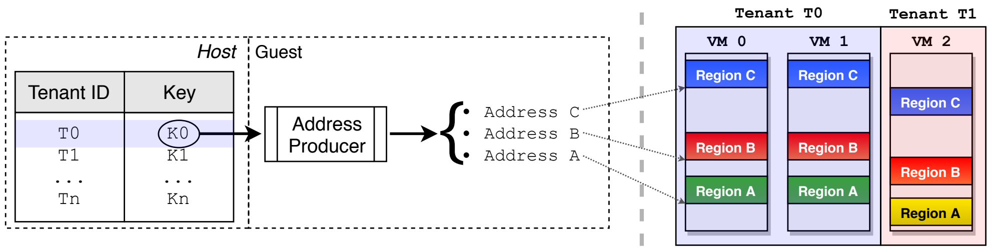

Figure 2: Design of KASLR-MT. A tenant key is transferred to the guest to deterministically produce the final addresses of kernel memory regions.

has significant impact on its code and data *regions*, along with the modules code *region*. Second, the `vmemmap` *region* is highly affected by its own address. And third, the Linux kernel *physical address*, along with `physmap` and `vmalloc virtual addresses` have little influence on memory contents.

6. KASLR-MT: A Multi-Tenancy KASLR

In this section we present KASLR-MT, a kernel randomization design for multi-tenant cloud systems, compatible with memory deduplication while providing a strong level of security. The two core ideas of KASLR-MT are:

1. **Full randomize low impact regions**. Memory regions that have zero or negligible impact when the base address are randomized, will be randomized as in the original implementation.

2. **Shared randomization on high impact regions**. Memory regions that have high or moderated impact when the base address are randomized, will be randomized per Tenant. That is, all kernels belonging to the same Tenant will have their memory regions at the same address locations.

As stated in section 4 and studied in section 5, kernel randomization produces alterations in *memory contents* that breaks memory sharing. To remedy this situation and yield memory sharing by deduplication in the host, kernel memory layouts of guest virtual machines should be as similar as possible.

A quick workaround to achieve this, although it is not a solution by itself, would be to disable kernel randomization in the guests. If kernel randomization is completely disabled, guest kernels are deterministically placed in a default address settled at compile time. Comparing guest instances of the same kernel, all of them produce the same kernel memory layout. As a consequence, relocations that are different when the kernel is randomized are equal when it is not. Even though this approach can be a suitable method in certain environments, for example private clouds properly secured for external attacks, we do not recommend disabling any security mechanism. As stated in section 2.4, the fact of disabling KASLR to obtain better memory deduplication savings introduces serious weaknesses.

In order to properly tackle the randomization-vs-sharing problem, we propose Kernel Address Space Layout Randomization Multi-Tenancy (KASLR-MT), a para-virtualization based solution that enables the hypervisor to provide the kernel memory layout to the guest virtual machines in multi-tenant cloud environments. It gives the host machine the ability to decide the locations of different kernel memory regions of guest virtual machines. This solution is designed for multi-tenant cloud systems, where different tenants are owners of different groups of virtual machines. Thus, virtual machines belonging to a same tenant can attain a common kernel memory layout, allowing the host machine to share more memory. Security is kept, since the kernel memory layout is still unpredictable from the attacker perspective.

The cloud infrastructure maintains a table with one-to-one correspondence, linking Tenant-ID and a unique random key. Each unique key serves to deterministically produce base addresses of guest kernel memory regions. The table relation must be bijective so that there are no duplicate keys in the cloud infrastructure, and keys must be unpredictable. Otherwise, since the algorithm to produce addresses from a given key must be highly deterministic, an attacker who can predict the key of a victim tenant will be able to guess the kernel address space layout of the victim's virtual machines.

Figure 2 shows the design of KASLR-MT. The table relating each tenant with its key is stored in the host machine, so that the hypervisor transfers the corresponding key to a virtual machine when it is turned on. Guest kernels will use the key as input to the the Address Producer to get memory addresses of regions that want to randomize. On the right part of the figure, the kernel memory of three different virtual machines are depicted. Tenant T0 owns two of them (VM0 and VM1) while the other virtual machine (VM2) belongs to Tenant T1. Both VM0 and VM1 obtain the same memory layout because both kernels are using the same key (K0) as input to the Address Producer algorithm. On the other hand, VM2 uses the key of Tenant T1 (K1), obtaining a different memory layout.

The table information needs to be distributed to all the host machines forming the cloud infrastructure. Coherence of this distributed state in the infrastructure is important to support virtual machine migration. If, otherwise, the keys were only kept locally in host machines, a migrated virtual machine whose kernel memory layout was generated with a

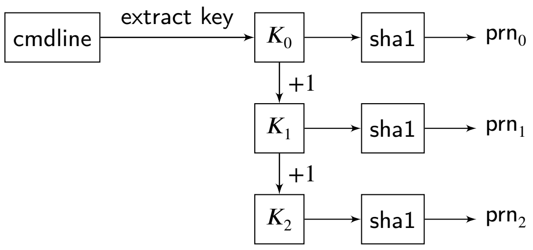

Figure 3: Block diagram describing our proof-of-concept implementation of the Address Producer component.

different key would introduce layout diversity within a same group of virtual machines.

The lifetime of a tenant's key goes from when its first virtual machine starts to when its last active virtual machine turns off, independently of the state of other virtual machines belonging to different tenants. A key cannot be changed if any virtual machine is running, since it would introduce layout diversity as well; i.e., kernel memory layout of guests started after a key change would likely differ from those started before. When the last virtual machine of a given tenant turns off, the key can be safely purged. In fact, we strongly recommend to purge it, in order to force the creation of a new key for subsequent guest kernels.

KASLR-MT combines the deduplication effectiveness of disabling kernel randomization and the statistical defense provided by the kernel randomization protection. The approach followed by KASLR-MT is similar to the one commonly used by some major operating systems (e.g., Windows and Mac OSX) to randomize the virtual address of userspace libraries, using a per-boot ASLR scheme: a random system-wide value is computed once at system startup, and it is used to calculate the virtual address where libraries are loaded at. This random value is not changed until the next system reboot. Similarly, our design uses a random key, which determines the guest kernel memory layout, for all the virtual machines of a tenant until the moment when the last one shuts down. However, with per-boot userspace ASLR, a local attacker already knows the address space layout of the target application. This is not the case for KASLR-MT, which offers the same protection as KASLR. Further details can be found in section 8.

7. KASLR-MT Linux Implementation

In this section we present the KASLR-MT implementation in the Linux kernel based on the chosen randomization forms explained in section 5.

On the one hand, those *kernel random base addresses* with low or none impact on memory contents will not provide us great additional sharing benefits. Hence, they can be randomized as in the original implementation to maximize the unpredictability of their location. On the other hand, *kernel addresses* affecting contents when they are randomized should be *transferred* by the hypervisor. Therefore, the configuration we have chosen is the following:

Randomized addresses:

- Kernel Physical Address
- Direct Physical Mapping (`physmap`)
- `Vmalloc/ioremap`

Transferred addresses:

- Kernel Virtual Address
- Virtual Memory Map (`vmemmap`)
- Modules

To communicate the key to the guest Linux kernel, we use the kernel's command-line parameters, passing directly its corresponding key value. Since Linux normally prints the contents of the cmdline to the kernel ring buffer at the beginning of its boot process, we need to retrieve the key and clean-up the cmdline buffer before it is printed out, to avoid leaking this information. If the key is leaked, every address being obtained through KASLR-MT could be calculated.

Once the key is obtained, we need to get three addresses from it. Actually, we need to get a pseudo-random number per address, which will be used to calculate deterministically the final virtual address. To achieve this in a secure and robust way, we could use a cryptographically secure pseudo-random number generator (CSPRNG), seeding it with the transferred key to produce pseudo-random numbers, or a proper key derivation function (KDF) chain, as used in some encryption algorithms (e.g., AES-GCM-SIV [6]). However, since our purpose is to develop a proof of concept to evaluate our design, we have followed a simple approach to simplify the comprehension, inspired in these kind of algorithms. The general idea is depicted in figure 3. After extracting the key from the kernel command-line, a digest of the key itself is calculated by using the `sha1` algorithm, and the result is treated as a pseudo-random number to get the final kernel base virtual address. The same procedure is done two more times, to extract the pseudo-random numbers for obtaining the `vmemmap` address and the modules base, incrementing the key by one before the digest calculation.

Each pseudo-random number is used instead of a random number as done in standard KASLR. Listing 2 exemplifies how the virtual address of the kernel base is obtained, using the KASLR-MT pseudo-random number instead of calling to `kaslr_get_random_long()`. The remaining addresses are calculated similarly.

Regarding the key length, given that we are using the kernel command-line as input method, we consider that keys should use common ascii alphanumeric characters. However, keys should have enough entropy to resist brute force attacks against a leaked kernel address. If a kernel address is leaked, an attacker could be able to recover the key by brute forcing the hash algorithm. Based on that the maximum entropy that can be obtained through `get_random_bytes()` is 64 bits, we consider this value as a proper entropy for a KASLR-MT key value. In addition, we want to use common printable ascii characters, avoiding the space and DEL characters. Therefore, the valid ascii characters are from value 0x21 ('!')

```
static unsigned long find_random_virt_addr(...)
{
        ...
        slots = (KERNEL_IMAGE_SIZE - minimum - image_size) /
                        CONFIG_PHYSICAL_ALIGN + 1;

---     random_addr = kaslr_get_random_long("Virtual") % slots;
+++     if (kaslrmt_enabled)
+++         random_addr = kaslrmt_prn[0] % slots;
+++     else
+++         random_addr = kaslr_get_random_long("Virtual") % slots;

        return random_addr * CONFIG_PHYSICAL_ALIGN + minimum;
}
```

Listing 2: Changes in arch/x86/boot/compressed/kaslr.c file of the Linux source code.

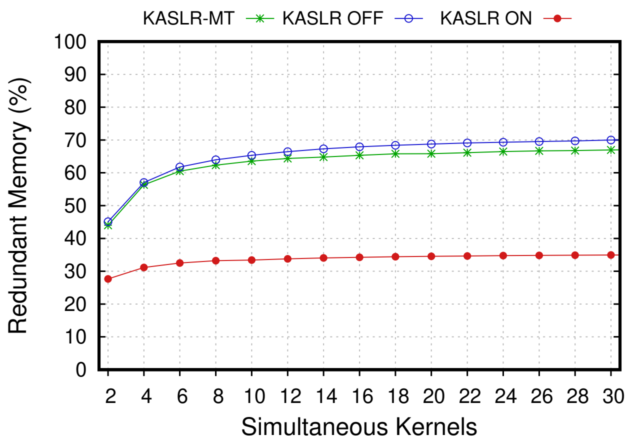

Figure 4: Redundant memory curve for different number of simultaneous kernels.

until value 0x7e ('~'). This is a total of 94 characters in our alphabet of valid characters for a key. Considering a desired entropy of 64 bits, the key size must be at least 10 characters:

$$x * \log_2(94) = 64 \rightarrow x \approx 9.76$$

8. Evaluation

In this section, we evaluate the memory deduplication saving effectiveness of KASLR-MT as well as the security considerations.

8.1. Memory Deduplication Savings

In order to measure how much kernel memory is being saved by deduplication with KASLR-MT compared with Linux KASLR, we have launched another experiment, running from 2 to 30 simultaneous virtual machines. Each virtual machine configuration is the same as used in section 5, a generic GNU/Linux Ubuntu 19.04 with Linux 5, with Gnome desktop and full networking (NAT mode provided by Qemu). The physical machine used to run the experiments has an Intel Xeon W-2155 processor and 32 GiB of SDRAM memory. The hypervisor used is KVM (Linux kernel 4.19-ARCH) along with Qemu VMM version 4.0.0.

Userspace activities modify the kernel state (changes in data structures as existing processes, open files, etc.). Inevitably, even though these alterations are not related with kernel randomization, they are present in the measurements. The possible combinations of userspace workloads are endless. For this reason, virtual machines run the GNU/Linux Ubuntu distribution in the experiment with the aim of obtaining a real environment with a representative userland workload. However, since the scope of this paper is to study the effects that kernel randomization has in memory contents, memory pages belonging to userland should be discarded.

Figure 4 shows the resulting percentage of redundant memory for each case. With only two kernels, if kernel randomization is enabled there is a 27.65% of redundant memory, which increases to 45.09% by disabling it. With our solution, the value obtained is close to the latter, a 44.02%. The percentage increases logarithmically as more kernel memories are added, until they approach a theoretical limit, where the curve grows slower. However, when kernel randomization is enabled, the curve has less slope; this is caused by a greater number of pages with matchless contents. This limit is approximately 70% when kernel randomization is not enabled, 67% in our solution, and 35% when kernel randomization is enabled.

Taking a stabilized case, for example with 30 kernels, we have splitted the memories of all the kernels by region, as shown in figure 5. From it, we can confirm which are the kernel regions benefited from our solution. Linux code region gets exactly the same sharing as when kernel randommization is disabled, while its data region (including `.data`, `.bss` and other data sections) is close, with only 2.73% of sharing loss. Modules code and data regions are also benefited, although there is another factor independent of kernel randomization affecting their contents similarity (elucidated in section 9). Similarly, `vmemmap` can share a 34.89% of its contents with our solution, an 8.29% less than when kernel randommization is disabled; but still a good portion, considering that its contents are almost entriely matchless when kernel randommization is enabled. With regard to `vmalloc`, we can see that it has similar redundant memory, independently of kernel randomization.

8.2. Security Considerations

Regarding security implications of KASLR-MT, it extends the memory saving benefits keeping the protection provided by KASLR.

Our proposed solution is intended to be beneficial in multi-tenant cloud systems, where several tenants are owners of one or more groups of virtual machines, and all of them share the resources of a single physical machine by the use of virtualization technologies.

The trade-off for more memory sharing is to share akin kernel address space layouts among virtual machines belonging to the same group. Effectively, this solution maintains an equivalent protection offered by kernel randomiza-

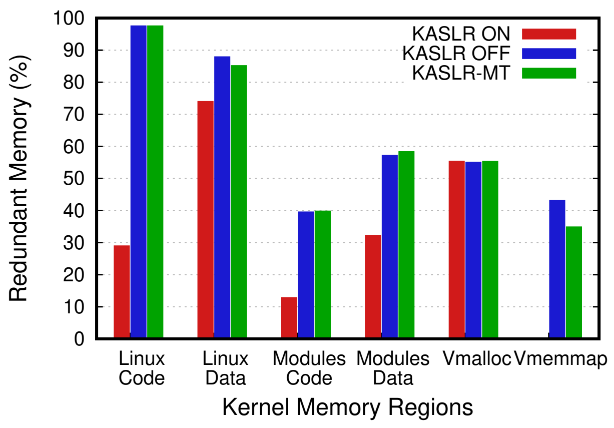

Figure 5: Percentage of redundant memory per kernel region, for 30 simultaneous kernels.

tion. On the one hand, it protects against external attackers including network applications interacting with the kernel and unknown tenants running virtual machines in the same host machine. On the other hand, attacks from those virtual machines that share the same kernel address space layout are equivalent to local attacks (userspace applications attacking its own kernel). In all of these cases, the kernel memory layout is unpredictable from the attacker perspective.

There is a case where KASLR-MT has a drawback: an attacker successfully bypassing the kernel randomization of a particular machine will be able to use that leak to bypass the kernel randomization of another machine belonging to the same tenant. However, the effort required to bypass KASLR-MT is the same as KASLR.

9. Discussion

In this section, we discuss some affairs that have not been thoroughly detailed for being out of scope.

As advanced in sections 5 and 8, the loading order of Linux loadable modules is not deterministic. Thus, even if kernel randomization is disabled, a similar effect occurs in the modules code and data regions, caused by the intrinsic randomization derived from the unpredictability of their loading order. For example, a kernel with only two modules, loading first A and then B in a first execution, could load them in the inverse order after a reboot. Since Linux modules are relocatable objects, and relocations are dynamically patched at load time, part of their contents depends on the virtual address where they are loaded at. For this reason, contents of the modules region of two identical kernels will differ if their modules are loaded in different order, independently of kernel randomization.

Even if a pre-established loading order is determined for a certain group of Linux modules, the boundless variety of different possible modules makes it a complex issue. Following the previous example and without considering kernel randomization, let us assume that we establish an arbitrary order to load three specific modules: first A, followed by B and then C. Two kernels loading only these three modules will match, as desired. However, only if one of them decides not to load module B, the loading address of C will differ.

Taking into account that the list and number of modules being loaded in the kernel of a virtual machine may vary depending on the task in which it is involved, we need to find a different solution for the specific case of loadable modules. Focusing on the root of the problem, content dissimilarity appears because relocations are fixed-up with position-dependent information. Therefore, a plausible way to get more memory sharing chances in loadable modules could be using position-independent code, instead of compile them as relocatable objects. With this approach, some data pages would inevitably have to contain private non-shareable information (e.g., per-module Global Offset Table), but at least the entire code and part of the data regions could be shareable.

10. Conclusions

In this paper we presented KASLR-MT, a new Linux kernel randomization design for multi-tenant cloud systems, compatible with memory deduplication, which maximizes memory savings rate while providing a strong security.

We identified why the most widely and effective technique used to mitigate attacks at kernel level, KASLR, fails to provide protection and shareability at the same time. To design KASLR-MT, we performed a deep analysis on how kernel randomization affects to the shared memory. Then, we propose KASLR-MT, the first effective and practical Kernel ASLR memory protection that maximizes the memory deduplication savings rate while providing a strong security.

We have implemented and tested KASLR-MT in the Linux kernel and our results showed that KASLR-MT is not intrusive, very scalable and maximizes the memory savings rate while providing a strong security.

References

[1] Arcangeli, A., Eidus, I., Wright, C., 2009. Increasing memory density by using ksm, in: Proceedings of the linux symposium, Citeseer. pp. 19–28.

[2] Computer History Museum, . What happened on september 9th. URL: http://www.computerhistory.org/tdih/September/9/. [Retrieved: Sep, 2018].

[3] Filipe, R., Barreto, J., 2011. End-to-end data deduplication for the mobile web, in: 2011 IEEE 10th International Symposium on Network Computing and Applications, IEEE. pp. 334–337.

[4] Fábrega, F.J.T., Javier, F., Guttman, J.D., 1995. Copy on write.

[5] Gisbert, H.M., Ripoll, I., 2014. On the effectiveness of nx, ssp, renewssp, and aslr against stack buffer overflows, in: 2014 IEEE 13th International Symposium on Network Computing and Applications, IEEE. pp. 145–152.

[6] Gueron, S., Langley, A., Lindell, Y., 2017. Aes-gcm-siv: Specification and analysis. IACR Cryptology ePrint Archive 2017, 168.

[7] Gupta, D., Lee, S., Vrable, M., Savage, S., Snoeren, A.C., Varghese, G., Voelker, G.M., Vahdat, A., 2010. Difference engine: Harnessing memory redundancy in virtual machines. Communications of the ACM 53, 85–93.

[8] Kaur, G., Bala, A., Chana, I., 2019. An intelligent regressive ensemble approach for predicting resource usage in cloud computing. Journal of Parallel and Distributed Computing 123, 1

– 12. URL: http://www.sciencedirect.com/science/article/pii/S0743731518306063, doi:https://doi.org/10.1016/j.jpdc.2018.08.008.

[9] Kim, S., Kim, H., Lee, J., Jeong, J., 2014. Group-based memory oversubscription for virtualized clouds. Journal of Parallel and Distributed Computing 74, 2241 – 2256. URL: http://www.sciencedirect.com/science/article/pii/S0743731514000033, doi:https://doi.org/10.1016/j.jpdc.2014.01.001.

[10] Lindemann, J., Fischer, M., 2018. A memory-deduplication side-channel attack to detect applications in co-resident virtual machines, in: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA. pp. 183–192. URL: http://doi.acm.org/10.1145/3167132.3167151, doi:10.1145/3167132.3167151.

[11] Mell, P., Grance, T., et al., 2011. The nist definition of cloud computing.

[12] Miłós, G., Murray, D.G., Hand, S., Fetterman, M.A., 2009. Satori: Enlightened page sharing, in: Proceedings of the 2009 conference on USENIX Annual technical conference, pp. 1–1.

[13] Neves, P., Ferreira, P., Barreto, J., 2013. Leveraging web prefetching systems with data deduplication, in: 2013 IEEE 12th International Symposium on Network Computing and Applications, IEEE. pp. 259–262.

[14] Oliverio, M., Razavi, K., Bos, H., Giuffrida, C., 2017. Secure Page Fusion with VUsion, in: Proceedings of the 26th Symposium on Operating Systems Principles, ACM, New York, NY, USA. pp. 531–545. URL: http://doi.acm.org/10.1145/3132747.3132781, doi:10.1145/3132747.3132781.

[15] PaX, 2003. Pax address space layout randomization (aslr). URL: https://pax.grsecurity.net/docs/aslr.txt. [Retrieved: Sep, 2018].

[16] Payer, M., 2016. HexPADS: A Platform to Detect “Stealth” Attacks, in: Caballero, J., Bodden, E., Athanasopoulos, E. (Eds.), Engineering Secure Software and Systems, Springer International Publishing, Cham. pp. 138–154.

[17] Perla, E., Oldani, M., 2010. A Guide to Kernel Exploitation: Attacking the Core. Syngress Publishing.

[18] Ranjbari, M., Torkestani, J.A., 2018. A learning automata-based algorithm for energy and sla efficient consolidation of virtual machines in cloud data centers. Journal of Parallel and Distributed Computing 113, 55 – 62. URL: http://www.sciencedirect.com/science/article/pii/S074373151730285X, doi:https://doi.org/10.1016/j.jpdc.2017.10.009.

[19] Sharma, P., Kulkarni, P., 2012. Singleton: system-wide page deduplication in virtual environments, in: Proceedings of the 21st international symposium on High-Performance Parallel and Distributed Computing, ACM. pp. 15–26.

[20] Steven, S., 2011. Reducing runtime memory in windows 8. URL: https://blogs.msdn.microsoft.com/b8/2011/10/07/reducing-runtime-memory-in-windows-8/. [Retrieved: Sep, 2018].

[21] Suzaki, K., Iijima, K., Yagi, T., Artho, C., 2011. Memory deduplication as a threat to the guest os, in: Proceedings of the Fourth European Workshop on System Security, ACM, New York, NY, USA. pp. 1:1–1:6. URL: http://doi.acm.org/10.1145/1972551.1972552, doi:10.1145/1972551.1972552.

[22] Tziritas, N., Khan, S.U., Xu, C.Z., Loukopoulos, T., Lalis, S., 2013. On minimizing the resource consumption of cloud applications using process migrations. Journal of Parallel and Distributed Computing 73, 1690 – 1704. URL: http://www.sciencedirect.com/science/article/pii/S0743731513001585, doi:https://doi.org/10.1016/j.jpdc.2013.07.020. heterogeneity in Parallel and Distributed Computing.

[23] Vaño, F., Marco, H., 2018a. How Kernel Randomization is Canceling Memory Deduplication in Cloud Computing Systems, in: 2018 IEEE 17th International Symposium on Network Computing and Applications (NCA), IEEE. pp. 373–376. URL: https://ieeexplore.ieee.org/abstract/document/8548338.

[24] Vaño, F., Marco, H., 2018b. Slicedup: A Tenant-Aware Memory Deduplication for Cloud Computing, in: The Twelfth International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2018), pp. 15–20. URL: https://www.thinkmind.org/download.php?articleid=ubicomm_2018_1_30_10065.

[25] Vmware Docs, 2018. Vmware vsan: Using deduplication and compression. URL: https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.virtualsan.doc/GUID-3D2D80CC-444E-454E-9B8B-25C3F620EFED.html. [Retrieved: Sep, 2018].

[26] Waldspurger, C.A., 2002. Memory resource management in vmware esx server. ACM SIGOPS Operating Systems Review 36, 181–194.

[27] Wang, Z., Sun, D., Xue, G., Qian, S., Li, G., Li, M., 2018. Ada-things: An adaptive virtual machine monitoring and migration strategy for internet of things applications. Journal of Parallel and Distributed Computing URL: http://www.sciencedirect.com/science/article/pii/S0743731518304404, doi:https://doi.org/10.1016/j.jpdc.2018.06.009.

[28] Xu, Z., Liu, G., Wang, T., Xu, H., 2017. Exploitations of uninitialized uses on macos sierra, in: 11th {USENIX} Workshop on Offensive Technologies ({WOOT} 17).

[29] Zhou, H., Li, Q., Choo, K.K.R., Zhu, H., 2018. Dadta: A novel adaptive strategy for energy and performance efficient virtual machine consolidation. Journal of Parallel and Distributed Computing 121, 15 – 26. URL: http://www.sciencedirect.com/science/article/pii/S0743731518304520, doi:https://doi.org/10.1016/j.jpdc.2018.06.011.

Fernando Vano-Garcia is a PhD researcher at the University of the West of Scotland, United Kingdom. His main research interests include cybersecurity, memory management in cloud computing critical infrastructures and virtualization technologies, among others. He is also technical program committee member of international scientific conferences. He completed his BSc Computer Engineering degree at Universitat Politecnica de Valencia, and his MSc in Cybersecurity at Universidad Carlos III de Madrid, Spain.

Hector Marco is an associate professor and cybersecurity researcher at the University of the West of Scotland, UK. He holds a PhD in Computer Science, Cybersecurity, from Universitat Politecnica de Valencia, Spain. Hector is senior member of the Institute of Electrical and Electronics (IEEE), and member of the Engineering and Physical Sciences Research Council (EPSRC) in UK. Previously, he was research associate at the Universitat Politecnica de Valencia where he co-founded the "cybersecurity research group". Hector was part of the team developing the multi-processor version of the XtratuM hypervisor to be used by the European Space Agency in its space crafts. He participated in multiple research projects as Principal Investigator and Co-Investigator. Hector is author of many papers of computer security and cloud computing. He has been invited multiple times to reputed cybersecurity conferences such as Black Hat and DeepSec. Hector has published more than 10 Common Vulnerabilities and Exposures (CVE) affecting important software such as the Linux kernel. He has received honors and awards from Google, Packet Storm Security and IBM for his security contributions to the design and implementation of the Linux ASLR. Hector's professional interests include low level cybersecurity, kernel and userland security, virtualization security and applied cryptography.