IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 12, 2020, accepted August 24, 2020, date of publication August 27, 2020, date of current version September 16, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3019774

An Info-Leak Resistant Kernel Randomization for Virtualized Systems

FERNANDO VANO-GARCIA, (Member, IEEE), AND HECTOR MARCO-GISBERT, (Senior Member, IEEE)
School of Computing, Engineering and Physical Sciences, University of the West of Scotland, Paisley PA1 2BE, U.K.
Corresponding author: Fernando Vano-Garcia (fernando.vano-garcia@uws.ac.uk)

ABSTRACT Given the significance that the cloud paradigm has in modern society, it is extremely important to provide security to users at all levels, especially at the most fundamental ones since these are the most sensitive and potentially harmful in the event of an attack. However, the cloud computing paradigm brings new challenges in which security mechanisms are weakened or deactivated to improve profitability and exploitation of the available resources. Kernel randomization is an important security mechanism that is currently present in all main operating systems. Function-Granular Kernel Randomization is a new step that aims to be the future of the kernel randomization, because it provides much more security than current kernel randomization approaches. Unfortunately, function-granular kernel randomization also impacts significantly on the performance and potential benefits of memory deduplication. Both function-granular kernel randomization and memory deduplication are desired and beneficial; the first for the strong protection it gives, and the second for the reduction of costs in terms of memory consumption. In this paper, we analyse the impact of function-granular kernel randomization on memory deduplication revealing why it cannot offer maximum security and shareability of memory simultaneously. We also discuss the reasons why having a full position independent kernel code counter-intuitively does not solve the problem introducing a challenge to kernel randomization designers. To solve these problems, we propose a function-granular kernel randomization modification for cloud systems that enables full function-granular kernel randomization while reduces memory deduplication cancellations to almost zero. The proposed approach forces guest kernels of the same tenant to have the same random memory layout of memory regions with high impact on deduplication, ensuring a high rate of deduplicated pages while the kernel randomization is fully enabled. Our approach enables cloud providers to have both, high levels of security and an efficient use of resources.

INDEX TERMS Virtualization, security, KASLR, memory deduplication, memory management.

I. INTRODUCTION

Cloud computing has reshaped our society and the way we interact with other people. This paradigm has progressively become the *de-facto* model over the last years, constantly growing and adapting to new challenges to bring us new opportunities and new paths for doing business. Through the use of virtualization technologies, it offers on-demand boundless computing resources, which allows cloud providers to minimize operating costs through *economies of scale* [1]. In addition, it leads to highly energy-efficient [2] infrastructures by aggregating user demands, enabling optimal resource exploitation along with flexible and efficient multiplexing of the workload based on the demand.

Infrastructure as a Service (IaaS) [3] is considered one of the fundamental building blocks of cloud computing. This is in part because users can configure virtualized environments with a high degree of flexibility without having to worry about deploying large physical computing rooms. At this level, the efficient resource management is fundamental to deal with a proper cloud infrastructure [4]–[6]. Hardware resources are a critical asset in the business, and they must be managed and utilized adequately. Cloud service providers will obtain more benefits if they are able to operate more virtual machines with the same resources [7], [8]. Furthermore, IaaS providers must guarantee users that the highest possible level of security is attained to safeguard confidentiality, integrity and availability.

Unfortunately, although cloud providers seek to maximize the security of their infrastructure while maximizing the exploitation of their resources, it is not always possible. Today's security mechanisms are far from perfect and, in many cases, they introduce cost-prohibitive overheads.

The associate editor coordinating the review of this manuscript and approving it for publication was Adnan M. Abu-Mahfouz.

An information leakage or *info-leak* is the result of the exploitation of a vulnerability that allows the attacker to disclose a secret from a memory area, either its content or its location [9]. Attackers typically need to conduct info-leaks to know the targeted memory layout before continuing with subsequent stages of an attack. Without an info-leak, all attacks that require prior knowledge of valid addresses are severely hindered. The importance of providing protection against this type of attacks along with the lack of protection against them in current security mechanisms has motivated the emergence of new proposals to satisfy these needs.

Function-Granular Kernel Address Space Layout Randomization (FG-KASLR) [10] is a novel feature that adds more protection to the widely adopted kernel randomization security mechanism. With it, kernel and modules functions are independently randomized, increasing the entropy of the kernel memory mapping and thus significantly reducing the negative impact of a possible information leak. For that reason, it is a desired kernel hardening technique that will potentially become a mainstream feature in all major operating systems in the near future.

Unfortunately, the design of function-granular kernel randomization is not trivial, since it entails a series of new challenges with significant consequences on the correct performance of the kernel. On the one hand, it can introduce run-time performance overhead compared to the standard kernel randomization [11] for a number of reasons, such as the increment of cache misses when running kernel code. On the other hand, as we have identified in this paper, it also introduces challenges to cost-effective resource management in virtualized systems such as cloud environments. One of them is the introduction of a prohibitive memory overhead due to the memory deduplication cancellation. Since it is not recommended to disable security protections such as kernel randomization, IaaS cloud providers will sustain a forfeit of memory resources.

To overcome this challenging problem, we analyse the impact of function-granular kernel randomization on memory deduplication and we identify why other approaches fail. Finally, we propose a solution that maintains the levels of security provided by the function-granular kernel randomization protection mechanism while reducing the memory deduplication cancellation to almost zero.

The main contributions of this paper are the following:

- We identify the particular reasons that prevent memory deduplication to merge memory contents because of the function-granular kernel randomization protection mechanism.
- We present a comprehensive analysis of the impact of function-granular kernel randomization on memory deduplication, pointing out the elements that have most impact for each affected Linux kernel memory region.
- We propose a function-granular kernel randomization approach that fully protects the kernel while significantly reduces the impact of memory deduplication cancellation, obtaining for most memory regions a similar memory deduplication rate than the one obtained when the randomization is disabled.
- We implement the proposed solution in the Linux kernel v5.5 for the x86_64 architecture, evaluating its effectiveness in terms of memory savings and discussing security aspects.

The rest of the paper is organized as follows. Section II provides a detailed background on the current related work. Section III presents the attacker model. The Linux function-granular kernel randomization is described in Section IV. The new problem arisen from the finer randomization granularity is explained in section V. In section VI, we present an exhaustive analysis of the particular influence that the different randomization elements have on the Linux kernel memory regions. Afterwards, and based on the results obtained from the analysis, section VII describes our proposed solution and its implementation in Linux. Section VIII provides an evaluation of the proposed solution. Section IX contains a brief discussion about clarifications that are out of the scope. Finally, the paper concludes in section X.

II. RELATED WORK

A. KERNEL SAMEPAGE MERGING

Kernel Samepage Merging (KSM) is the implementation of the memory deduplication technique in the Linux operating system. It is a memory-saving technique that consists in identifying identical chunks of physical memory and merge them into a single copy using Copy-On-Write (COW) semantics [12], freeing the redundant duplicates for a more efficient use of memory. In Linux, like in most of the modern operating systems, the entire memory is organized and divided in chunks, known as pages [13]. This enables KSM to merge all of the identical pages into a single copy. KSM handles pages that are present in memory, so that swapped pages are excluded.

The *base* page size supported by Linux is 4096 bytes, although other pages of greater size can also be used. For example, the kernel uses pages of 2 MiB. A greater page size implies less page table entries and less translation lookaside buffer (TLB) faults, resulting in higher performance [14]. However, it also produces a bigger waste of memory due to memory fragmentation and the reduction of chances to find other matching pages [15], because a single different bit prevents the sharing of the entire page.

KSM operates in the kernel, as an optional component of the memory manager. When it is used with virtualization technologies (e.g., in cloud environments), the deduplication is applied to the entire guest memory region corresponding to the virtual machine (often called guest physical memory) [16]. In host machines running several virtual machines, all the pages of all guest physical memories are candidates to be shared, being compared with each other looking for their matches. Figure 1 shows an example of how the physical memory usage of the host machine can be reduced by using memory deduplication. The figure shows two

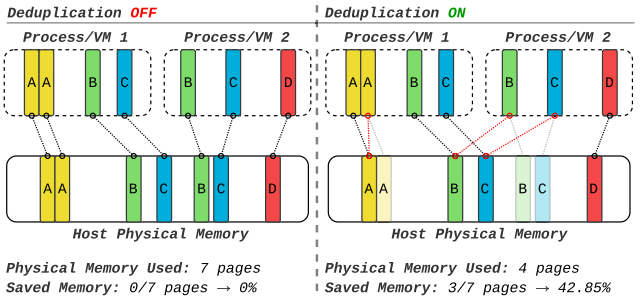

FIGURE 1. Example of memory deduplication. Two processes/virtual machines are shown along with the physical memory of the host machine. The coloured rectangles (A, B, C, D) represent memory pages with different contents. With memory deduplication turned off (left), all pages are present in the physical memory of the host. On the right, when it is turned on, pages with the same contents can be merged, keeping a single physical copy and eliminating redundant ones, thus reducing memory consumption.

processes/virtual machines and the representation of memory pages, comparing the same situation when deduplication is enabled and disabled. In this example, the host machine can save 42.85% of the memory by merging redundant copies of the present pages.

To achieve an efficient search and insertion/deletion, KSM indexes the pages using red-black trees [17], [18]. On the one hand, gathering and comparing all the candidate pages together increases the chances of finding matching pages to be shared and increase the deduplication effectiveness. On the other hand, it enables different types of side-channels [19]–[21] that might compromise the confidentiality. Different solutions have been researched [22]–[24], [24], [25] and, depending on the adversary model of a cloud provider, memory deduplication can be used without the need to sacrifice security.

One of the most relevant weakness to the purpose of this paper is that deduplication is highly dependent on memory contents, and thus extremely susceptible to decreasing its effectiveness upon content variation. This problem occurs for example when two identical objects containing absolute references to their internal parts are loaded in different memory addresses. This leads to variations in the memory contents due to such references, which ultimately affects the deduplication of these contents since they are no longer identical. This problem will be presented in more detail in Section V. Our proposal solves this issue by eliminating memory content variations related with the randomization of kernel regions. In this way, the effectiveness of KSM can be preserved without having to sacrifice its resource exploitation benefits.

Thanks to the memory management virtualization support, it is possible for hypervisors (e.g., Kernel-based Virtual Machine) to implement memory deduplication using pages of 4096 bytes, independently of the page size of the guest view. It is a desired feature for cloud computing providers because of its potential benefits regarding server consolidation [26] due to its ability to reduce the memory footprint across virtual machines [17], [27], decreasing the total cost of managing and ownership.

B. STANDARD LINUX KASLR

Kernel Address Space Layout Randomization (KASLR) or *Kernel Randomization* is a security technique that hinders the successful exploitation of vulnerabilities that rely on knowing addresses [28], [29] of kernel memory regions by randomizing their location at boot-time. This technique performs well without degradation of security after a long time without rebooting the system, even if the machine has a long run time. However, it needs a high-quality entropy source for generating cryptographically secure random numbers to determine the location to load the different memory regions [30], [31]. Otherwise, the final locations would be predictable and then it will be easier for attackers to guess correctly a valid address [32], [33] and bypass the protection.

The general concept of randomizing addresses of memory regions is also widely applied for userland programs [34]. However, the application of this technique in kernel space has completely different implications and limitations than in userland so that, in practice, they are actually two different techniques. For example, the use of position-independent code (PIC) along with a Global Offset Table (GOT) permits to place a userland library on virtually any location without having to modify addresses in the code (e.g. jumps or data access) by using offsets relative to its position; but this approach is not always possible/desired for the kernel. Furthermore, the randomization of kernel memory regions is done at boot-time and it cannot be changed until next reboot, which is much less frequent than restarting userland applications. This has significant direct implications such as the (in)ability to conduct aggressive brute force attacks against the kernel and the relevance of an info-leak to reveal kernel addresses.

The KASLR implementation of the currently recent Linux versions randomizes only the base addresses of several parts of the kernel. For example, the kernel code and data are loaded as a block at a random position, but no randomization is applied between the code and data memory areas. The same applies to the other randomized kernel memory regions, which are each entirely randomized as a single block. From now on, we will refer to this approach as *coarse-grained* kernel randomization. Figure 2 shows an example of how the standard Linux KASLR randomizes the base address of kernel memory regions. It shows two memory regions of a Linux kernel for two different boots: in the first boot (left) KASLR is turned off and in the second boot (right) it is turned on. It also shows an attacker who has control over Region 1 and performs an attack to Region 2. Without the protection of kernel randomization, the location of Region 2 is well known to the attacker, so she has no problem performing her task. In contrast, in the case of the right where KASLR is turned on, since the base address of Region 2 has been randomized, the attacker no longer knows the location of the target. Since the attack cannot be successfully executed without this information, the attack has been prevented.

One of the problems that the standard coarse-grained kernel randomization approach raises is that it is not

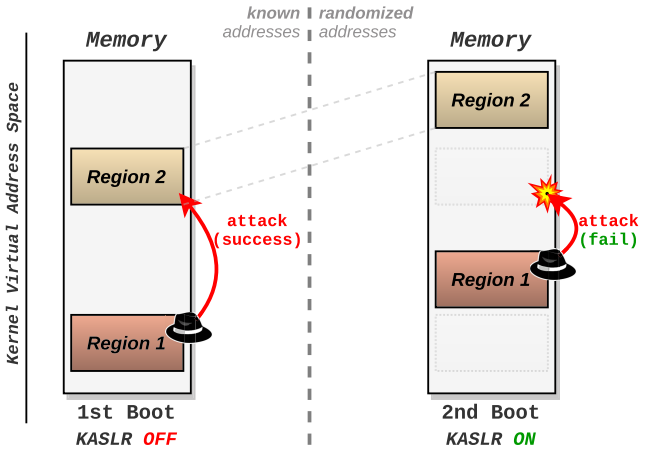

FIGURE 2. Example of standard KASLR showing two kernel memory regions and an ongoing attack from Region 1 to Region 2. For the attack to be successful, the attacker must know the address of Region 2. On the left, since KASLR is turned off, this address is known to the attacker, so the attack succeeds. On the right side, the same situation is shown with KASLR turned on. Since the addresses of the kernel regions have been randomized, the attacker does not know the current location of Region 2 and the attack is prevented.

compatible with memory deduplication when both methods are applied in environments based on virtualization technologies [35]. In addition, since the randomization is applied at boot time and unchanged until next reboot, any info-leak disclosing a specific chunk of a kernel memory region, it is actually uncovering the location of the entire kernel memory region, thus circumventing the KASLR protection for that particular region. Following the example presented in the previous paragraph, for a kernel memory region containing both code and data together, a single info-leak revealing an address pertaining to either the code or the data regions will immediately de-randomize both memory areas. In that case, the KASLR bypass will be effective until next reboot [36], [37]. For this reason, a finer-grained kernel randomization is needed to address such challenges. This is discussed in more detail in section IV. Our proposal offers a finer-grained kernel randomization (at function level) while being fully compatible with memory deduplication, therefore offering higher protection without losing the benefits provided by deduplication.

C. KASLR-MT

Kernel Address Space Layout Randomization Multi-Tenant (KASLR-MT) [35] is a kernel randomization solution for multi-tenant cloud systems that remedies the problem of memory deduplication cancellation caused by the randomization effects on guest memory contents. KSM and KASLR techniques conflict when both are enabled in virtualized systems such as cloud environments [35], [38]. The reason is that the randomization of kernel memory regions causes undesired effects on the memory sharing effectiveness of KSM because the latter tries to merge host memory pages with identical content while guest kernel randomization introduces differences in the memory contents of the guest virtual machine. Cloud systems running multiple virtual machines with several tenants are the most affected by the significant loss of memory sharing opportunities.

To remedy the breakage of memory sharing, kernel layouts of guest virtual machines should be as similar as possible. One possible short-term workaround to solve the problem and restore the benefits of memory deduplication would be to fully disable the protection provided by function-granular kernel randomization in the guest kernels. In this way, the kernel memory regions of all guest virtual machines would always be deterministically located in their corresponding default location without any randomness. This way, it would guarantee a maximum memory sharing because these addresses are statically assigned at compile time. However, this option cannot be considered as acceptable because removing the protection provided by kernel randomization introduces serious security weaknesses. KASLR-MT addresses this problem by enabling the hypervisor to instruct how guests virtual machines map their memory regions. As a result, the deduplication effectiveness of completely disabling kernel randomization is combined with statistical defense provided by the standard coarse-grained kernel randomization protection.

Attacks evolve very quick and protection techniques must be always be revisited, adapted or even fully changed in order to provide an effective protection. Unfortunately, this is the case for KASLR-MT, a recent proposal that is not compatible with function-granular kernel randomization. Since it is based on the coarse-grained kernel randomization approach, it randomizes kernel memory regions as entire blocks. For this reason, in a similar way as with standard KASLR, it does not provide protection against info-leaks. In fact, when advanced approaches for solving this problem such as function-granular randomization are applied, the memory sharing cancellation re-appears due to variations derived from the finer granularity randomization. This problem will be thoroughly detailed in Section V. Our proposal offers additional protection (e.g., against info-leak and correlation attacks) by using a finer grained randomization while preventing the memory sharing cancellation. As a result, we provide function-granular protection and the memory benefits at the same time.

D. COMPARISON WITH PROPOSAL

In order to compare our proposal with the three mechanisms discussed above in this section, Table 1 summarizes the three mechanisms, considering the most relevant strength and weakness for our purpose in this paper and pointing out the main benefits obtained from our proposal.

KSM is a very useful and desired memory saving mechanism that allows providers to reduce the physical memory usage, but unfortunately it is highly dependent on memory contents. For this reason, it is susceptible to a decrement of its effectiveness when memory contents of certain objects are subject to variations. It is the case, for example, when standard Linux KASLR is present in guest virtual machines, which introduces alterations in the memory

TABLE 1. Brief comparison of the three related work mechanisms (KSM, Standard Linux KASLR and KASLR-MT) with our work, showing their most relevant strength and weakness for the purpose of this paper.

Mechanism	Strength	Weakness	Proposal Benefits
KSM	Reduces the total physical memory utilization and memory footprint across virtual machines in virtualized systems.	It is highly dependent on memory contents, thus extremely susceptible to decreasing its effectiveness upon content variation.	Our proposed solution eliminates the differences in memory contents produced by function-granular kernel randomization, enabling the protection while preserving the effectiveness of KSM.
Linux KASLR	Hinders the successful exploitation of unknown attacks against the kernel that rely on knowing valid kernel addresses.	Introduces alterations in the kernel memory contents, causing the cancellation of the memory sharing benefits obtained by KSM. Besides, it does not provide protection against info-leaks.	Our proposed solution provides additional info-leak protection while preventing the sharing cancellation due to alterations in memory contents, thus keeping the benefits provided by deduplication.
KASLR-MT	Fixes the memory sharing cancellation for the coarse-grained kernel randomization approach, such as Standard Linux KASLR.	It is not compatible with fine-grained randomization, and therefore does not provide protection against info-leaks.	As with standard KASLR, our proposed solution solves the memory sharing cancellation while providing info-leak protection thanks to a finer grained kernel randomization.

contents of the guest, causing the cancellation of the memory sharing benefits obtained by KSM in the host. This problem is fixed by KASLR-MT, but unfortunately this approach is still insufficient, as it follows the coarse-grained randomization approach. Therefore, it is still weak to info-leak attacks.

Different from the standard Linux KASLR and KASLR-MT, our proposal solves both problems. On the one hand, it provides additional protection for info-leak and correlation attacks by means of randomizing the kernel memory with a finer-granularity. On the other hand, it eliminates the differences in memory contents produced by function-granular kernel randomization, preserving the effectiveness of KSM. Therefore, we allow guest kernels to enable the additional protection provided by function-granular kernel randomization while keeping it compatible with KSM, offering resource exploitation benefits for host machines.

III. ASSUMPTIONS AND ATTACKER MODEL

As mentioned in Section II-C, disabling kernel randomization in order to have an effective memory deduplication is not a real option since it introduces serious weaknesses that could compromise the full cloud provider. Randomizing the kernel mitigates the exploitation of vulnerabilities relying on knowing kernel virtual memory addresses, for example return-oriented programming (ROP) attacks [28], [39], [40], which allow attackers to dynamically *re-compile* the kernel code at execution time. Without kernel randomization, attacks exploiting those vulnerabilities will always be successful, since those kernel addresses are not a secret that the attackers need to obtain.

In cloud environments, where there are several virtual machines that can directly or indirectly interact with each other, this is even more risky. Any kernel vulnerability could compromise all guest virtual machines of the entire cloud, even if the guests live in different physical machines. In that case, attackers do not need to conduct any prior attack to determine the location of the kernel in memory. The location is well known and the attack will always succeed.

On the other hand, even under the assumption that the protection provided by kernel randomization is present, we must also consider info-leaks, a widely used attack approach to bypass kernel randomization. As briefly introduced in Section I, info-leaks allow attackers to disclose secrets from memory. Info-leaks are listed in the Common Weakness Enumeration (CWE) top 25 most dangerous software errors than can lead to serious vulnerabilities in software [41], occupying the fourth place in the last report (2019). It is extremely important to provide protection against this type of attacks, since it is one of the most effective strategies that attackers can use to bypass KASLR. If this bypass occurs in the current coarse-grained kernel randomization model, the leakage of a single address discloses the location of an entire region. In the case of code regions, this valuable information can be used for subsequent stages of the ongoing exploitation, such as building a ROP payload. In other cases, where the leaked information is actually the secret that the attacker wanted to obtain, the consequences are even worse. Therefore, in any case it is very important to provide protection against info-leaks to minimize the attack surface and thus to mitigate its consequences and to enhance the security provided by the kernel randomization technique.

Therefore, the scope and the goal of our research is to provide a function-granular kernel randomization protection that mitigates info-leak attacks while minimizing to a negligible extent its impact on memory deduplication. This will enable cloud providers to use the protection provided by the newest and more secure kernel randomization techniques without losing the benefits of memory deduplication. It is hard or impossible to assume that operating systems are free from info-leak attacks. Recently, an info-leak in the Linux kernel through 5.7.6 was disclosed under the CVE-2020-15393 identifier [42]. Hence, a practical and effective solution should consider info-leak attacks.

Additionally, our model threat assumes that a local attacker has or can easily get full control of an entire virtual machine and therefore we aim to protect virtual machines of a particular tenant from other tenants and from the Internet. Hence, our

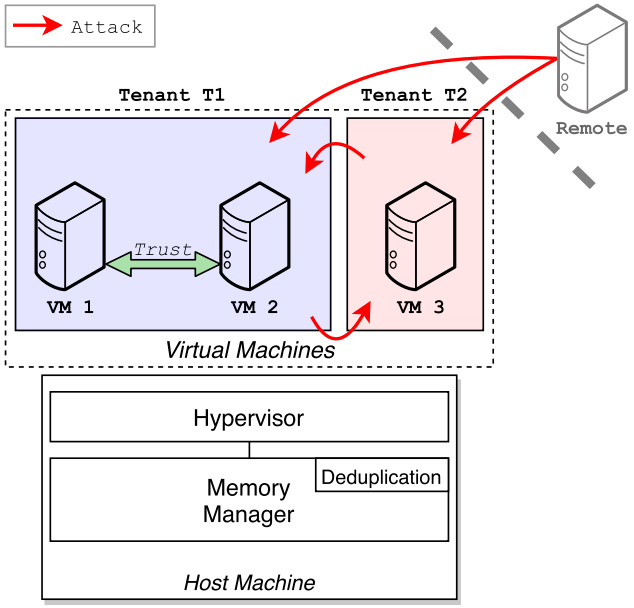

FIGURE 3. Overview of the attacker model, showing an hypervisor running three virtual machines. Two of them (VM−1 and VM−2) belong to one tenant (T1), while the third (VM−3) belongs to another tenant (T2). Attacks from remote machines and from other tenants must face the full info-leak resistant kernel randomization protection to succeed.

goal is that attackers that exploit a kernel vulnerability from remote machines or from virtual machines pertaining to other tenants face the full kernel randomization protection, while keeping the impact on memory deduplication to the minimum. In the event of a successful info-leak attack, the information extracted by the attacker must be insufficient to know the complete layout of the affected kernel memory region.

Figure 3 shows the overview of the described attacker model. The hypervisor is running three virtual machines, two of which (VM−1 and VM−2) belong to one tenant (T1), and the third (VM−3) belongs to another tenant (T2). Attacks from remote machines and from other tenants are covered by the attacker model. It is assumed that virtual machines pertaining to the same tenant trust each other so if one of them is compromised, the others are susceptible to being attacked.

IV. LINUX FUNCTION-GRANULAR KERNEL RANDOMIZATION

As discussed above, kernel randomization is an important security mechanism that is currently present in all main operating systems. Although their different implementation may vary, all of them follow a similar pattern that consists of randomizing both physical and virtual address where the kernel is loaded. Until the moment of writing this paper, implementations randomized only a few random base addresses belonging to different parts of the operating system (coarse-grained kernel randomization). We discussed in Section II-B, that an info-leak disclosing any position of a particular kernel memory region will de-randomize the entire memory region until next reboot. In this regard, even if a given kernel randomization implementation has an extremely high

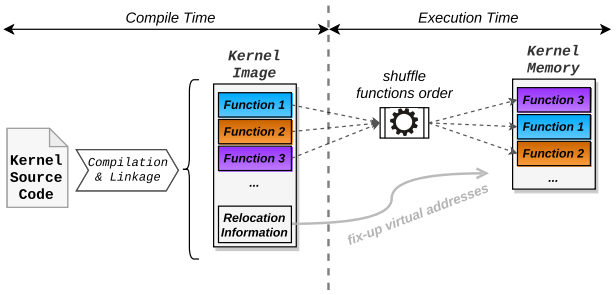

FIGURE 4. General outline of kernel function-granular randomization. The order of kernel functions is dynamically shuffled at boot-time, before the kernel starts its execution.

entropy, a single info-leak cancels it out completely, reducing it to zero.

Function-granular kernel randomization, also known as function re-ordering, is the next step in the progressive development of defense mechanisms designed to protect against respective attack strategies. With function-granular kernel randomization, the kernel and modules functions are randomly shuffled at boot time. Unlike the coarse-grained approach, an info-leak will only de-randomize a very tiny portion of the kernel code or data and will not be valid to de-randomize important functions that attackers typically use to elevate privileges or build ROP attacks. Figure 4 shows a general outline of this concept, where kernel functions are independently randomized by the bootloader at boot-time using a random number generator (RNG).

Function-granular kernel randomization provides not only more entropy but also mitigates info-leaks and correlation attacks [43]. For this reason, it is to be expected that the trend of the kernel randomization protection will be towards a finer grained approach. For example, Linux is currently undergoing early stages of development of a finer-grained kernel randomization approach named Function Granular Kernel Address Space Layout Randomization (FG-KASLR) [10], [44]. This new technique is a step forward for kernel security and it would likely be adopted by other operating systems in the near future.

The implementation of the Linux kernel randomization have been evolving over time. The first randomization additions in the kernel were introduced in version 3.14, in which the physical and virtual addresses were randomized, along with the modules load offset. Later, from version 4.8, kernel physical and virtual addresses were decoupled and randomized separately and three more memory regions were randomized: the physical memory mapping, vmalloc and virtual memory map (the third was added in version 4.9). The last addition was the randomization of the poking address, introduced in version 5.2. This address is used to patch dynamically a running kernel without having to reboot. It is useful to get a highly flexible and adaptable kernel without having to compile different versions depending on the available hardware, such as the number of CPUs.

The first *request for comments* (RFC) patch for function-granular kernel randomization was for the version

TABLE 2. Randomization elements in Linux. All of them must be randomized to have a full and function-granular kernel randomization.

Randomization Elements	Description
Kernel Physical Address	Randomizes the physical address where the kernel is loaded.
Kernel Virtual Address	Randomizes the virtual address where the kernel is loaded.
Function Reordering	Randomizes the kernel and modules *per-function*.
Modules Base	Randomizes the base virtual address where modules are loaded.
Physical Direct Mapping	Randomizes the virtual memory address where physical memory resides.
Vmalloc/ioremap	Randomizes the base address where the kernel allocates dynamic memory.
Vmemmap	Randomizes the base address of the Kernel virtual memory map.
Poking	Randomizes the special virtual address used to patch kernel code dynamically.

5.5, in early 2020 and still under discussion at the time of writing this paper [10], [44]. The current implementation of the Linux kernel is not PIC compliant. It uses text relocations, patching dynamically all the position-dependent references after the final randomized virtual address is settled. The relocation information is generated by the static linker. Then, the linker generated output binary is parsed for relocation information and symbol locations, and generates a simple table of addresses which contain relocations which will need to be adjusted once the final base address of the kernel is determined at boot time.

FG-KASLR follows the same approach going a step further. It uses an existing compiler option to place functions into individual executable code sections at build time. Section header information about these sections is preserved in the final output binary as part of the kernel build process. These section headers contain the address ranges of each individual function that was not collected back into the main executable code segment. The address ranges are used to randomly shuffle and re-layout the kernel image at boot time. After that, relocation information is consulted to patch up the corresponding references.

Previously with simple base address layout randomization, the table of relocations appended to the kernel image during the build process only needed to include absolute relocations and relocations that were relative to other segments which were not moved. Because the relative location of symbols in sections that have been randomized has been changed, the generated table of relocations is substantially expanded to include relative relocations. Each relocation entry in an address range that has been randomized must be updated to reflect the code section's new location. In addition, the Linux kernel creates several data tables of addresses. These tables are inspected and adjusted to reflect any changes in addresses. Most of these tables are required to be sorted by address so that the kernel can do fast table lookups using a *bsearch* algorithm. After the addresses have been adjusted, each table will need to be re-sorted.

FG-KASLR extends the kernel randomization by adding a new step when loading the kernel. After randomizing the kernel base address, the loader is responsible of randomly shuffle the functions order. The same approach is used for loadable modules. As a result, it is required to randomize eight different addresses/offsets/components (*randomization elements*) to have a full FG-KASLR as Table 2 shows.

Although FG-KASLR is a novel and desired hardening technique that brings stronger security to the Linux kernel randomization mechanism, unfortunately, it re-introduces the important issue of memory deduplication cancellation that KASLR-MT managed with coarse-grained kernel randomization. Section V discusses in detail how and why function-granular kernel randomization cancels the KSM benefits.

V. THE PROBLEM: RELATIVE OFFSETS

Kernel randomization can cause undesired effects on the memory sharing effectiveness, especially in virtualized systems. In this section, we discuss the conflict between memory deduplication in the host machine and function-granular kernel randomization in the guest virtual machines.

Section II-C briefly outlines the problem that affects memory sharing and the efficient utilization of memory resources when deduplication mechanisms such as KSM are combined with address randomization security mechanisms such as KASLR, especially in environments that rely on virtualization technologies. This issue was explored in previous research [35], [38], providing solutions for the standard coarse-grained kernel randomization approach. However, motivated by info-leak attacks, newer and more secure kernel randomization are being proposed. Function-granular kernel randomization is an example of this and the technique is currently being tested in the Linux hardened community.

As of the date of writing this paper and to the best of the authors' knowledge, KASLR-MT is the only design that solves the problem of memory deduplication breakage in the presence of kernel randomization in cloud systems without compromising the security of users. Unfortunately, with the appearance of function-granular kernel randomization, the problem re-emerges with new challenges invalidating the current solution.

In standard kernel randomization, the absolute references breaks shareability when the kernel is randomized [35] due to text relocations, dynamically patching, etc. The kernel code is the most affected memory area and this is because all kernel code pages that are dynamically patched by absolute relocations at boot-time cannot be merged. An example that causes

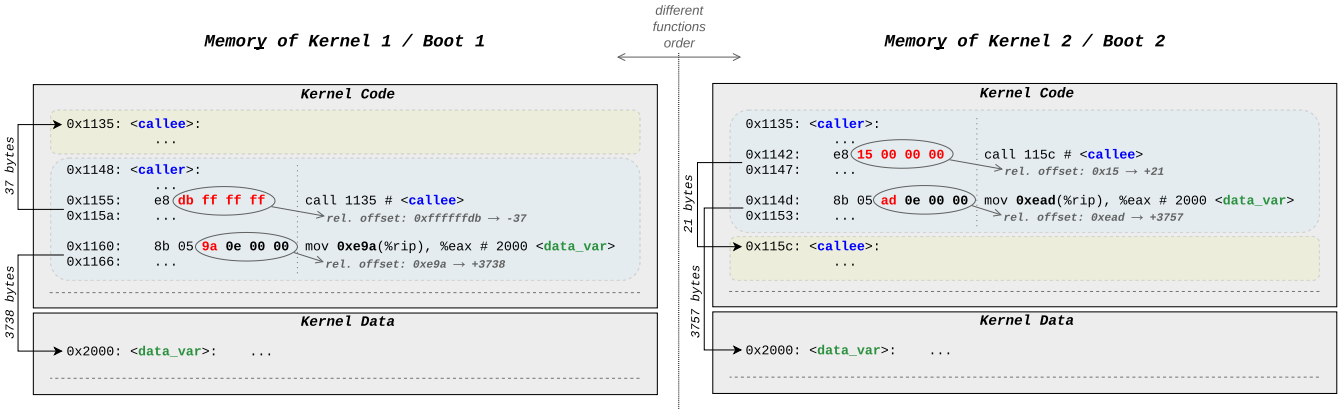

FIGURE 5. Example of memory contents altered by different relative offsets, showing a snippet of the memory of two identical kernels with function-granular kernel randomization. The code represents a `caller()` function that calls another, `callee()`, and accesses a data variable (`data_var`). Because of function-granular kernel randomization, the order of both functions is shuffled at boot-time. Consequently, memory contents differ because the relative offsets are not constant, which breaks their shareability.

this is when a certain kernel function references another one. Unfortunately, this approach randomizes the kernel regions as entire blocks and info-leak attacks will fully bypass the kernel randomization.

In this case, the main culprit is the utilization of absolute relocations. For the currently used coarse-grained kernel randomization, where kernel regions are randomized as entire blocks, alternatives such as compiling the kernel with position-independent code could provide additional benefits and alleviate the problem, because the use of relative offsets in this case do not change the memory contents regardless of the address where the code is loaded.

The main challenge with function granular kernel randomization that invalidates previous solutions is that randomly reordering kernel functions will produce different code. Since function-granular kernel randomization dynamically shuffles the order in which the functions are placed, the relative distance from a given function to another will be different and, therefore, the relocation patch will alter the memory contents. It is important to note that the relative offsets are a problem even if the kernel developers adapt their code to be PIC compliant. Therefore, it is an important problem that cloud providers must overcome, to which there is no available solution at this time.

Since the relative distance between functions is not determined at compile time, function-granular kernel randomization uses more relocations than the coarse-grained kernel randomization. Once all functions are randomly loaded into memory, relative offsets such as those used for calling functions must be relocated. As an illustrative note, the compiled image of the kernel contains approximately 849000 more relocations only by enabling function-granular kernel randomization.

This problem not only affects to code but also to the sharing of data memory. This is because the differences caused by randomization affect data variables when their contents depend on the location of any randomized region being referenced. For example, a function pointer in the kernel data region containing the virtual address of a shuffled function. The kernel contains different structures with pointers referencing to other kernel locations (e.g., functions and variables). Consequently, a single value or pointer that refers to a randomized location breaks the shareability of the entire page. This issue concerns all kernel memory regions containing any kind of data, such as vmalloc, vmemmap and module data.

Figure 5 sketches an example of the problem. It shows a snippet of the memory of two identical kernels with function-granular kernel randomization. Alternatively, it can also be seen as two different boots of the same kernel, since the changes in memory contents occur in the same manner. The example shows the memory content of two functions and a data variable. One of the functions calls the other and later it accesses to the variable in the kernel data region. The most relevant part is the opcode value of the instructions because that is the actual content present in memory. It can be seen in red, that the opcode value of Boot 1 (`dbffffff`) is different from the opcode value of Boot 2 (`15000000`). Those opcode differences are because the distance between the next instruction and the destination is different (different relative offsets).

This is produced when the function-granular randomization shuffles the order in which the functions are loaded. Therefore, the relative distances within a given block are no longer constant. This concerns both distances to other functions and data. To the left of Figure 5, the `callee` function is loaded first at the virtual address 0×1135 followed by the `caller` function at (0×1148). The relative offset in the instruction that calls to `callee` is -37 bytes (backwards), and the distance between the data access instruction and the location of the accessed variable is 3738 bytes. On the right, the order of the functions has been shuffled and the `caller` function is loaded first at 0×1135 followed by `callee` at $0 \times 115c$. The relative offsets are now 21 bytes (forward) to the `callee` function and 3757 bytes to the data variable.

```
0xffffffff81000000 <startup_64>:
0xffffffff81000000:    48 8d 25 51 3f 80 01    lea    0x1803f51(%rip), %rsp
        # dst: 0xffffffff82803f58; rip: 0xffffffff81000007; rel. offset: dst - rip = 0x1803f51

0xffffffff81000007:    e8 e4 00 00 00          callq ffffffff810000f0 <verify_cpu>
        # dst: 0xffffffff810000f0; rip: 0xffffffff8100000c; rel. offset: dst - rip = 0xe4
        ...
0xffffffff81000020:    eb 20                   jmp    ffffffff81000042 <secondary_startup_64+0x12>
        # dst: 0xffffffff81000042; rip: 0xffffffff81000022; rel. offset: dst - rip = 0x20
        ...
0xffffffff81000030 <secondary_startup_64>:
        ...
0xffffffff81000047:    f7 05 8f 2b 63 01 01 00 00 00   testl $0x1, 0x1632b8f(%rip)
        # dst: 0xffffffff82632be0; rip: 0xffffffff81000051; rel. offset: dst - rip = 0x1632b8f
        ...
0xffffffff8100005c:    48 03 05 ad 1f 81 01    add    0x1811fad(%rip), %rax
        # dst: 0xffffffff82812010; rip: 0xffffffff81000063; rel. offset: dst - rip = 0x1811fad
        ...
0xffffffff810000a0:    48 8b 25 a1 68 92 01    mov    0x19268a1(%rip), %rsp
        # dst: 0ffffffff82926948; rip: 0xffffffff810000a7; rel. offset: dst - rip = 0x19268a1
        ...
0xffffffff82e63131 <vmemmap_pte_populate>:
        ...
0xffffffff82e631de:    48 23 35 4b d8 ae ff    and  -0x5127b5(%rip), %rsi
        # dst: 0xffffffff82950a30; rip: 0xffffffff82e631e5; rel. offset: dst - rip = 0xffaed84b (2's)
```

List. 1. Samples of affected x86_64 machine instructions with relative offsets, extracted from the Linux kernel v5.5 binary. The offsets in the instructions' opcode are highlighted in bold to indicate the part of the contents that changes when the functions are shuffled.

These alterations in memory contents are the root of the problem and prevents KSM to merge those pages. Although the figure shows only changes in memory contents of kernel code, the problem is also present in data regions. For example, a data variable containing the virtual address of the `callee` function, since this address is different (0×1135 on the left and $0 \times 115c$ on the right) the variable value in data region will be different. In fact, the problem exists whether this data reference is absolute or relative. It is important to note that, in this illustrative scenario, the first function is always loaded at the same base virtual address for the sake of simplicity. However, the base address of the kernel memory region where the first function is loaded is also randomized.

Therefore, function-granular kernel randomization affects the shareability of kernel memory contents, preventing KSM to merge kernel code and data regions. Even though current Linux kernel implementation is not PIC compliant, there is a widespread presence of instructions with relative offsets, especially on architectures such as x86_64 and ARM with support for calculating addresses relative to the program counter (PC), also known as PC-relative addressing support. Listing 1 shows a few real x86_64 instructions with relative offsets extracted from the Linux v5.5, which confirms that the Linux kernel is affected by the problem. The relative offsets are highlighted and their meaning is detailed through the comments, indicating the subtraction of the destination and the address of the next instruction. For example, the first shown instruction is a Load Effective Address (`lea`) instruction, which is the first instruction executed when the kernel starts running. This instruction loads the address `0xffffffff82803f58` into the `rsp` register. Since the address of the next instruction is `0xffffffff81000007`, the relative offset ($0 \times 1803f51$) is the distance between them. It reveals that the problem affects any instruction that computes a distance relative to its position, regardless of its semantics. This clearly points out that PIC code does not solve the problem.

Memory deduplication and function-granular kernel randomization could actually coexist in a system but, in practice, the deduplication is silently failing. Therefore, knowing the impact of function-granular kernel randomization on deduplication is key to know the extent of the problem and to propose a practical solution.

VI. FUNCTION-GRANULAR RANDOMIZATION IMPACT

Function-granular kernel randomization produces different memory contents across several kernels, reducing the effectiveness of memory deduplication in cloud systems using virtualization technologies. In this section, we present a comprehensive analysis of that randomization impact on memory deduplication for each affected kernel memory region.

Our aim for this section is to measure rigorously to what extent a particular kernel region (Linux code, Linux data, modules code, modules data, vmalloc and vmemmap) changes its contents when different randomization elements of function-granular kernel randomization are enabled (kernel physical address, kernel virtual address, function reordering, modules base, physical mapping, vmalloc, vmemmap and poking virtual address). Given that there are eight randomization elements, we have tested and analyzed $2^8 = 256$ combinations per each kernel memory region. We have executed 4 probes per each combination, being a total of $256 * 4 = 1024$ executions. Then, the percentage of equal pages can be obtained by comparing the probes for each given combination. The OS used for executing the tests is Ubuntu 20.04 LTS with Linux v5.5.

By doing this analysis, we accurately identify **1) the influence** of function-granular kernel randomization for each combination, to identify which randomization elements have more impact on the effectiveness of memory deduplication; **2) the best combination** that maximizes the security provided by kernel randomization while minimizing the negative effects on memory deduplication. This will be key to identify which randomization elements have minimum impact on memory deduplication. As we will discuss later in our proposed approach, those elements can be randomized due to their low or negligible impact. Finally, we obtain **3) non-biased** results that are independent of the amount of virtual machines executed in the experiment. Otherwise, the results obtained might be influenced by merely tweaking a certain number of virtual machines. For example, supposing that 50% of the total memory can be saved when running two identical kernels, then adding a third kernel identical to the others would report 66% of memory saved by deduplication. Although that measure indicates actual memory savings, it is not adequate to determine the real impact of the effects produced by function-granular kernel randomization. For these reasons, instead of calculating the differences of deduplicated memory before and after randomization, we measure the degree of memory contents variation of the kernel memory regions after applying randomization.

Therefore, we focus on calculating the percentage of equal pages for each case, independently of the size of the kernel memory region being analysed. This measure is particularly tailored to measure the impact caused by randomizing kernel areas exclusively, and thus recognizing memory pages that change because of the kernel randomization by comparing two or more memory dumps of the same object. To eliminate non-randomization noise in the measurements, local redundancy of each object is not taken into account when determining the percentage of equal pages. This accurately determines how memory pages are changed due to kernel randomization.

It is important to note that although the analysis shows the results of the Linux kernel code and data regions separated, they are not actually separately randomized in current implementations. Both regions are randomized together in accordance with the obtained kernel virtual address. A part from that, as a feature of function-granular kernel randomization, the order of the code functions is randomly shuffled.

On the other hand, the Linux direct physical mapping is a fairly special case. It is not actually a a real memory region, but a virtual mapping to the entire physical memory. For this reason, the physmap cannot be treated as a virtual memory region with contents and it is discarded in the tests. However, it is included as a randomization element because, although it is purely a virtual mapping, its virtual base address is randomized and this can influence one or more kernel memory regions if they contain values that refer to the randomized virtual address of the direct physical mapping. An example of these references are pointers to dynamically allocated physically contiguous memory areas.

Another important aspect to consider is the page table size. Although Linux kernel guests use 2 MiB pages, the Linux implementation of memory deduplication (KSM) operates with pages of 4 KiB, regardless of the guest's view. In our analysis, we consider that the block size to be compared is 4 KiB, as this is the minimum block size in which KSM operates. As a result, a single bit difference in a 4 KiB block is reported as a mismatch of the entire page, since KSM could not merge it.

Because the total number of combinations obtained in the analysis is too large to be listed ($256*6 = 1536$ rows), Table 3 shows a synthesis of the most significant cases. The table is primarily divided into the six *Linux kernel memory regions* being analysed. The first row of each of the six memory regions is the best case for the memory deduplication. This is when the kernel randomization is disabled. The last row is the worst case, where the function-granular kernel randomization is fully enabled (all the randomization elements are enabled). To effectively summarize from 1536 to 22 rows we are not considering cases whose difference in terms of deduplication is less than 5%. This way, the best possible combination for a given configuration can be reached quickly. For example, in a kernel randomization configuration where the the kernel virtual address is randomized, Table 3 shows that the best combination for the Linux data memory region is the third one, with a 32.6% of randomization overhead. For each kernel memory region, the best combination for the memory deduplication when randomizing the highest number of randomization elements is highlighted.

The results of the randomization effects on the Linux code point out that memory sharing of this region is drastically impacted by the kernel virtual address and by the function reordering. When any of these two are randomized, the percentage of equal pages is reduced from 100% to almost zero, independently of whether other areas are randomized or not. Only a small percentage of pages remain unaffected (0.3 and 0.1 respectively). This reveals that the absolute and relative references are widespread across the kernel code. On the one hand, randomizing the kernel virtual address affects absolute references but not relative ones. This is because even if the kernel is randomized in a coarse-grained way (without function reordering), all memory pages with relocations of absolute addresses break the sharing benefits but relative offsets within the same region are kept intact. On the other hand, the randomization of the functions order not only affects absolute references but also to relative offsets. This is due to the increment of entropy in the order in which kernel functions are placed as Figure 5 shows.

The results for the Linux data memory region show that the best case (kernel randomization off) has 82.5% of equal pages. As it is a data region, it is expected that this region contains non-shareable data, independent of kernel randomization (e.g., timestamps), so obtaining 82.5% of equal pages as the best case is not a surprise. The Linux data memory region is also impacted by the kernel virtual address and by function reordering. Enabling the rest of randomization

TABLE 3. Analysis of the impact that function-granular kernel randomization has on the shareability of Linux memory regions. A black dot (●) indicates that the randomization element is enabled, while a white dot (○) indicates that the randomization element is disabled. The table is divided into six kernel memory regions. For each kernel memory region, the best combination for the memory deduplication when enabling the highest number of randomization elements is highlighted in green background.

Linux Kernel Memory Regions	Randomization Elements								% Equal Pages
	poking vaddr.	vmemmap area	vmalloc/ ioremap	physical mapping	modules base	function reordering	kernel vaddr.	kernel paddr.	
Linux Code	○	○	○	○	○	○	○	○	100
	●	●	●	●	●	○	○	●	100 (-0.0)
	●	●	●	●	●	●	●	●	0.1 (-99.9)
Linux Data	○	○	○	○	○	○	○	○	82.5
	●	●	●	●	●	○	○	●	81.5 (-1.0)
	●	●	●	●	●	○	●	●	49.9 (-32.6)
	●	●	●	●	●	●	○	●	44.3 (-38.2)
	●	●	●	●	●	●	●	●	37.2 (-45.3)
Modules Code	○	○	○	○	○	○	○	○	33.3
	●	●	●	●	○	○	○	●	31.3 (-2.0)
	●	●	●	●	●	○	●	●	8.6 (-24.7)
	●	●	●	●	●	●	●	●	0.1 (-33.2)
Modules Data	○	○	○	○	○	○	○	○	51.2
	●	●	●	●	○	○	○	●	49.8 (-1.4)
	●	●	●	●	○	○	●	●	40.6 (-10.6)
	●	●	●	●	●	○	●	●	35.8 (-15.4)
	●	●	●	●	●	●	●	●	30.8 (-20.4)
Vmalloc Space	○	○	○	○	○	○	○	○	5.0
	●	●	●	●	●	●	●	●	4.0 (-1.0)
Virtual Memory Map	○	○	○	○	○	○	○	○	88.2
	●	○	●	●	●	●	●	●	83.1 (-5.1)
	●	●	●	●	●	●	●	●	0.0 (-88.2)

elements has a small impact on deduplication, just a 1% of overhead. However, only by randomizing the kernel virtual address, the percentage of equal pages drops to less than 50%. Similarly, deduplication is decreased to around 44% by randomizing the functions order. The reason is that Linux data contains references to parts of the kernel itself (to either code and data regions).

Regarding the Linux loadable modules, it could be expected at first glance to obtain an outcome similar to the Linux kernel code and data regions. However, there is a particular aspect in modules that makes them slightly more complex to analyse. When a module is loaded, all their relative references to other modules are patched and therefore its contents depend on the position of other modules. Since the loading order of Linux loadable modules is not deterministic, those memory patches will produce different memory contents and therefore the percentage of equal pages will be reduced. It important to note that once a module is loaded in a different address, all subsequent modules will be affected. Unfortunately, this is out of control of the kernel randomization and for this reason, the best case when the randomization is disabled only reports a 33.3% of equal pages.

The percentage of equal pages for the modules code is reduced by more than 72%, from 31.3% to 8.6% when the kernel virtual address and/or the modules base offset are randomized. However, the randomization of functions has a much greater impact. In this case, it drops almost entirely down to 0.2%. This case is not shown in the table because the difference with respect to the worst case (which is basically adding the kernel virtual address and modules base offset) is only 0.1% and, therefore, less than the 5% established as a threshold. The reasons of this behaviour are the same as with Linux code.

Once a certain module is placed in a memory location randomly assigned to it, and after its functions are shuffled, it is necessary to patch all its relocations. This includes relocations of relative offsets, such as those present in function calls, and relocations of absolute addresses. All of these relocations modify the contents of the module's code region (i.e., by accessing to a variable in its data region). Furthermore, the issue is present not only with references to the module itself but also with any reference to the kernel symbols (e.g., `printk()` function).

The outcome in modules data is similar to modules code, but more relaxed. The three most significant randomization elements that impacts modules are the kernel virtual address, the modules base offset and function reordering. The best case has 51.2% of equal pages, while the following case,

disabling the named addresses, has 49.8%. Of these three randomization elements, the function reordering is the one impacting most, followed by the modules base and the kernel virtual randomization.

The results of the vmalloc kernel region show about 4% to 5% of equal pages in all cases, regardless of the randomization elements being enabled or disabled. This behavior is certainly expected, since this a memory region that serves dynamically allocated virtually contiguous memory that is mainly used to store data temporally. Therefore, based on the obtained results, one can state that function-granular kernel randomization has minimal influence on the contents of the vmalloc memory region.

The last kernel memory region analyzed is the Linux Virtual Memory Map. This region contains lists of objects with references to previous and next objects located in the same region. This aspect is evidenced by the fact that randomizing the vmemmap virtual address has a high deduplication impact on its own memory region. The best case has 88.2% of equal pages, and 83.1% when enabling all randomization elements except vmemmap. Then, the percentage of equal pages drops to zero when the vmemmap virtual address is randomized. In our experiments, we observed that randomizing the kernel physical memory has a slight influence on the vmemmap region. That is reasonable since the objects located in this region represent metadata about the physical memory, so this is caused by information that depends on the physical location of the memory pages such as the page frame number. However, since this influence is less than 5%, the configuration that maximizes the number of enabled randomization elements with minimal impact on memory contents for this region is when all randomization elements are enabled except vmemmap itself.

VII. PROPOSED SOLUTION

Section V states that as of the date of writing this paper and to the best of the authors' knowledge, KASLR-MT is the only kernel randomization design for cloud systems but unfortunately it was not designed with fine-grained kernel randomizations in mind. Therefore there is no solution that provides both info-leak attacks protection and high rates of shared memory desired by cloud providers.

In this section, we present our proposal to enable cloud providers to take advantage of modern kernel randomization protection mechanisms while having high rates of deduplicated memory allowing them to take more profit from their resources and still offer to users the same level of protection as function-granular kernel randomization, including info-leaks. Afterwards, we present the implementation of the proposal for Linux on x86_64.

A. FUNCTION-GRANULAR RANDOMIZATION COMPATIBLE WITH MEMORY DEDUPLICTION

In order to properly tackle the problem of severe memory sharing breakage derived from the memory contents alterations introduced by the randomization of kernel memory regions and functions, we propose an effective function-granular kernel randomization for cloud systems compatible with memory deduplication. The benefits obtained from this proposal include a more efficient use of available resources by removing the cancellation of memory sharing by the influence of relative offsets, as well as a strong protection against info-leaks attacks to guests.

Following the analysis results obtained in Section VI, there are randomization elements with minimum impact and others with higher impact on the effectiveness of memory deduplication. The design of the proposed approach allows the hypervisor to instruct how guests virtual machines map their memory regions. Guests use the information provided by they hypervisor to calculate the addresses of each Linux kernel memory region. This way, the kernel randomization is always enabled and different randomization elements can be tuned depending on the deduplication impact.

A randomization element has **none or low impact** for a given kernel region if the memory contents of that region are not altered when that element is randomized. For example, Table 3 shows that enabling poking *virt.addr.*, vmemmap, vmalloc, physmap, modules offset and kernel *phys.addr.* reduces 0% the number of equal pages for the Linux code memory region. A randomization element has **high impact** if the fact of enabling it has a huge impact on the memory deduplication. For example, Table 3 shows that the function reordering and kernel virtual address randomization elements reduce 99.9% the percentage of equal pages for the Linux code memory region.

The proposed solution is aimed for virtualized systems with multiple tenants running their own virtual machines. Hence, kernels of one tenant share a common randomized memory layout. This increases the shared memory and keeps the effectiveness of the kernel randomization, since the final layout remains unpredictable for an external attacker. This applies also for non-identical virtual machines, because the minimum granularity of KSM is 4096 bytes, and there are still opportunities to deduplicate some parts of the virtual machine. It provides the maximum possible deduplication benefit when the kernel randomization is enabled. Regarding the negative effects on the memory deduplication caused by function-granular randomization, currently there is no other publicly known solution to this problem as it is completely new. As discussed in Section V, approaches such as adapting the kernel to be PIC compliant cannot solve the problem. This is mainly because the function-granular kernel randomization has relative offsets breaking the shareability that unfortunately PIC cannot prevent, since they are already relative.

As depicted in Figure 6, the main idea of the proposed approach consist on having a cloud infrastructure which maintains a table with one-to-one correspondence, linking Tenant-ID and a unique random key. Each unique key serves to feed each virtual machine of a particular tenant, enabling them to deterministically calculate the addresses of all memory regions and the addresses of all shuffled kernel and

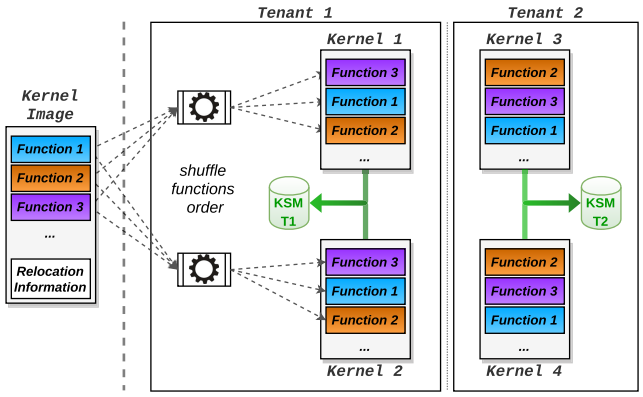

FIGURE 6. Overview of the proposed function-granular kernel randomization. Two tenants are shown, each one running two kernels. The kernels of Tenant 1 determine the same order of functions, which in turn is different from the kernels of Tenant 2. For Tenant 1, the functions in the kernels of Tenant 2 are unknown and unpredictable, and vice-versa. For the kernels of each tenant, KSM (deduplication) can merge the memory contents.

modules functions. Cloud providers must ensure that keys are not duplicated. Besides, since the algorithm to produce addresses for a given key has to be highly deterministic, keys must be totally unpredictable. Otherwise, an attacker could predict the kernel memory layout of a victim.

To provide support for live migration, the relevant columns of this table must be shared among the hosts machines forming the cloud infrastructure. Otherwise, two guests running in two different hosts belonging to the same tenant will have different kernel memory layout and this will prevent to deduplicate the kernel even if both guests are later running under the same host.

A tenant's key lifespan starts from the time the first VM is launched and ends when the last VM is shut down, even if the VMs are running on separate physical machines. Once any VM is running, the key cannot be changed. Otherwise, this would lead to kernels from the same group producing different memory layouts, breaking the shareability. The key of a given tenant can be safely cleared at the moment the last virtual machine is shut down. This key renewal is recommended as it enforces the generation of new kernel memory layouts for subsequent guests. This way, the kernel randomization layout is not always the same but is re-randomized over time. Each randomization element can be individually randomized in two different ways:

1) **Per-Tenant**: For a particular randomization element, a random key is produced by the host and associated with a tenant. The key is shared among all the virtual machines of the same tenant so that they can generate the same base address/offset/shuffle. Guests belonging to the same tenant will have the same (random) memory base address for that particular randomization element. For example, guests produce the same virtual base address of a kernel memory region. Similarly, if the function reordering is randomized Per-Tenant, the kernel and modules functions order will be the same among all guests of the same tenant. Our proposal uses a Per-Tenant approach for randomization elements with high impact on any kernel memory region.
2) **Per-VM**: For a particular randomization element, a random key is produced by the host and associated with a virtual machine. When a guest virtual machine reboots, the base address/offset/shuffle will be different, and it is not shared with other virtual machines belonging to same or other tenants. Our proposal uses a Per-VM approach for randomization elements with low impact on all kernel memory regions.

TABLE 4. Randomization approach for the different Linux randomization components.

Randomization Element	Type
Kernel Physical Address	Per-VM
Kernel Virtual Address	Per-Tenant
Function Reordering	Per-Tenant
Modules	Per-Tenant
Physical Direct Mapping	Per-VM
Vmalloc/ioremap	Per-VM
Vmemmap	Per-Tenant
Poking	Per-VM

B. LINUX IMPLEMENTATION

Based on the results of section VI and considering the two ways to randomize the different randomization elements based on their impact on memory deduplication (low or high), we have implemented the proposed approach maximizing the deduplication while the function-granular kernel randomization is fully enabled. Those correspond with the green rows of Table 3 and the memory sharing rate is similar to that obtained when function-granular kernel randomization is disabled. For this implementation, we consider that a randomization element with high impact is when the reduction of equal pages is greater than 5%. This value depends on the developers decision and Table VI can be consulted to obtain the randomization elements that must be applied for a desired % of equal pages.

It is important to note that the green rows selected to implement our proposal contain randomization elements that are enabled in one row but disabled in another. For example, looking at green rows of Table VI, it can be seen that the kernel virtual address is not randomized for the Linux kernel code memory region but it is for vmalloc and virtual memory map. To properly implement an effective function-granular kernel randomization while keeping high memory sharing rates we must combine all selected rows to obtain which of the randomization elements will be enabled. If a randomization element is enabled in all rows, it will be randomized *Per-Tenant*, otherwise *Per-VM*. Table 4 shows the type applied to each randomization element used in our implementation.

In our proof of concept, the hypervisor passes via kernel's command-line a random key and whether a randomized

```
478 static void shuffle_sections(...)
...     ...
483
--    parse_prandom_seed();
++    if (cloud_fgkaslr_enabled)
++        cloud_fgkaslr_shuffle_init(fgkaslr_seed);
++    else
++        parse_prandom_seed();
485
486   for (i = size - 1; i > 0; i--) {
...     ...
```

List. 2. Modifications to the current `shuffle_sections()` function to enable the proposed solution to generate a deterministic kernel function layout.

element is either Per-VM or Per-Tenant. For the latter, we used a single byte where a `0` means randomization Per-VM and a `1` randomization Per-Tenant. This is all the information required by guests to implement the proposed approach. Since Linux normally prints the contents of the `cmdline` to the kernel ring buffer at the beginning of its boot process, the key is cleaned-up after being retrieved, ensuring that the `cmdline` buffer cannot leak the key. If the key is leaked, every address being randomized from that key could be calculated. The key management can be further secured with advanced leakage-resilience key policies [45], [46].

The hypervisor must ensure that the random key contains enough random material to initialise a cryptographic random number generator. Once the key is obtained by the guest, it is used to derivate all memory addresses to be randomized. In our proof of concept, guests use this key to feed a ChaCha20 cryptographic random number generator [47] to deterministically obtain random numbers to be used to calculate the kernel base virtual address, modules base, vmemmap address as well as the kernel and modules function order. This approach does not add any additional overhead for the system performance nor does it impact on the guest kernel boot time. This is in part because the operation of extracting pseudo-random numbers from a ChaCha20 PRNG is less costly than generating a high quality true random number.

The function reordering element is not just a single address, but a random number per function is obtained to calculate the final memory address where the function will be loaded. As shown in listing 2, since the random number generator has been initialized uniquely using the key from the host, the kernel and modules functions will appear random from the outside. This approach ensures compatibility for future releases of the new Linux kernel randomization.

VIII. EVALUATION

In this section, we evaluate the effectiveness of the proposed solution in terms of security and memory deduplication. We are comparing our proposal against the already existing approaches, FG-KASLR and KASLR-MT, when function-granular kernel randomization and memory deduplication are fully enabled.

To quantify the amount of kernel memory being saved, a second experiment has been conducted. We ran distinct series of simultaneous virtual machines, starting with 2 and gradually adding more VMs up to 30. All these virtual machines use the same configuration as the one mentioned in section VI: a generic GNU/Linux Ubuntu 20.04 LTS with Linux v5.5, 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 (Skylake server microarchitecture) and 32 GiB of SDRAM memory. The hypervisor used is KVM (Linux kernel 5.4-ARCH) along with Qemu VMM version 5.0.0.

The kernel memory space can be modified not only by the kernel itself but also indirectly by the userspace. There are userspace activities that modify the internal state of the kernel. Although such variations are not due to the randomization of the kernel, they may appear in the measurements taken. The number of possible userspace workload combinations is unlimited. Moreover, it would be plausible to create custom workloads to produce biased results, for example, by creating userspace activities that produce a high memory deduplication rate, but also the contrary. For instance, a userland process can indirectly produce certain contents to be in the kernel memory by using any system call that involves a copy from userland to a kernel buffer. The kernel function `copy_from_user()` copies a block of data from the process address space to the kernel, so that copying data with high similarity can cause the shared memory rate to increase. Likewise, copying random data can have the opposite effect. In an effort to be as less biased as possible, our guests run a default GNU/Linux Ubuntu distribution.

A. MEMORY SAVINGS

From the experiment results, we observed that the percentage of redundant memory grows logarithmically over time as more kernels are added. The percentage of redundant memory was stabilized when 30 kernels were running simultaneously and no significant percentage benefit was observed by adding more kernels to our tests. Therefore, we run 30 kernels to obtain results since this enabled us to have results independent of the number of virtual machines. Figure 7 shows a comparison of the percentage of redundant memory when 30 Linux kernels are running, splitting each kernel memory region. This percentage is calculated by observing how many pages KSM can merge for each approach, comparing FG-KASLR (**FG**), KASLR-MT (**MT**), the proposed solution (**P**), and the best case for memory sharing (**MAX**), which corresponds with disabling kernel randomization.

This confirms that previous approaches are cancelling memory sharing by sub-page content modifications due to the function-granular randomization. It also confirms that the proposed approach is close to the best case scenario in terms of redundant memory. The kernel code is the memory region that benefits most, being able to share 97.6% of its memory region which is a significant improvement compared to the 25.9% achieved by the base function-granular kernel randomization. Kernel data is also improved from 72.4% to 88.3%, which means that 15.9% more of the

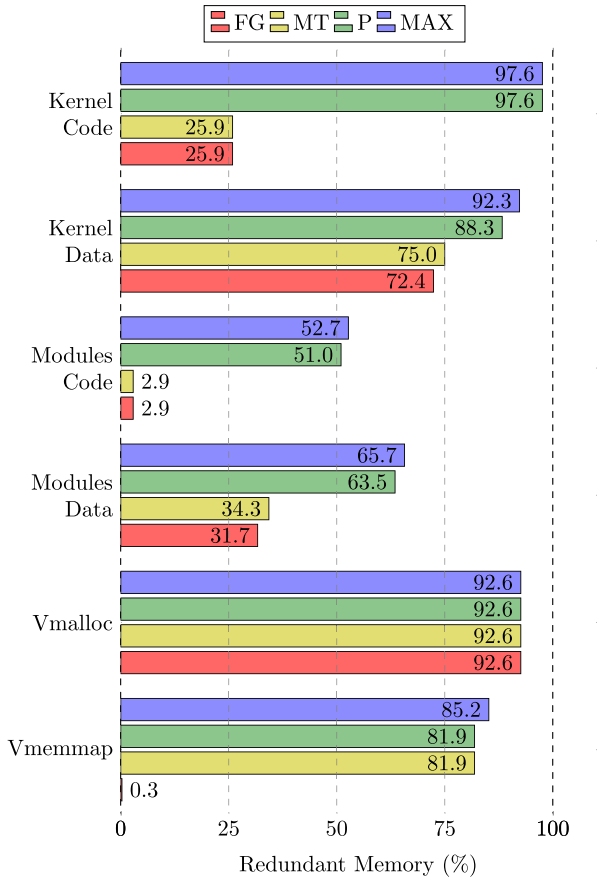

FIGURE 7. Percentage of redundant memory for each kernel memory region, comparing FG-KASLR (FG), KASLR-MT (MT), the proposed solution (P), and the maximum possible value (MAX), which corresponds with disabling kernel randomization.

total memory data will be shared. Modules code shareability improvement goes from 2.9% to 51.0%, which means that half of the modules code memory region is shared in contrast of the 3% of shareability achieved by FG-KASLR. Modules data shareability has been improved by a factor of 2, from 31.7% to 63.5%. Regarding vmalloc, the percentage of memory that can be deduplicated is very high, 92.6% and any improvement would be very low. The vmemmap memory region is the most benefited, since current kernel randomization reduces the percentage of shared memory to 0.3% while our proposal raises this value to 81.9%. Overall, the proposed approach provides a very significant memory deduplication improvement, overcoming all already exiting approaches.

B. SECURITY ASPECTS

The security of the proposed solution lies on hiding the kernel layout to attackers. With our design, the kernel randomization elements (i.e., base addresses of memory regions and functions reordering) are either randomized per tenant or per virtual machine. In comparison with the standard randomization approach, no weaknesses or strengths have been introduced to the randomization algorithm itself, because either randomizing per tenant or per virtual machine, every kernel region will

TABLE 5. Summary of security protection comparing KASLR-MT (MT) and the proposed solution (P). A tick (✓) means that the corresponding approach provides protection against Info-Leak attacks targeting the kernel memory region of the corresponding row.

Attack Target / Kernel Region	Mitigates Info-Leak Attacks to Kernel from:					
	Userspace		Inter-Tenant		Remote	
	MT	P	MT	P	MT	P
Kernel Code	✗	✓	✗	✓	✗	✓
Kernel Data	✗	✓	✗	✓	✗	✓
Modules Code	✗	✓	✗	✓	✗	✓
Modules Data	✗	✓	✗	✓	✗	✓
Vmalloc	✗	✓	✗	✓	✗	✓
Vmemmap	✗	✓	✗	✓	✗	✓
Physical Memory	✗	✓	✗	✓	✗	✓

be seen as a full randomized memory region from outside the tenant. Since the addresses are unknown, exploits requiring absolute addresses will fail. Besides, info-leak attacks trying to de-randomize large blocks of code will also fail due to the protection provided by the function reordering feature. Since the functions order has been randomized, the scope of the info-leak is severely restricted as the attacker can only de-randomize the attacked function, but not the entire code region containing all functions. Consequently, attackers does not have any advantage and the function-granular kernel randomization protection is not weakened.

Table 5 shows a comparison of the proposed solution with the recent KASLR-MT. In the table, a green tick on a cell indicates that the corresponding approach mitigates info-leak attacks targeting the kernel memory region of the corresponding row. For example, the first row corresponds with scenarios where the kernel code memory region is the target of the attack. In the case of an attacking process from userspace is able to exploit an info-leak against this kernel memory region, in the case of KASLR-MT the entire kernel code region will be leaked. The reason is that KASLR-MT follows the coarse-grained approach and it does not offer info-leak protection. However, since our proposal randomizes this region per-function, the scope of the info-leak attack is significantly reduced. The resulting leakage will depend on the implementation and the concrete address being attacked, but in general terms the extent of the attack will be significantly reduced to a small portion of memory (e.g., a single function) instead of the entire memory region. The same procedure applies to all the kernel memory regions, for example in case of a leakage of a function pointer present in the Vmalloc area.

Hence, unlike KASLR-MT, our approach is able to mitigate info-leak attacks from remote attackers such as attacks coming from the Internet, from virtual machines running in other tenants and even from userland, regardless whether the virtual machine being attacked belong to same tenant or not. This provides a wide and reasonable protection level where only administrators with root privileges belonging to the same tenant can attack themselves, which

seems very unlikely. The limitation of this strategy, as with KASLR-MT, is that it is assumed that virtual machines of the same tenant trust each other, as indicated in the attacker model (section III). Consequently, if this condition is not met, providers will not be able to get the full benefits of our proposal. Therefore, if a lack of trust exists between virtual machines of the same tenant, the guest kernel regions will be randomized per-vm, which will imply a higher cost of memory usage. The rest of the scenarios are covered and protected by the proposal.

Finally, in addition to info-leak attacks prevention, our approach also mitigates code reuse attacks, since addresses will appear fully random for attackers. This is because randomizing at function level not only mitigates info-leaks but makes harder code reuse attacks. Therefore, attacks such as malicious installed applications, remote network applications interacting with the kernel and attacks coming from other tenants will face the maximum protection that function kernel randomization provides. As a result, attackers will not find any advantage to bypass the kernel randomization of any of the memory regions that were pre-determined by our proposal. The proposed solution is as secure as the original kernel randomization, whose security has been already proved [37], [48]–[50], but our proposal enables to have both security and high memory saving.

IX. ADDITIONAL GAINS DISCUSSION

In this section, we discuss some affairs that have not been thoroughly detailed for being out of scope but have positive security implication in our proposed approach.

As described in Section III and throughout the article, the goal of our approach is to protect the kernel against attacks from remote machines and from virtual machines belonging to other tenants located in the same physical machine. Our proposal can also provide protection against userland attacks as discussed in Section VIII-B and it assumes that all machines belonging to the same tenant are trustworthy. However, depending on whether the randomization of a memory region is per virtual machine or per tenant, the final implementation could also extend its protection against attacks coming from the same tenant.

Based on the results obtained in the analysis section VI to obtain a reasonable trade-off (less than 5% of deduplication loss) we decided to randomize some memory regions per tenant and others per virtual machine. Table 4 shows that the kernel physical address, the physical direct mapping, vmalloc and the poking virtual address are randomized per virtual machine. Therefore the locations of those memory regions will be different among the virtual machines running in the same tenant. In practice, this means that a virtual machine cannot use its own memory layout to determine the location of those memory regions of another virtual machine running in the same tenant.

Hence, attacks relying on knowing the addresses of those memory regions will be prevented even if the attack comes from a virtual machine of the same tenant. For example, exploits targeting physical memory such as abusing the virtual dynamic shared object (vDSO) by altering paging structures [51] will fail since they require the physical address where the kernel was loaded. In general, addresses randomized per virtual machine protect the associated memory regions even from attacks coming form virtual machines running in the same tenant.

TABLE 6. Summary of additional protection benefits in cases out of the attacker model. Attacks from userspace will always need to face the full function-granular kernel randomization protection. For attacks from the kernel of the same virtual machine or from other from the same tenant will also need to face the kernel randomization protection for those memory regions being randomized Per-VM. For attacks from the kernel, it is assumed that the address of the targeted kernel region is unknown for the attacker.

Attack Source		Additional Protection
Same Tenant	Kernel	Per-VM
	Userspace	Always
Same VM	Kernel	Per-VM
	Userspace	Always

Therefore, the design allows a flexible configuration that can be adapted to different needs. The choice we made regarding which memory regions are randomized per tenant and which per virtual machine showed in this article are the ones that present a reasonable trade-off between security and memory deduplication benefits. However, from the analysis provided in Section VI, other configurations can be used to provide protection against attacks coming from the same tenant.

Finally, the design is flexible enough to provide different randomization information to different virtual machines of the same tenant, allowing special virtual machines to increase its shareability or security. For example, a tenant running 10 virtual machines could have one of them more exposed and therefore a wise decision could be to have more memory regions randomized per virtual machine rather than per tenant. Under this scenario, 9 virtual machines will have a good balance between security (see the security protections in Table 5) and shareability, and the one more exposed will slightly reduce the overall shared memory to have a full independent kernel randomization.

Table 6 presents a summary of the additional protection benefits that are provided by our proposal even though they are not included in the attacker model, as discussed in this section. It shows that all attacks originated from userspace are fully protected by our proposal, either from the same virtual machine or from another of the same tenant. Similarly, attacks originated from other memory regions within the kernel side are protected if the targeted region has been randomized using the Per-VM randomization type.

X. CONCLUSION

Function-granular kernel randomization aims to be the future kernel randomization by improving the current kernel randomization approach not only by extending the entropy

but also by preventing info-leaks and correlation attacks. Unfortunately, it also prevents memory deduplication to work effectively reducing to almost zero the number of pages that can be shared among the same kernel running on different virtual machines. For some memory regions, our analysis showed that the number of pages that can be merged among all kernel memory regions falls drastically when the function-granular kernel randomization is enabled.

After identifying why function-granular kernel randomization fails to provide protection and shareability at the same time, we performed a comprehensive analysis of its negative impact on memory deduplication. Then, based on the analysis results, we proposed an effective and practical function-granular kernel randomization approach for cloud systems that provides high rates of memory sharing and keeps security.

We have implemented our proposal in the Linux kernel version 5.5 and results showed that the proposed approach maximizes the memory deduplication savings rate while providing a strong security equivalent to FG-KASLR. This enables cloud providers to have both, high levels of security and an efficient use of resources.

In summary, our proposal allows guest kernels to enable the additional protection for info-leak and correlation attacks by means of randomizing the kernel memory with a finer-granularity. On the other hand, it eliminates the differences in memory contents produced by function-granular kernel randomization, preserving the effectiveness of memory deduplication techniques such as KSM and offering resource exploitation benefits for host machines.

ACKNOWLEDGMENT

Thanks to K. Accardi for her Linux patches and feedback on the Function Granular Kernel Address Space (FG-KASLR).

REFERENCES

[1] P. Krugman, "Scale economies, product differentiation, and the pattern of trade," *Amer. Econ. Rev.*, vol. 70, no. 5, pp. 950–959, 1980.

[2] A. Berl, E. Gelenbe, M. Di Girolamo, G. Giuliani, H. De Meer, M. Q. Dang, and K. Pentikousis, "Energy-efficient cloud computing," *Comput. J.*, vol. 53, no. 7, pp. 1045–1051, 2010.

[3] P. Mell and T. Grance. (2011). *The NIST Definition of Cloud Computing*. Accessed: Jul. 25, 2020. [Online]. Available: https://csrc.nist.gov/publications/detail/sp/800-145/final

[4] N. Tziritas, S. U. Khan, C.-Z. Xu, T. Loukopoulos, and S. Lalis, "On minimizing the resource consumption of cloud applications using process migrations," *J. Parallel Distrib. Comput.*, vol. 73, no. 12, pp. 1690–1704, Dec. 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731513001585, doi: 10.1016/j.jpdc.2013.07.020.

[5] H. Zhou, Q. Li, K.-K.-R. Choo, and H. Zhu, "DADTA: A novel adaptive strategy for energy and performance efficient virtual machine consolidation," *J. Parallel Distrib. Comput.*, vol. 121, pp. 15–26, Nov. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518304520, doi: 10.1016/j.jpdc.2018.06.011.

[6] G. Kaur, A. Bala, and I. Chana, "An intelligent regressive ensemble approach for predicting resource usage in cloud computing," *J. Parallel Distrib. Comput.*, vol. 123, pp. 1–12, Jan. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518306063, doi: 10.1016/j.jpdc.2018.08.008.

[7] M. Ranjbari and J. A. Torkestani, "A learning automata-based algorithm for energy and SLA efficient consolidation of virtual machines in cloud data centers," *J. Parallel Distrib. Comput.*, vol. 113, pp. 55–62, Mar. 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S074373151730285X, doi: 10.1016/j.jpdc.2017.10.009.

[8] Z. Wang, D. Sun, G. Xue, S. Qian, G. Li, and M. Li, "Ada-things: An adaptive virtual machine monitoring and migration strategy for Internet of Things applications," *J. Parallel Distrib. Comput.*, vol. 132, pp. 164–176, Oct. 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731518304404, doi: 10.1016/j.jpdc.2018.06.009.

[9] F. J. Serna, "The info leak era on software exploitation," Black Hat USA, Las Vegas, NV, USA, Tech. Rep., 2012.

[10] K. Accardi. (2020). *Function Granular KASLR*. Accessed: Jul. 25, 2020. [Online]. Available: https://lwn.net/Articles/821207/

[11] M. Larabel. (Jun. 2020). *Benchmarking the Performance Overhead to Linux's Proposed FGKASLR Security Feature*. Accessed: Jul. 25, 2020. [Online]. Available: https://www.phoronix.com/scan.php?page=article&item=kaslr-fgkaslr-benchmark

[12] F. J. T. Fábrega, F. Javier, and J. D. Guttman, "Copy on write," MITRE Corp., McLean, VA, USA, Tech. Rep., 1995.

[13] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in multics," in *Proc. ACM Symp. Oper. Syst. Princ. (SOSP)*, 1967, pp. 1–12.

[14] S. Mittal, "A survey of techniques for architecting TLBs," *Concurrency Comput., Pract. Exper.*, vol. 29, no. 10, p. e4061, 2017.

[15] F. Guo, Y. Li, Y. Xu, S. Jiang, and J. C. Lui, "Smartmd: A high performance deduplication engine with mixed pages," in *Proc. USENIX Annu. Tech. Conf. (USENIXATC)*, 2017, pp. 733–744.

[16] A. Binu and G. S. Kumar, "Virtualization techniques: A methodical review of XEN and KVM," in *Proc. Int. Conf. Adv. Comput. Commun.* Berlin, Germany: Springer, 2011, pp. 399–410.

[17] A. Arcangeli, I. Eidus, and C. Wright, "Increasing memory density by using KSM," in *Proc. Linux Symp.*, 2009, pp. 19–28.

[18] N. Rauschmayr and A. Streit, "Reducing the memory footprint of parallel applications with KSM," in *Facing the Multicore-Challenge III*. Berlin, Germany: Springer, pp. 48–59.

[19] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Secur. Privacy Mag.*, vol. 8, no. 6, pp. 40–47, Nov./Dec. 2010, doi: 10.1109/MSP.2010.187.

[20] K. Suzaki, K. Iijima, T. Yagi, and C. Artho, "Memory deduplication as a threat to the guest OS," in *Proc. 4th Eur. Workshop Syst. Secur. (EUROSEC)*, New York, NY, USA, 2011, pp. 1:1–1:6. [Online]. Available: http://doi.acm.org/10.1145/1972551.1972552, doi: 10.1145/1972551.1972552.

[21] J. Lindemann and M. Fischer, "A memory-deduplication side-channel attack to detect applications in co-resident virtual machines," in *Proc. 33rd Annu. ACM Symp. Appl. Comput. (SAC)*, New York, NY, USA, 2018, pp. 183–192. [Online]. Available: http://doi.acm.org/10.1145/3167132.3167151, doi: 10.1145/1972551.1972552.

[22] F. Vano-Garcia and H. Marco-Gisbert, "Slicedup: A tenant-aware memory deduplication for cloud computing," in *Proc. 12th Int. Conf. Mobile Ubiquitous Comput., Syst., Services Technol. (UBICOMM)*, Nov. 2018, pp. 15–20. [Online]. Available: https://www.thinkmind.org/download.php?articleid=ubicomm_2018_1_30_10065

[23] M. Oliverio, K. Razavi, H. Bos, and C. Giuffrida, "Secure page fusion with VUsion," in *Proc. 26th Symp. Oper. Syst. Princ. (SOSP)*, New York, NY, USA, 2017, pp. 531–545. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132781, doi: 10.1145/3132747.3132781.

[24] S. Kim, H. Kim, J. Lee, and J. Jeong, "Group-based memory oversubscription for virtualized clouds," *J. Parallel Distrib. Comput.*, vol. 74, no. 4, pp. 2241–2256, Apr. 2014, doi: 10.1016/j.jpdc.2014.01.001.

[25] M. Payer, "HexPADS: A platform to detect 'stealth' attacks," in *Engineering Secure Software and Systems*, J. Caballero, E. Bodden, and E. Athanasopoulos, Eds. Cham, Switzerland: Springer, 2016, pp. 138–154.

[26] C.-R. Chang, J.-J. Wu, and P. Liu, "An empirical study on memory sharing of virtual machines for server consolidation," in *Proc. IEEE 9th Int. Symp. Parallel Distrib. Process. With Appl.*, May 2011, pp. 244–249.

[27] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," *Commun. ACM*, vol. 53, no. 10, pp. 85–93, 2010.

[28] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proc. 14th ACM Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313, doi: 10.1145/1315245.1315313.

[29] B. Luo, Y. Yang, C. Zhang, Y. Wang, and B. Zhang, "A survey of code reuse attack and defense," in *Proc. Int. Conf. Intell. Interact. Syst. Appl.* Cham, Switzerland: Springer, 2018, pp. 782–788.

[30] W. Herlands, T. Hobson, and P. J. Donovan, "Effective entropy: Security-centric metric for memory randomization techniques," in *Proc. 7th Workshop Cyber Secur. Experimentation Test (CSET)*, 2014, pp. 1–8.
[31] H. Marco-Gisbert and I. R. Ripoll, "Address space layout randomization next generation," *Appl. Sci.*, vol. 9, no. 14, p. 2928, Jul. 2019.
[32] J. Ganz and S. Peisert, "ASLR: How robust is the randomness?" in *Proc. IEEE Cybersecur. Develop. (SecDev)*, Sep. 2017, pp. 34–41.
[33] H. M. Gisbert and I. Ripoll, "On the effectiveness of NX, SSP, RenewSSP, and ASLR against stack buffer overflows," in *Proc. IEEE 13th Int. Symp. Netw. Comput. Appl.*, Aug. 2014, pp. 145–152, doi: 10.1109/NCA.2014.28.
[34] PaX. (2003). *PaX Address Space Layout Randomization (ASLR)*. Accessed: Jul. 25, 2020. [Online]. Available: https://pax.grsecurity.net/docs/aslr.txt
[35] F. Vano-Garcia and H. Marco-Gisbert, "KASLR-MT: Kernel address space layout randomization for multi-tenant cloud systems," *J. Parallel Distrib. Comput.*, vol. 137, pp. 77–90, Mar. 2020, doi: 10.1016/j.jpdc.2019.11.008.
[36] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 368–379.
[37] Y. Jang, S. Lee, and T. Kim, "Breaking kernel address space layout randomization with Intel TSX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur. (CCS)*, New York, NY, USA, 2016, pp. 380–392. [Online]. Available: http://doi.acm.org/10.1145/2976749.2978321, doi: 10.1145/2976749.2978321.
[38] F. Vano-Garcia and H. Marco-Gisbert, "How kernel randomization is canceling memory deduplication in cloud computing systems," in *Proc. IEEE 17th Int. Symp. Netw. Comput. Appl. (NCA)*, 2018, pp. 373–376. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/8548338
[39] J. Moreira, S. Rigo, M. Polychronakis, and V. P. Kemerlis, "DROP THE ROP fine-grained control-flow integrity for the Linux kernel," Black Hat Asia, Singapore, Tech. Rep., 2017.
[40] T. Shuo, H. Yeping, and D. Baozeng, "Prevent kernel return-oriented programming attacks using hardware virtualization," in *Proc. Int. Conf. Inf. Secur. Pract. Exper.* Berlin, Germany: Springer, 2012, pp. 289–300.
[41] MITRE. (Sep. 18, 2019). *CWE Top 25 Most Dangerous Software Errors*. Accessed: Jul. 27, 2020. [Online]. Available: https://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html
[42] (Jun. 29, 2020). *CVE-2020-15393*. Accessed: Jul. 25, 2020. [Online]. Available: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-15393
[43] H. Marco-Gisbert and I. Ripoll, "Offset2lib: On the effectiveness of full-ASLR on 64-bit Linux," in *Proc. In-Depth Secur. Conf.*, 2014, pp. 1–9.
[44] E. Jake. (2020). *LWN—Finer-Grained Kernel Address-Space Layout Randomization*. Accessed: Jul. 25, 2020. [Online]. Available: https://lwn.net/Articles/812438/
[45] J. Li, Q. Yu, and Y. Zhang, "Hierarchical attribute based encryption with continuous leakage-resilience," *Inf. Sci.*, vol. 484, pp. 113–134, May 2019.
[46] J. Li, Q. Yu, Y. Zhang, and J. Shen, "Key-policy attribute-based encryption against continual auxiliary input leakage," *Inf. Sci.*, vol. 470, pp. 175–188, Jan. 2019.
[47] D. J. Bernstein, "ChaCha, a variant of salsa20," in *Proc. Workshop Rec. SASC*, vol. 8, 2008, pp. 3–5.
[48] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is dead: Long live KASLR," in *Proc. ESSoS*, 2017, pp. 161–176.
[49] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A.-R. Sadeghi, "Lazarus: Practical side-channel resilient kernel-space randomization," in *Proc. Int. Symp. Res. Attacks, Intrusions, Defenses*. Cham, Switzerland: Springer, 2017, pp. 238–258.
[50] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break it, fix it, repeat," in *Proc. 15th ACM Asia Conf. Comput. Commun. Secur. (AsiaCCS)*, 2020, pp. 1–13.
[51] N. A. Economou and E. E. Nissim. (2016). Getting physical: Extreme abuse of Intel based paging systems. Canada Global Defence and Security Trade Show. [Online]. Available: https://cansecwest.com/slides/2016/CSW2016_Economou-Nissim_GettingPhysical.pdf

FERNANDO VANO-GARCIA (Member, IEEE) received the B.Sc. degree in computer engineering from Universitat Politècnica de València and the M.Sc. degree in cybersecurity from Universidad Carlos III de Madrid, Spain. He is currently a Ph.D. Researcher with the University of the West of Scotland, U.K. His main research interests include cybersecurity, memory management in cloud computing, critical infrastructures and virtualization technologies, among others. He has participated in research projects as Co-Investigator. He is author of many articles of computer security in operating systems and cloud computing. He has also contributed on several occasions as a reviewer for international scientific conferences and reputable scientific journals.

HECTOR MARCO-GISBERT (Senior Member, IEEE) received the Ph.D. degree in computer science, cybersecurity from Universitat Politecnica de Valencia, Spain. He is currently an Associate Professor and a Cybersecurity Researcher with the University of the West of Scotland, U.K. He is a member of the Engineering and Physical Sciences Research Council (EPSRC), U.K. Previously, he was a Research Associate with the Universitat Politecnica de Valencia, where he co-founded the cybersecurity research group. He was part of the team developing the multi-processor version of the XtratuM hypervisor to be used by the European Space Agency in its spacecrafts. He participated in multiple research projects as a Principal Investigator and Co-Investigator. He is author of many articles of computer security and cloud computing. He has been invited multiple times to reputed cybersecurity conferences such as Black Hat, DeepSec, and Stanford University. He has published more than ten 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. His professional interests include low level cybersecurity, kernel and userland security, virtualization security, and applied cryptography.