Adelie: Continuous Address Space Layout Re-randomization for Linux Drivers

Ruslan Nikolaev*
rnikola@psu.edu
The Pennsylvania State University
University Park, PA, USA

Hassan Nadeem
hnadeem@vt.edu
Virginia Tech
Blacksburg, VA, USA

Cathlyn Stone
stonecat@vt.edu
Virginia Tech
Blacksburg, VA, USA

Binoy Ravindran
binoy@vt.edu
Virginia Tech
Blacksburg, VA, USA

ABSTRACT

While address space layout randomization (ASLR) has been extensively studied for user-space programs, the corresponding OS kernel's KASLR support remains very limited, making the kernel vulnerable to just-in-time (JIT) return-oriented programming (ROP) attacks. Furthermore, commodity OSs such as Linux restrict their KASLR range to 32 bits due to architectural constraints (e.g., x86-64 only supports 32-bit immediate operands for most instructions), which makes them vulnerable to even unsophisticated brute-force ROP attacks due to low entropy. Most in-kernel pointers remain static, exacerbating the problem when pointers are leaked.

Adelie, our kernel defense mechanism, overcomes KASLR limitations, increases KASLR entropy, and makes successful ROP attacks on the Linux kernel much harder to achieve. First, Adelie enables the *position-independent code* (PIC) model so that the kernel and its modules can be placed anywhere in the 64-bit virtual address space, at any distance apart from each other. Second, Adelie implements *stack re-randomization* and *address encryption* on modules. Finally, Adelie enables *efficient continuous KASLR* for modules by using the PIC model to make it (almost) impossible to inject ROP gadgets through these modules regardless of gadget's origin.

Since device drivers (typically compiled as modules) are often developed by third parties and are typically less tested than core OS parts, they are also often more vulnerable. By fully re-randomizing device drivers, the last two contributions together prevent most JIT ROP attacks since vulnerable modules are very likely to be a starting point of an attack. Furthermore, some OS instances in virtualized environments are specifically designated to run device drivers, where drivers are the primary target of JIT ROP attacks. Using a GCC plugin that we developed, we automatically modify different kinds of kernel modules. Since the prior art tackles only user-space programs, we solve many challenges unique to the kernel code. Our evaluation shows high efficiency of Adelie's approach: the overhead of the PIC model is completely negligible and re-randomization cost remains reasonable for typical use cases.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; • **Software and its engineering → Operating systems**.

KEYWORDS

return-oriented programming (ROP), address space layout randomization (ASLR), position-independent code (PIC), operating system

ACM Reference Format:
Ruslan Nikolaev, Hassan Nadeem, Cathlyn Stone, and Binoy Ravindran. 2022. Adelie: Continuous Address Space Layout Re-randomization for Linux Drivers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '22), February 28 – March 4, 2022, Lausanne, Switzerland.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3503222.3507779

1 INTRODUCTION

As the sophistication of security attacks and countermeasures grows for user space programs, OS kernels attract an increasing number of attackers, with an ever increasing number of kernel vulnerabilities being discovered in the code of popular commodity OSs [22–24]. Certain vulnerabilities such as CVE-2018-14634 are very serious and seem to have existed for over a decade [7]. Vulnerabilities – the starting points of attacks – are even more likely to be present in device drivers [14, 18, 43, 50] since they are typically not as rigorously tested as core kernel components, as each installation uses only a subset of drivers. Moreover, the number of common vulnerabilities and exposures (CVE) calculated specifically for drivers continues to increase across different OSs exponentially (Figure 1).

There are several reasons why OS kernels are attractive to attackers. First and foremost, defense against attacks is typically more challenging in the kernel space, which includes low-level code involving system calls, device drivers, and interrupt handlers. Second, defense mechanisms for kernel space vulnerabilities (and their inclusion in commodity OSs) often lag behind their user space counterparts. A case in point: address space layout randomization (ASLR) for the Linux kernel [45], a well-known technique against control-flow attacks, is limited to a paltry 2GB range on x86-64 due to instruction immediate operand constraints [37]. Attackers

*Most of the work was done while the author worked at Virginia Tech.

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of the United States government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
ASPLOS '22, February 28 – March 4, 2022, Lausanne, Switzerland
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9205-1/22/02…$15.00
https://doi.org/10.1145/3503222.3507779

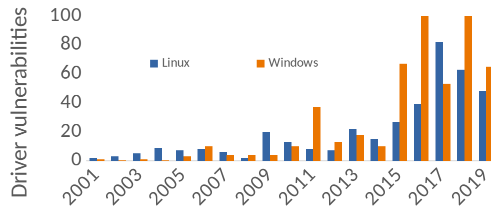


Figure 1: CVEs for device drivers [21].

can further assume that certain addresses are page-aligned, which makes even simple brute-force attacks feasible.[1]

Furthermore, OS kernel code, especially that of typical OSs in widespread production-use such as Linux, is large and complex. On the one hand, this complicates the design and implementation of defense mechanisms [30]. One the other hand, it gives great flexibility for an attacker. In fact, there is a great reward for an attacker: tampering with the kernel and gaining control of the entire system effectively enables the attacker to bypass many defense mechanisms that are deployed in user space to protect applications.

We present Adelie, an OS kernel defense mechanism that contributes to Linux security in several uniquely distinguishing ways. First, it extends kernel ASLR (KASLR) for the entire 64-bit address space efficiently by using position-independent code (PIC). Second, it implements stack re-randomization and address encryption techniques. Finally, it implements continuous address space re-randomization.

Adelie's first contribution directly enhances the security of the Linux kernel and its modules and helps to efficiently implement techniques in the other contributions. Since drivers are more likely to have vulnerabilities than core kernel code [14, 18, 43, 50] and because modules expose most ROP gadgets (see Section 6), we scope our last two contributions to kernel modules only and the most vulnerable of them – device drivers. As we further discuss in Section 6, this targeted practical approach which incurs very little overhead suffices as all these techniques combined (continuous ASLR, stack re-randomization, and encryption) prevent ROP attacks even in the presence of gadgets which originate in the core kernel or non-rerandomized modules. Thus, Adelie's second and third contributions together provide a strong defense against just-in-time (JIT) ROP attacks [62] in the entire Linux kernel.

Adelie's mechanisms are designed specifically for kernel space and solve unique challenges which are not present in prior art for user space [16, 69]. Adelie uses a zero-copying method for moving code and static data. In addition, Adelie efficiently keeps track of and unmaps previously used addresses.

We evaluate Adelie's PIC support using Sysbench, Kernbench, and microbenchmarks that characterize OS-heavy workloads. We also evaluate the cost of re-randomization for several drivers using microbenchmarks and real-life server applications (Apache and mySQL). Adelie's re-randomization overhead is small (< 2%) and entirely negligible for supporting 64-bit KASLR with PIC modules.

[1] For 4KB pages, an attacker needs $\leq 2^{31-12} = 512\text{K}$ attempts, likely $< 1\text{K}$ if the attacker knows an OS version, etc. An early boot-time (brute-force) attack, when a FS journal (ext4) is not yet flushed, will leave no traces at all.

The paper makes the following contributions:

1. Compiling and running all modules from the Linux kernel tree (e.g., over 5000 modules in Ubuntu 18.04 which we used for testing) as PIC to extend kernel's ASLR range.[2]
2. Mechanisms for stack re-randomization, address encryption, and continuous ASLR on Linux modules. Our work is the first to target large-scale OS kernels such as Linux. Due to the gargantuan engineering effort required, plus legacy and low-level code in the kernel, we ruled out re-randomization for the entire kernel. Re-randomization also incurs extra overheads. For these practical reasons, we re-randomize only on the most vulnerable components. We argue that such a targeted approach is not only justified from a performance perspective, but is also more likely to be upstreamed into mainline Linux. Vulnerable drivers are a likely starting point of an attack. As further elaborated in Section 6, our approach achieves strong protection against ROP gadget injection for the entire kernel ecosystem regardless of whether gadgets (e.g., in the core kernel which is not re-randomized) still exist. We implement a GCC plugin [63], which automatically converts existing modules to re-randomizable modules.
3. Demonstrating the mechanisms and their generality using notable drivers compiled as re-randomizable modules.

Adelie is deployed in a real-life system. When using virtualization and dividing the system into multiple guest OSs, device drivers can run either in a privileged virtual machine (VM), known as Dom0 in Xen, or in dedicated VMs, known as driver VMs in Xen. When using driver VMs, drivers are fundamentally the *only* vulnerable components in the corresponding guest OS. We used Adelie in an open-source enterprise system [47] to re-randomize a network driver in Dom0. This system comprises multiple VMs.

2 BACKGROUND

In this section, we discuss control-flow attacks, remedies against them, and the specifics of ELF binaries and PIC.

2.1 Return-Oriented Programming (ROP)

Typical attacks modify the control-flow state of the program by executing an unintended sequence of instructions. Since data-bound checks are not always enforced by a programmer, an attacker can exploit this vulnerability using buffer overflow. Such attacks generally overwrite function return addresses on the stack, thereby hijacking the program's control flow. Modern CPUs support the Write-XOR-Execute feature, known as the NX (Non-Execute) bit in x86-64 [44], to prevent a memory page from being both writable and executable. Data pages are marked as NX nowadays [12], making direct code injection impossible.

However, the NX mitigation can be bypassed via code reuse attacks such as *return-oriented programming (ROP)* [9, 13, 55, 61]. In these attacks, a stack buffer overflow is exploited to overwrite a return address on the stack with the address of a selected function (e.g., from *libc*). By carefully overwriting the stack, an attacker chains the execution of a set of functions. A variation of this technique, *stack pivoting*, manipulates the stack pointer register to point

[2] This contribution was submitted to one of the Linux kernel mailing lists as a series of patches: see https://www.openwall.com/lists/kernel-hardening/2019/03/21/6 and https://www.openwall.com/lists/kernel-hardening/2019/03/21/7.

to memory where the crafted payload resides [53]. Given a large enough set of loaded instructions, it is possible to identify a control flow instruction, such as a return or indirect jump, and create a sequence of valid instructions (*"gadget"*), which, when executed, yields a desired behavior. Jump-Oriented Programming (JOP) is a variant of ROP, where a jump instruction alters control flow [9, 13].

ROP's fundamental premise is that given a large enough set of already loaded instructions or even arbitrary executable bytes, one can piece together a sequence of instructions that is functionally valid on a given ISA and accomplishes the desired goal. This is because of the density of ISAs, in particular ISAs such as x86-64. By exploiting a buffer overflow, an attacker can therefore overwrite a stack return address with the address of the first instruction of a gadget, whose last instruction in turn overwrites the stack return address with the first instruction of another gadget, thereby executing a chain of gadgets which yields arbitrary program behavior from existing code.

ROP is Turing-complete [11] – i.e., given a sufficiently large binary, any functionality can be emulated by chaining gadgets. Several tools have been developed to automatically create ROP payloads from program binaries [57, 60] – e.g., the ROPgadget tool [57] can create an attack payload that spawns a shell that can accept arbitrary commands from an attacker.

Even though attackers rarely use ROP alone, preferring to use it only as a bridge to accessing more direct means of controlling execution, existence of such tools and ROP compilers shows that the opportunity for arbitrarily powerful unintended execution is not rare but inherent in binary code, and must be mitigated.

A related feature, SMAP (Supervisor Mode Access Prevention), available on recent x86-64 CPUs and supported by Linux, prevents the OS kernel from being tricked to use data or code from user space. Adelie assumes this feature is enabled.

2.2 Protection against Control-Flow Attacks

Code randomization is a common technique to defend against control-flow attacks. ASLR [51, 72] is a well-known technique used in modern OSs to protect user-space programs by randomizing the memory address locations at which a program executes. Similar to ASLR, various fine-grained randomization techniques [5, 36, 68] have been developed to defend against control-flow attacks. Apart from the difficulty of applying some of these techniques at the OS level [30], none of these techniques alone can effectively stop JIT ROP attacks [62].

2.3 Position-Independent Code (PIC)

For better ASLR support, user space code is typically compiled as position-independent executables (PIE) and/or shared libraries. Such code uses the "RIP-relative" addressing mode [1, 39] in x86-64, where 32-bit offsets are added to the instruction pointer, effectively allowing the program to execute anywhere in the 64-bit virtual address space.[3] Moreover, relative addresses, even if leaked, do not directly reveal the absolute addresses needed for ROP attacks. This widely advocated [56] model is steadily gaining popularity in Linux distributions [27] for user space protection.

The Linux kernel itself and its modules, however, do not employ the position-independent model. While there exists a preliminary effort [33] to compile a kernel image as a PIE, it falls short on addressing the same problem for kernel modules. Since modules still use 32-bit KASLR, [33] currently extends the kernel's KASLR range to 3GB only, which does not make a significant practical difference. Moreover, most of the code nowadays is compiled outside of the kernel image, e.g., Ubuntu 18.04's out-of-the-box kernel has over 5000 modules.

Our work extends the position-independent model for kernel modules. Thus, it complements the existing PIE patch so that the entire 64-bit range can be used for KASLR. Moreover, our design enables the kernel and the modules to lie any distance apart from each other, i.e., they do not necessarily need to be placed within a ±2GB range of each other.

[3] AMD CPUs currently restrict their virtual addresses to 48 bits. Intel has recently extended virtual addresses to 57 bits with 5-level paging. Virtual addresses can be further extended to 64 bits in the future.

2.4 Meltdown

Meltdown (CVE-2017-5754) is a CPU attack aimed at exploiting the kernel portion of page tables. Fortunately, Linux's KPTI mitigation based on the KAISER [32] page table isolation is fully transparent to the user. This mitigation does not impact any of our design choices.

2.5 Spectre-V2

Spectre-V2 (CVE-2017-5715) is an attack aimed at exploiting CPU vulnerabilities based on speculative execution and branch prediction [40]. This attack affects most existing CPUs, even outside of the x86-64 realm. The system is vulnerable due to indirect CALL (or JMP) instructions.

Linux uses a mitigation [66] which replaces indirect function calls with direct calls to special *retpoline thunks*, which, in turn, use a workaround based on a RET instruction trampoline to prevent speculative execution. The Linux kernel implements retpoline through special macros such as CALL_NOSPEC and JMP_NOSPEC for assembly code and relies on compiler support for C.

As x86-64 does not support 64-bit offsets for direct calls, indirect calls must be used for 64-bit addresses. It is crucial to reduce the number of indirect calls since they use retpolines.

2.6 GOT and PLT

ELF shared (dynamic) libraries [37] provide a special mechanism for handling external symbols. Since external symbols are unknown at compile time, compilers rely on a *global offset table* (GOT) to retain this information. Instead of specifying addresses directly, the compiled code needs to retrieve addresses from the corresponding GOT entries.

GOT is also important for other reasons. Dynamic libraries need to be shared across multiple processes that can use different virtual address ranges. Thus, PIC is typically preferred (or, in the case of x86-64, required) for shared libraries to avoid the copying overhead. Since absolute references to code and data are not identical in different processes, GOT encapsulates these addresses so that only GOT needs to be updated when sharing the code with a different process. Another reason to use GOT in x86-64 is for storing complete 64-bit addresses as most instructions support only 32-bit displacements.

We found GOT to be particularly useful for continuous re-randomization. Although the kernel uses a single address space, and shared libraries are not directly useful in the kernel, we still want to efficiently support multiple mappings to the same code due to the ongoing re-randomization. GOT allows to do so without modifying the underlying code. Although not directly provisioned by ELF shared libraries, we create multiple GOTs for different purposes within the same module to facilitate continuous re-randomization.

Dynamic libraries also use *procedure linkage tables* (PLTs) to transparently interpose on exported functions (e.g., a custom *malloc(2)* can interpose on *libc*). PLT is also used for lazy binding by dynamic linker trampolines. Although PLT is typically useless for the kernel, we use it when the retpoline mitigation is required for better code efficiency (Section 4.1).

2.7 Kernel Re-randomization Challenges

Shuffler [69], CodeArmor [16], and TASR [6] previously explored re-randomization for user space. Although Shuffler's, CodeArmor's, and TASR's goals are partially aligned with ours, the challenges are not identical. User-space techniques do not handle low-level code such as system calls, interrupt handlers, etc. Kernel code is also often written to be agnostic to threads that use it: function calls can go all the way from system calls initiated by different user processes. Moreover, Shuffler benefits from existent rich support for PIC in user space. Early attempts [30] to solve a similar problem for more componentized OS designs such as MINIX [35] pointed out many similar challenges of implementing ASLR and re-randomization in OS environments.

We also take a different approach for performing re-randomization. Unlike Shuffler and CodeArmor, Adelie avoids *binary-level* transformation. As code is available for the kernel and most of its modules, it makes sense to have a solution that requires a relatively small number of changes while benefiting from *source code* access. Shuffler also has limitations such as requiring an executable and its associated libraries to be within ±2GB from each other, effectively transforming dynamically-linked applications into monolithic, statically-linked binaries. Similar limitations exist for CodeArmor, which does not benefit from PIC. By design, CodeArmor chose to transform PIC to absolute-address code (i.e., subject to the 2GB limit unless executables are compiled with costly MCMODEL=LARGE [37]) to simplify its re-randomization process due to its extra layer of indirection.

2.8 Driver VMs

Xen *driver domains*, unprivileged guest VMs that run device drivers, are used in both desktop [58] and enterprise [2] setups.[4] Driver VMs isolate potentially vulnerable/malicious drivers (or devices) from the privileged VM (Dom0). They also offload Dom0, thereby increasing performance. Driver VMs have direct access to the underlying hardware [71] and can be used for both networking and storage by using special *paravirtualized* I/O drivers for communication.

Adelie is by no means limited to driver VMs, but they exemplify a use case where driver re-randomization complements already existent strong protections against vulnerable code.

[4] We integrated Adelie into a similar enterprise system [47] for better security.

3 DESIGN

In this section, we discuss the challenges that pertain to transforming modules to PIC as well as module re-randomization.

Our design is Linux-centric, but other OSs that rely on ELF modules (e.g., BSD) can implement similar mechanisms by making similar changes in their respective kernels. Non-ELF systems can adopt certain aspects of our approach (e.g., Windows also extensively uses the "RIP-relative" mode in x86-64) but may need some adaptations or substitutes for GOT and PLT.

3.1 Threat Model

Adelie aims to protect the Linux kernel from all known code reuse attacks that are applicable to kernel space, i.e., both traditional ROP and JIT ROP attacks. In Section 6, we discuss why Adelie provides protection for the entire kernel ecosystem even though we typically want to re-randomize vulnerable modules only, regardless of whether ROP gadgets are available elsewhere (e.g., in the core kernel). We only focus on x86-64 and assume that all existent protections such as the NX bit are enabled by the kernel. Although not required, users can leverage the stack protector mechanism used by the kernel [46] for even stronger security guarantees.

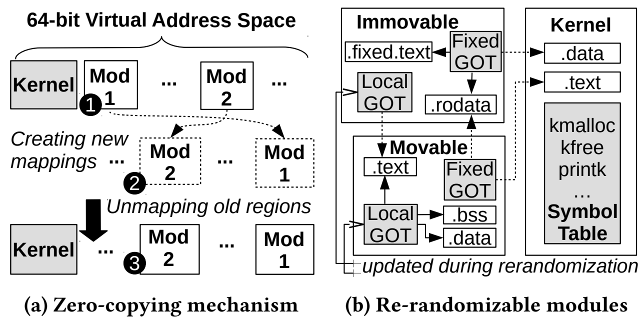
We make the following assumptions:

1. Attackers perform traditional control-flow attacks based on code reuse. NX already prevents code injection attacks.
2. Attackers primarily target kernel modules (drivers), which are more likely to have vulnerabilities in the first place.
3. We do not specifically aim to protect from non control-flow attacks [15]. However, since we re-randomize the static data layout, such attacks become more challenging.
4. Privilege escalation by data modification (e.g., *struct cred*) requires some vulnerability. Short of trivial kernel bugs, an attacker is likely to use vulnerable modules (drivers). The attacker attempts code reuse attacks by constructing a ROP chain which modifies data (e.g., by leveraging *ioctl(2)* from user space).
5. Hijacking of kernel page tables is infeasible since the core kernel itself is unlikely to be vulnerable.

3.2 Goals for Continuous ASLR

We define the following main goals for Adelie:

Generality We aim to transform *all* modules to the 64-bit KASLR model. For continuous KASLR and stack re-randomization, we aim to make the process as automatic as possible.
Performance For re-randomization, the number of absolute addresses must be minimized. Since address layout changes frequently, we aim to avoid the cost of copying code and data.
Entropy Every module should be any distance apart from other modules and the kernel. Thus, leakage in one kernel module or the kernel does not automatically reveal the larger picture of the address space layout.
Security We aim to protect against code reuse attacks by minimizing the time duration during which module addresses remain valid. We also encrypt return addresses with a key that is continuously re-randomized.



(a) Zero-copying mechanism **(b) Re-randomizable modules**

Figure 2: Adelie's design.

3.3 Extending KASLR

Our first step is to extend Linux's KASLR support. Since position-independent code can reside anywhere, PIC extends KASLR to 64 bits while avoiding costly (absolute-address) models such as MCMODEL=LARGE [37]. By fully converting all modules to the PIC model, we can also attain performance benefits outlined in our goals by eliminating code modification during re-randomization.

As discussed in Section 2.3, there exists preliminary support for compiling the Linux kernel as a position-independent executable (PIE). It can be considered analogous to running position-independent executables in user space.

PIE, for the most part, only changes the absolute address mode to the "RIP-relative" mode. All global variables are still assumed to be within ±2GB reach.[5] Thus, a kernel PIE avoids the global offset table (GOT). The procedure linkage table (PLT) is not used by kernel PIEs since the kernel does not directly call any outside function (as opposed to user space PIEs which call functions from shared libraries).

Kernel modules, however, cannot use PIE because they can be placed any distance apart from each other and the kernel. Instead, Adelie uses a more general PIC model from shared libraries with GOT and PLT support. However, as discussed further in Section 4.1, we do not convert modules to shared libraries but use position-independent relocatable objects.

3.4 Continuous Module Re-randomization

The most vulnerable kernel code (i.e., device drivers, certain kernel libraries, etc.) can be compiled as modules and subsequently re-randomized. The core kernel and non-vulnerable modules do not have to be re-randomized because stack address encryption, as described below, defends against *any* ROP gadgets. Gadgets are also likely to come from code which is more predictable to an attacker, such as device drivers, for which we use continuous ASLR. We now describe how we achieve the goal of efficient run-time re-randomization of kernel modules.

Zero-Copying Mechanism. One critical aspect of our work is that, unlike Shuffler [69], we completely avoid copying code and static data while re-randomizing addresses. Similarly to CodeArmor [16], we remap existing pages to new addresses but favor a more fine-grained memory reclamation technique to quiescent state-based reclamation (QSBR) [34]. In Figure 2a, we demonstrate the high-level principle. Initially, at instant 1, both the kernel and the modules are in some randomly chosen address space, any distance apart from each other. This is achieved using our extended 64-bit KASLR. Periodically, module locations are re-randomized by a special *randomizer* kernel thread. This thread creates new mappings, as shown at instant 2. Finally, when old regions are no longer used, we unmap them. In this process, no copying is actually made; we simply create new page table entries that point to the same physical memory locations.

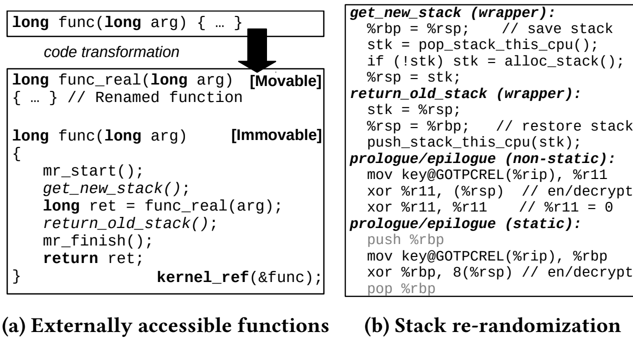
Module Organization. Re-randomizable modules consist of two logical parts: the movable (most of the code and data) and immovable, which mostly implements glue code for the kernel. While for convenience the immovable part is placed in the module, it can be viewed more as an integral part of the kernel. In Figure 2b, we show a typical module layout. Because the immovable part can be any distance away from the movable part, we maintain two different sets of GOTs. Each set of GOTs stays within ±2GB of the corresponding logical part. Each part can reference global (kernel or fixed) symbols or module-local symbols. Only module-local symbols need to be updated when modules move. For this reason, each set contains two GOTs for fixed and local addresses, respectively. We only update (i.e., reallocate) local GOTs when moving modules. Moreover, we minimize the number of entries in local GOTs (see Section 4.1). In the figure, local GOT entries always point to the movable part. Fixed GOT entries are either referring to the kernel or to the immovable part of the module.

Controlling Address Space Lifetime. It is crucial to unmap pages from previously used virtual addresses in a timely manner, so that ROP gadget addresses quickly become obsolete and useless for an attacker. However, kernel code is quite complex in the way functions from modules are called. A long chain of functions can be called, eventually leading to some user space thread making a system call. It is infeasible to use existing techniques of stack unwinding (e.g., as in Shuffler [69]) while pausing the execution of the entire kernel.

Instead, we opted to use delayed unmapping. We let pending calls finish using previously obtained virtual addresses. Any call that follows re-randomization must use new virtual addresses. As soon as the last pending call completes, the previous address range is immediately unmapped. Since almost all kernel space calls should be relatively quick (or, otherwise it would indicate bugs in the code), unmapping is not delayed substantially enough for an attacker to benefit from it.

The challenge is to track pending calls with little overhead and in a scalable manner. We use the Hyaline reclamation scheme [48, 49], which is similar to epoch-based reclamation (EBR) [28, 34]. Hyaline's performance is very similar to that of EBR, but Hyaline enables much easier integration into the Linux kernel since it is context-agnostic and does not make any assumptions about how threads are managed. Both Hyaline and EBR largely solve the same problem of efficient, optimistic memory access to blocks that are concurrently being deallocated by other threads. We enclose operations that access potentially disappearing memory blocks

[5]PIE optimizes external symbols because they can be allocated within ±2GB from the executable image even if imported from shared libraries.



(a) Externally accessible functions **(b) Stack re-randomization**

Figure 3: Code transformations.

with `mr_start` and `mr_finish`. These operations postpone memory reclamation until after all pending calls (i.e., those that called `mr_start`) execute `mr_finish`. Memory blocks are not deallocated directly, instead they are first *retired* with a special `mr_retire` operation. The deallocation takes place only after pending calls complete.

As in CodeArmor, unmapping can be delayed, but that does not appear to be a practical issue for modules we tested. Linux also has built-in mechanisms to detect calls that block for too long. Blocking is more likely to manifest when using softirqs/workqueues. However, softirqs/workqueues do not require `mr_finish` to wait until the request is completed, and the re-randomization routine will only need to modify the function handler address. Only inside the actual handler (when scheduled), do we need to call `mr_start`/`mr_finish` again. Blocking, in general, can be more challenging but typically can be handled easily, i.e., by inserting `mr_start`/`mr_finish` and controlling the state around these calls manually.

Function Wrapping. Externally accessible functions from re-randomizable modules are specially wrapped for two reasons: (1) a module moves in the address space while the kernel retains absolute addresses to the module code; by placing a wrapper function into the immovable part of the module, the kernel can reference that wrapper function instead; since we create the local GOT in the immovable part as well, there is an easy way to update references to the original function which is placed into the movable part. (2) lifetime control: almost all function calls initiated from the outside must be protected by the special `mr_start` and `mr_finish` calls from the memory reclamation algorithm.

In Figure 3a, we show how an externally accessible function is transformed by renaming it and placing a wrapper with the original name into the immovable part. The kernel references the wrapper rather than the original function. A special compiler plugin (Section 4) automatically wraps functions.

Stacks. We continuously re-randomize stacks, so that an attacker cannot hijack the control-flow pointers placed on the stack. Changing stacks is not a trivial task as the kernel maintains multiple stacks for user- and kernel space. Each thread needs its own stack, and stack re-randomization must be fast.

We maintain a per-CPU lock-free (LIFO) list of stacks. We substitute stacks at the beginning of the function wrapper by dequeuing the head of the per-CPU list. New stacks are allocated on demand as needed. In Figure 3b, we show `get_new_stack` and `return_old_stack` code used in wrapper functions. Old `%rsp` is saved into `%rbp`, and `%rsp` is replaced with the newly dequeued stack. When exiting, the stack is restored and returned to the list. Since each CPU has its own list, the contention is low and can only occur due to the re-randomizer thread deleting old stacks. The re-randomizer thread generates new LIFO lists for each CPU. Old list heads are atomically replaced with new heads. The old lists are garbage collected and freed (when it is safe).

Return addresses can potentially be hijacked by jumping to non-rerandomizable code. Since they are not strictly controlled as other non-rerandomizable pointers (Section 6), it is crucial to encrypt them. For that reason, the *prologue* and *epilogue* of each function in re-randomizable modules are identically modified to encrypt and decrypt (XOR with a key) the return address as shown in Figure 3b. Static functions occasionally use custom calling conventions, and we cannot use `%r11` as a scratch register. Since frame pointers are also set in the prologue and epilogue, we recycle `%rbp`; we also optimize out `push` or `pop`, respectively. In case of `%r11`, we clear the register to avoid accidental key leakage. The encryption key is randomly generated and stored in the local GOT, which gets reallocated during re-randomization. The key changes every re-randomization. Even if an attacker manages somehow to read the encryption key from the GOT, it becomes obsolete every time the module is re-randomized. Moreover, the absolute address of the local GOT also changes due to re-randomization.

Since up to six arguments are passed through registers in x86-64 (we have not discovered any function to wrap with > 6 arguments), we simply replace stack pointers for a duration of the call. If any arguments have pointers to stack data, they will simply reference the original stack. However, all functions that are executed from the module will use the new stack.

4 IMPLEMENTATION

The overall implementation effort is reasonable. The changes for PIC modules (≈ 727 LoC) assume that the Linux kernel is already patched by the existent PIE patch [33], which is related to our change but only extends KASLR for the kernel itself. With our change, *all* (≈ 5000) modules use 64-bit (basic) KASLR. Our system is stable and self-hostable on different machines (with Ubuntu 18.04's default configuration). We confirmed that our system can still work in specialized scenarios, e.g., when running Linux as Dom0 in Xen.

With respect to PIC adaptation, obstacles were primarily in assembly files, non-standard calling conventions, C macros with inline assembly, hypervisor interactions, and special calls (e.g., exception handlers).

Re-randomization of modules is implemented using a common part for all modules (≈ 2815 LoC). Using our GCC plugin (≈ 1400 LoC), we automatically modified and tested network (E1000E, E1000, ENA), storage (NVMe), USB 3.0 (xHCI), and file system (FUSE, ext4) drivers. Although it is infeasible to rigorously test every single device driver, our approach is more or less general; we also confirmed that the plugin (at least) successfully compiles *all* kernel modules.

Our GCC plugin is *only* needed for continuous ASLR. It detects and wraps all functions and variables that are exposed to the kernel by using predefined macros and modifying the compiler output

```
call/jmp *foo@GOTPCREL(%rip) → call/jmp foo; nop        [local calls]
                                                         No PLT

call/jmp foo@PLT → call/jmp foo              With       [local calls]
                                             PLT
foo@PLT: mov foo@GOTPCREL(%rip), %rax                   [PLT stubs for
         JMP_NOSPEC %rax                                 global calls]

mov foo@GOTPCREL(%rip), %R → lea foo(%rip), %R          [local symbols]
```

Figure 4: Summary of run-time patching.

accordingly. Similarly, for all functions inside modules, the plugin automatically applies *prologue* and *epilogue* changes as discussed above.

Our plugin makes its best efforts with respect to the C-language semantics. Although exceptions from typical rules are certainly possible, they should be rare in practice since programmers generally follow good-style Linux guidelines in the code. Also, if problems with symbols arise, it should be very easy to detect external (non-rerandomizable, kernel) addresses since they are marked as U (undefined) in the corresponding module symbol.

4.1 Executable Format

We kept the existing relocatable format and adapted it for PIC. This enables larger flexibility when handling GOT and PLT, as relocations are only finalized at run-time. Whereas shared libraries have just one GOT, we allocate four GOTs: two tables for the movable and two tables for the immovable module parts. One table from each pair is used for the imported kernel (or the immovable part) addresses, and the other one is used for module-local addresses which are periodically changed.

Since the location of symbols is often unknown, the compiler generates code which can be suboptimal for local symbols and calls. To optimize code, we patch it at run-time (see Figure 4).

If the retpoline mitigation [66] is disabled (for newer CPUs), we avoid PLT stubs by inlining them when compiling modules. When loading modules, we patch relocations to optimize instructions that use module-local symbols because they are known to be within the ±2GB range from the instruction pointer. Since a direct CALL/JMP is one byte shorter than its indirect counterpart, we pad it with NOP.

When retpoline is enabled, the above-described approach creates obstacles for local call optimizations as corresponding retpoline-based calls use longer sequences of instructions, clobber registers, etc. To support this case efficiently, we create PLT stubs using Linux's JMP_NOSPEC.[6] Local calls are optimized by just eliminating respective PLT stubs.

Linux keeps 32-bit relative addresses for handler functions when generating exception tables. Since the number of global handlers is small, we avoided intrusive changes by using PLT stubs for exception handlers even in the non-retpoline case. (While implementing this change, we discovered a subtle bug in relocation handling by the GNU assembler, which was subsequently resolved.)

Variables are also optimized. By default, the compiler generates code that uses GOT to retrieve variable addresses; we patch relocations to use the LEA instruction for local symbols.

[6] Linux occasionally uses non-standard calling conventions. `%rax` seems to be the only safe volatile register (also used for return values) across almost all conventions. We handle a few exceptions separately.

Aside from direct performance benefits, the aforementioned optimizations substantially reduce the total number of GOT and PLT entries, thereby reducing the risk of leaking absolute addresses to an attacker. Moreover, when performing continuous re-randomization, we have to change addresses in GOT entries for local symbols. Thus, we only need to change a few, thereby reducing the added cost of re-randomization.

We write-protect pages with GOT/PLT entries after initialization so that they cannot be overwritten by an attacker.

4.2 Module Re-randomization

A special process takes place when loading re-randomizable modules. First, we identify sections that belong to movable and immovable parts and allocate them separately. We only place ".fixed.text" (the section that contains function wrappers) and ".rodata" in the immovable part. The reason why we currently place read-only module data in the immovable part is mostly to avoid excessive changes in the module because certain constants and string literals can be directly passed to the kernel, and we do not want to re-randomize their location.

For re-randomizable modules, we use four different GOTs. Based on whether the code is an immovable or movable part and symbol locality, we choose one of these GOTs when generating a GOT entry. Occasionally, two tables will contain duplicates of the same symbol (e.g., a local GOT entry from the immovable part points to the same address as a GOT entry from the movable part). We use separate GOTs for movable and immovable parts because these parts can be any distance away from each other, while GOTs must always be placed within ±2GB reach from %rip due to the "RIP-relative" address mode.

During module re-randomization, a special kernel thread ("*re-randomizer*") periodically performs the following steps. First, a new virtual address space map is created; it maps to the same physical addresses as an old map. New local GOTs are allocated for both movable and immovable parts of the module. All entries from the previous GOTs are adjusted to point to the new memory address space when creating the new GOTs. Corresponding GOT pages in the new address space are remapped to point to the new GOTs. Subsequently, the re-randomizer thread calls a special function from the module to update its run-time function pointers (needed for some modules). Finally, the re-randomizer thread calls `mr_retire` from the memory reclamation algorithm to request unmapping. After that, the re-randomizer thread sleeps for the specified re-randomization period, and then repeats the entire process. The memory reclamation algorithm unmaps the previous address space when all pending calls complete.

4.3 Limitations

Our re-randomization approach is relatively coarse-grained. Although this may certainly have drawbacks, Adelie's current implementation makes the very first step towards continuous module re-randomization. If more fine-grained re-randomization is desired in the future, it can be attained at least at a function-granularity: we would need to create separate GOT tables per each function or a group of functions. This approach is especially useful for more fine-grained re-randomization of larger modules. Although each

Table 1: Server and client systems.

	Server (for Evaluation)	Load Generator
CPUs	Xeon Silver 4114 2.20GHz	Core i7 4770 3.40GHz
Cores	2x10, no HyperThreading	1x4, no HyperThreading
L1/L2 cache	64 / 1024 KB per core	64 / 256 KB per core
L3 cache	14080 KB	8192 KB
Memory	96 GB	16 GB
Network	Intel E1000E 1GbE	Intel E1000E 1GbE
Storage	Samsung 970 EVO NVMe	Samsung 860 EVO SSD
USB 3.0	Intel C620 xHCI	N/A

module can still be self-sufficient in terms of ROP gadgets, frequent address changes prevent the attacker from performing a successful attack.

Frequent address space remapping may contribute to re-randomization costs. More specifically, TLB needs to be flushed more frequently after page table updates. We did not find that to be a significant problem in Section 5 for re-randomization periods that we used. The TLB cost is also somewhat unavoidable for continuous re-randomization even for other alternatives (e.g., simple copying), where the page table would still have to be updated frequently due to the 64-bit address space being very sparse.

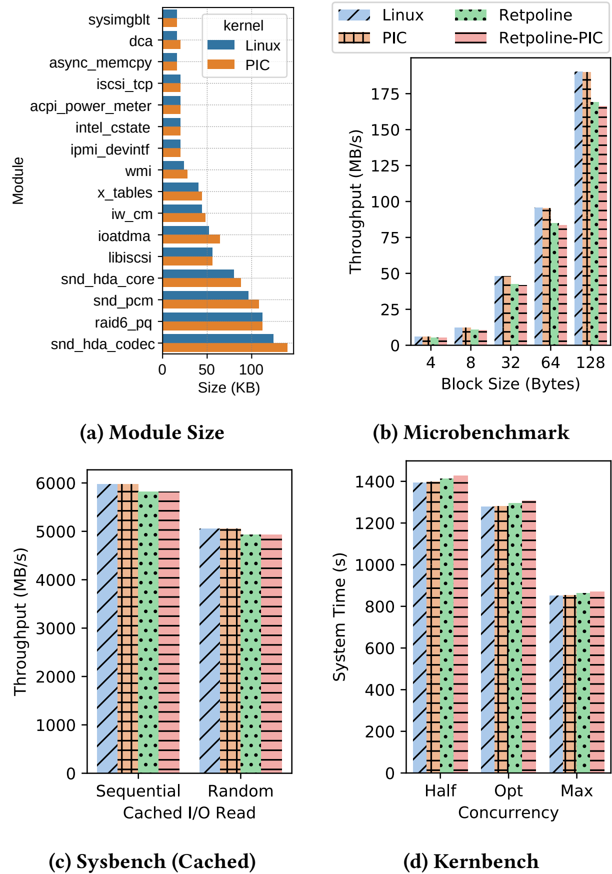
5 EXPERIMENTAL EVALUATION

Our primary goal is to determine Adelie's performance overheads (if any). We use various microbenchmarks to evaluate overall system performance when running position-independent modules. We also use macrobenchmarks including ApacheBench [3] and SysBench/OLTP (mySQL) [65] to evaluate the performance of real-life server applications, specifically when using continuous re-randomization. Finally, we considered the SPEC CPU benchmarks, but since they are CPU intensive, we did not observe significant differences, and consequently we do not present SPEC CPU results.

We evaluate each data point three times and present the average. Unless specified otherwise, the standard deviation is < 0.5%. As there is no other kernel re-randomization system for Linux, we compare directly against original Linux.

In all of our tests, we use Ubuntu 18.04 with standard packages but a different Linux kernel version (v5.0.4). In all test combinations, we use the kernel with the default Ubuntu configuration, which compiles over 5000 various kernel modules. *All* modules use the PIC model, presented in the paper. We tested a fair amount of modules on various hardware to confirm that PIC is bug-free across different modules.

Table 1 shows our experimental setup. For continuous re-randomization, we evaluate widely used device drivers: Intel E1000E (network) and NVMe (storage). Additionally, we tested re-randomizable xHCI, FUSE (file systems in user space), and ext4 modules as an extra load. Finally, we successfully ran other network drivers including E1000 (used in VirtualBox) and ENA (used in Amazon AWS clouds). This choice of re-randomizable modules is based on a reasonable assumption that device drivers are the most vulnerable components; we choose drivers from the most critical classes of drivers: network, storage, USB, and file systems.



(a) Module Size **(b) Microbenchmark**

(c) Sysbench (Cached) **(d) Kernbench**

Figure 5: PIC vs. non-PIC modules.

5.1 Position-Independent Modules

We first measured PIC modules (our first contribution), their overall impact on system performance, and memory footprint.

In Figure 5a, we randomly selected modules of different sizes to demonstrate the memory footprint related to the conversion to the PIC model. The difference in memory footprint of the retpoline and non-retpoline modules is insignificant here; thus, we only present retpoline-enabled (PIC) and vanilla Linux modules. Overall, the overhead is negligible for all modules.

Next, we ran micro- and macro- benchmarks to evaluate the performance impact of PIC modules. We use four setups: vanilla Linux without retpoline, vanilla Linux with retpoline, PIC modules without retpoline, and PIC modules with retpoline. We used the default Ubuntu configuration in all tests.

In Figure 5b, we ran our own microbenchmark which uses dd to read files with varying block sizes. This test is CPU bound due to the use of the buffer cache. This experiment revealed the real impact of retpoline (a Spectre-V2 mitigation). Figure 5b shows that without retpoline the performance of PIC and non-PIC is nearly identical. A slight performance hit of the PIC code (with enabled retpoline) is due to retpoline-safe indirect jumps to external functions in PLT stubs.

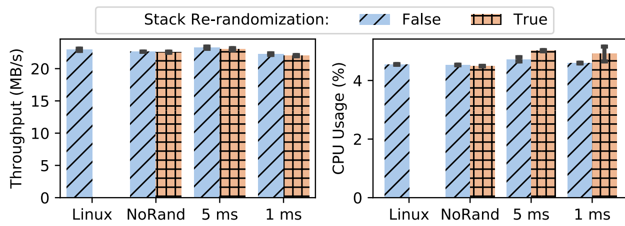


Figure 6: NVMe read throughput.

We used the sysbench file_io benchmark to measure the throughput on random and sequential reads. For this experiment, the files were cached in RAM to keep the results I/O invariant. The results in Figure 5c show that the performance of PIC-enabled and non-PIC systems is nearly identical.

Kernbench is a CPU throughput benchmark that is often used to compare kernels. We recorded the time spent in kernel space at three levels of concurrency. The results in Figure 5d show no substantial difference across different configurations.

Overall, PIC's cost is negligible even for enabled retpoline.

5.2 Evaluation of Re-randomization

To evaluate module re-randomization, we use multiple benchmarks. We first run experiments using typical I/O loads with ordinary device drivers. To estimate worst-case overheads, we also run a separate CPU bound test in Section 5.3 by designing a special driver which handles dummy IOCTL requests in a loop.

We use the retpoline-enabled kernel, as we previously already identified the cost of the retpoline mitigation. For all tests, we present CPU usage across all 20 cores. We evaluate up to *five* different device drivers. We focus on the most critical classes of drivers, which are likely to be very attractive to an attacker.

To reliably evaluate the NVMe driver under re-randomization, we designed an experiment that minimizes the effects of I/O in the underlying hardware. We created our own benchmark that measures read throughput of a file stored on the NVMe storage. The file is opened with O_DIRECT and O_SYNC flags using the open syscall, and a block size of 512 bytes is repeatedly read from the beginning of the file in a tight loop. The above-mentioned flags guarantee synchronous data transfer through the NVMe driver and prevent buffer caching in the kernel. We read the same block over and over again to leverage NVMe's internal DRAM cache in an effort to minimize I/O wait time. We measured the read throughput of the NVMe driver and recorded the CPU usage for the duration of the experiment. We set different re-randomization intervals (1 and 5 ms) and compare against vanilla Linux. Past works, e.g., Shuffler [69], argued that even 50 ms is more than sufficient to prevent typical attacks, but we use much shorter intervals to reflect Adelie's performance in the event that intervals must be shortened in the future as the sophistication of security attacks grows. We also measure results when performing no re-randomization. Apart from a slight increase in CPU usage (Figure 6), which is within the margin of error, the performance of NVMe storage remains largely unaffected by re-randomization. Enabling or disabling stack re-randomization does not create much impact.

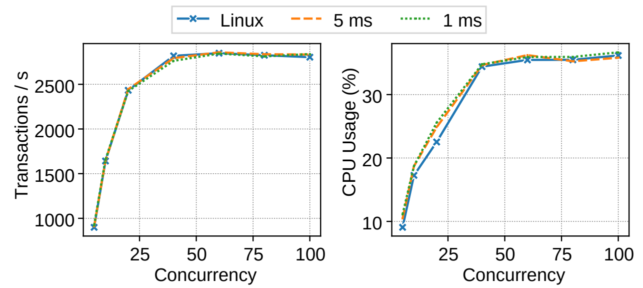


Figure 7: mySQL with re-randomizable modules.

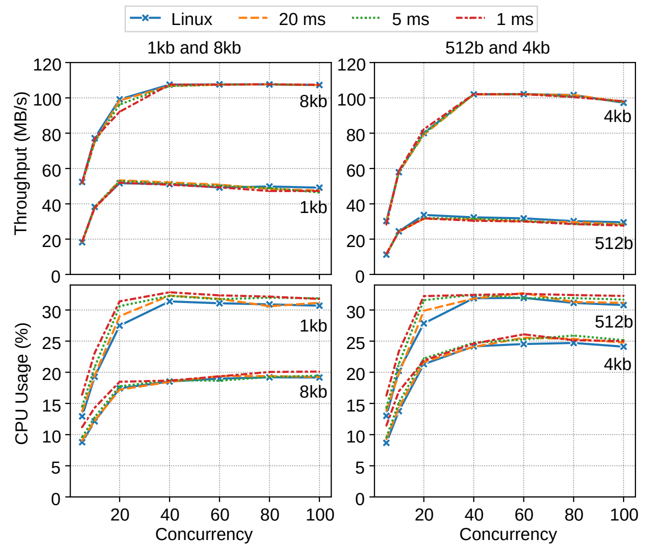


Figure 8: ApacheBench with re-randomizable modules.

We then measure network and storage intensive applications while re-randomizing corresponding drivers. We use mySQL and Apache with their default configurations from Ubuntu. We run the corresponding macrobenchmarks from a client machine which is directly connected to the network adapter of our testbed (server). We only present results with enabled stack re-randomization as no substantial difference is observed otherwise. We omit Linux with no re-randomization since its results are almost identical to that of vanilla Linux.

We measure mySQL performance using sysbench oltp on a database comprising 10 tables with 1,000,000 rows of data each. The database is partially cached in memory and the experiment is conducted with varying levels of concurrency and different re-randomization periods. We re-randomize both the E1000E and NVMe drivers. (Re-randomizing any of them alone yields very similar results.) Figure 7 shows that mySQL's rate of transactions is identical for vanilla Linux, 1 ms, and 5 ms. The CPU usage increases slightly ($< 2\%$) prior to the concurrency saturation point. The network throughput (going up to 110MB/s) is identical across all configurations.

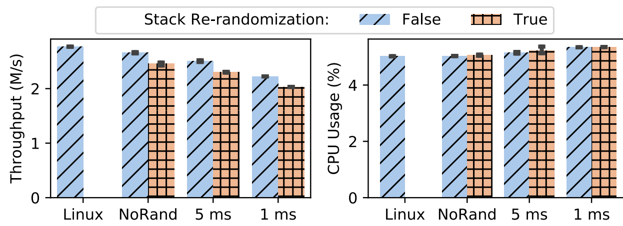


Figure 9: IOCTL throughput.

In Figure 8, we present results for Apache using different block sizes (512B-8KB). We re-randomize several modules: E1000E, NVMe, FUSE, ext4, and xHCI drivers. In this test, the pressure is applied mostly to E1000E with occasional NVMe accesses. Other drivers are not on the critical path; they just provide an extra re-randomization load. Re-randomizing any driver alone yields very similar results. Smaller blocks put more stress on the system as the total number of system calls increases. As Figure 8 shows, re-randomization does not impact the overall system throughput. Re-randomization increases CPU usage for smaller blocks ($\approx$ 2%, except smaller concurrency). We found that the CPU usage can be reduced by using 20 ms periods, which still provide sufficient security guarantees.

5.3 CPU-Bound Test

Most of our experiments on device drivers showed that re-randomization does not impact the device performance. This can be attributed to the fact that device drivers are I/O bound. The I/O wait time outweighs the CPU time by a large margin.

To test the extreme case of a CPU bounded driver, we designed a special microbenchmark. We created a dummy device driver that implements a null `ioctl` operation. We repeatedly make the `ioctl` syscall on the driver in a tight loop and measure the number of ioctl operations performed per second. Figure 9 shows the throughput (in million operations per second) along with the corresponding CPU usage. This benchmark captures the impact of function wrappers and stack randomization. We found that function wrappers cause a performance drop of $\approx$ 4% and stack randomization causes an additional drop of $\approx$ 6% when compared to the original Linux.

5.4 Scalability

From the macrobenchmarks, we found that driver re-randomization does not increase CPU usage significantly, which indicates great scalability. Furthermore, adding an extra driver has very little performance impact, as we observed in OLTP and ApacheBench benchmarks, which simultaneously re-randomize several potentially vulnerable modules in the presented results. Nonetheless, we wanted to get worst-case scenario estimates. The CPU usage of the re-randomizer thread is 0.4% for a period of 20 ms (shared across all modules). A typical server has an average utilization $\approx$ 20-30% [4, 10]. With the default Ubuntu configuration, a typical system uses around 100 modules. Estimating very roughly, even if we make *all* these modules re-randomizable and assume that CPU usage will increase by 0.36% per every group of five additional modules as in ApacheBench (a very pessimistic, worst-case estimate), our approach can comfortably re-randomize over 950 modules.

6 SECURITY ANALYSIS

Traditional ROP. Since attackers inject absolute addresses for ROP gadgets, and the kernel uses one half of the 64-bit virtual address space (57-bit for present CPUs), the probability of guessing a correct address is 2^{-56}, which is practically impossible. Moreover, an attacker can often assume that certain code is aligned at 2^{12} = 4KB page boundaries. Thus, the effective probability is $2^{-(56-12)} = 2^{-44}$, which is still unrealistic. In comparison, Shuffler's [69] and CodeArmor's [16] probability is only $2^{-(31-12)} = 2^{-19}$ (they use 32-bit offsets).

Blind ROP [8] uses *fork(2)* and does not apply to the kernel.

JIT ROP. The first step for the attack is to find some vulnerability such that ROP gadgets can be placed. We observe that: (1) vulnerabilities through buffer overflows are likely to be discovered in drivers, which we re-randomize; (2) stack re-randomization and address encryption (Section 3.4) already prevent ROP gadget injections through the return address from non-rerandomized (e.g., kernel) code; (3) the corresponding module is re-randomized, i.e., code locations keep moving; (4) the latter implies that pointers are also adjusted when re-randomizing by adding an offset.[7] If a hijacker inserts an absolute address to a non-rerandomized (e.g., kernel) code, this address is also adjusted and consequently becomes invalid due to completely random module movements. The only recourse for an attacker is to use non-rerandomizable (e.g., kernel) pointers inside the module. However, these pointers are strictly controlled by thin/secure wrappers (immovable part) and fixed GOTs (movable part). Fixed GOTs are write-protected, making hijacking impossible. Fixed GOTs are always accessed in RIP-relative mode, and the (movable-part) fixed GOT moves with the module (i.e., cannot be faked). Moreover, JIT ROP attacks must complete between re-randomization intervals of modules, and an interval of one kernel module does not coincide with that of another one, further reducing the probability of inserting (and even discovering) a working chain of ROP gadgets.

The entire attack must be performed within several milliseconds; all known attacks need several seconds to complete [69].

Address Hijacking. Aside from return-address hijacking, our approach defends against any *static data* or *stack* related attacks. In Linux, static data function pointers are very common; e.g., consider this definition in `fs/ext4/file.c`:

```
struct inode_operations ext4_file_inode_ops = {
    .setattr = ext4_setattr, ...        };
```

Our idea stems from the fact that pointers that can be hijacked in a (re-randomizable) module can only be of two types: (1) pointers to addresses within the module itself (the vast majority of the pointers in almost all modules). This is not a concern; hijacking these pointers is futile since these addresses are going be changed randomly when the module is moved (re-randomized). (2) pointers to non-rerandomizable kernel code, e.g., network API calls, VFS calls, printk, etc. The latter are guaranteed to be safely accessed through (read-only) GOT inside the movable part since there is no other way to access arbitrary 64-bit addresses.

[7] Those will typically be *stack* and *static data* pointers, e.g., the static ext4_file_inode_operations structure (below) references many functions. *Heap* pointers are less common and are modified during re-randomization since they are typically module-local. See more details in the next paragraph.

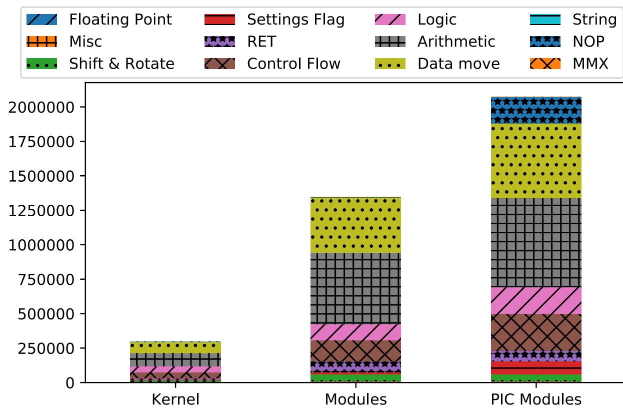


Figure 10: ROP gadget distribution (the number of gadgets).

The immovable part may also contain static data that exports *wrapped* module functions. The wrappers will access module-local functions through local GOT inside the immovable part, which is read-only. Typically, suitable static data itself is also read-only as in the example above (i.e., uses the *const* qualifier). Even if the data is writable, an attacker cannot easily modify it since the immovable part (as any other fixed kernel location) is arbitrarily far away from the *vulnerable* movable part.

A notable exception would be return addresses, which are writable, not adjusted during re-randomization (unlike other module-local addresses), and hard to manage in a controlled manner. For this reason, we encrypt return addresses, making non-rerandomizable address injections via them impossible.

Finally, even though *heap-based* hijacking seems to be less of a concern for the kernel in general (unlike typical user-space C++ code), it is still typically prevented by our approach. If a module allocates pointers in the heap (e.g., some data structure that is passed to the kernel), those addresses will be modified during re-randomization since they are typically module-local. Commonly, modules *export* their addresses to the kernel and *import* only a few addresses (e.g., kernel API functions).

Delayed Unmapping. A hypothetical problem arises when unmapping is delayed indefinitely. A similar concern exists in prior solutions, e.g., CodeArmor's QSBR, since unmapping cannot be triggered until pending calls are active. We have not found that to be a practical issue for modules we tested, since all in-kernel calls are very quick and do not have indefinite blocking. Linux also has built-in mechanisms to detect calls that block for too long, which safeguards from any potential issues. In Section 3.4, we discuss how we address specific corner cases such as long-term blocking and work queues so that unmapping is never delayed indefinitely.

ROP Gadget Distribution. We quantified ROP gadgets using the Ropper tool [59]. Our protection guarantees are not based on mere ROP gadgets availability, but their reduction certainly makes the life of an attacker even harder. Figure 10 shows distribution for the kernel, original modules, and our PIC-enabled modules in Ubuntu 18.04. The ROP gadgets are classified according to the type of their instructions. The immovable part of PIC modules has a negligible amount of gadgets as almost all code stays in the movable part. Most gadgets reside in modules, and only a fraction (15%) is in the core kernel. The number of gadgets does increase for PIC vs. non-PIC modules, but PIC also enables 64-bit KASLR, efficient continuous re-randomization, and reduces the possibility of absolute address disclosures, which is a good trade-off.

Table 2: ROP gadget categories.

	Non-PIC	PIC
With ROP Chain, no side-effect	4,320	4,358
With ROP Chain, with side-effect	1	1
Without ROP Chain	1,008	970
Number of Modules	5,329	5,329

To evaluate the quality of ROP gadgets, we constructed a specific example with NX. Table 2 shows that 80% of the modules contain enough gadgets for a chain to disable NX.

7 RELATED WORK

Modern attacks use ROP, even when using SGX [41]. ASLR has been widely researched from early user-space implementations [5, 51] to OS-specific approaches such as fine-grained ASR in MINIX [30]. Similarly, (32-bit) KASLR was implemented in Linux [45]. Adelie enables full 64-bit KASLR in Linux by transforming all modules to use PIC.

Continuous re-randomization was previously proposed and used in various contexts. Stabilizer [20] uses re-randomization to enhance performance evaluation in user space. Stabilizer, however, focuses on performance evaluation and lacks Adelie's security features, e.g., address encryption. Fine-grained ASR [30] is an early implementation of re-randomization for OSs that rely on component-based designs such as MINIX. Shuffler [69] is a user-space technique for continuous re-randomization to protect against blind and just-in-time ROP attacks, a similar problem that we address for kernel code. CodeArmor [16] is another user-space technique which uses page remapping and QSBR for better efficiency. Shuffler's and CodeArmor's goals are different from ours: they rely on binary transformation of user-space programs, whereas our technique benefits from Linux code availability as well as from the position-independent model used by our modules.

There also exist (user-space) techniques that rely on compiler support. Remix [17] extends the LLVM compiler to add extra nop-paddings, allowing runtime flexibility for moving code inside functions. This way ROP gadget locations change. However, attackers can still use function pointers to defeat this re-randomization scheme. TASR [6] is another system based on the assumption that re-randomization in the program should happen before input and output (between corresponding system calls). TASR is only suitable for user-space programs. Although authors report that TASR overheads are very low, [69] argues there are additional 30-40% overheads due to the use of the -Og compiler optimization flag (vs. -O2 which is normally used). The -Og flag is intrinsic to TASR. TASR also has limitations unacceptable for typical kernel-mode code, e.g., disallowing upcasting into function pointers, use of custom memory allocators, sizeof() assumptions, etc.

It is also possible to do arbitrary changes to existing binaries by a special recompiler. Egalito [70] modifies existing binaries such that they can change process layout entirely. More specifically, Egalito is demonstrated to work together with JIT-Shuffling, a continuous re-randomization technique which is similar to TASR and Shuffler.

Although Adelie does not need to restart kernel modules while re-randomizing address space layout, a number of techniques to do so were proposed in the past. One approach [64], which is specific to device drivers, enables user programs to still work correctly even when device drivers are restarted. This approach extends the kernel to use special shadow drivers which replace failed drivers. This approach cannot be applied directly to enhance security, as an attacker can still figure out ROP gadget locations for shadow drivers and insert them.

Other techniques aim to enhance OS security by other means. NICKLE [54] uses virtual machines to run kernel code in shadow regions. The kernel code can be transparently executed in a virtual machine in runtime. Similar techniques are also used by HookSafe [67] and hvmHarvard [31]. Other techniques such as [42] rely on special compiler support to protect kernel control data. Unfortunately, none of these techniques provide an exhaustive solution to the security problems that modern OSs have to deal with.

Adelie also differs from existing approaches in its focus on large-scale, monolithic OS kernels such as Linux, which use a low-level programming model and have many inter-connected components that lack strict boundaries and isolation from each other. Adelie also provides a comprehensive solution that can be adopted to a wide range of modules.

Control Flow Integrity (CFI) can prevent code reuse attacks by ensuring that the control flow of a program remains valid. In CFI, any indirect branch taken by an application must be in accordance with its control flow graph. CFI mechanisms were also applied to kernels (KCoFI) [19]. KCoFI is compiler-based and can complement Adelie. Unfortunately, CFI can still be defeated by a careful selection of ROP gadgets [25, 26].

Specialized hardware techniques, such as Morpheus [29], can mitigate control-flow attacks. They can also complement Adelie's KASLR defense mechanisms for stronger security.

Finally, some papers [52] focus on the NX bit enforcement and code diversification. This is still very desirable for Adelie, irrespective of continuous re-randomization, and can complement Adelie's defense mechanisms.

8 CONCLUSIONS

We presented Adelie, which contributes to KASLR in several ways. First, we extend KASLR to 64 bits using PIC, which substantially increases KASLR's entropy and makes traditional ROP attacks impractical for kernel space. It is intended for the entire kernel ecosystem. We successfully tested Ubuntu 18.04's default configuration (with $\approx$ 5000 modules) across different machines in a self-hosting mode.

The paper's other contributions are continuous re-randomization of kernel modules, return address encryption, and stack re-randomization. This is the first effort to use these techniques in kernel space. Kernel space poses additional challenges due to a low-level API involving system calls, interrupt handling, and hardware access in device drivers. Due to the tremendous engineering effort involved, along with legacy and low-level code in the core kernel, we did not consider re-randomizing the entire kernel. Moreover, re-randomization incurs additional overheads. For these practical reasons, we re-randomize only the most vulnerable components. However, as we justify in Section 6, we gain strong protection against ROP gadget injection for the entire kernel ecosystem regardless of whether gadgets (e.g., in the core kernel) still exist.

Driver VMs, specialized OSs that run device drivers, exemplify one particular use case where Adelie additionally strengthens the security of a system which is already built with security considerations in mind. In fact, Adelie currently re-randomizes drivers in an enterprise-level system, which heavily relies on guest VM isolation to protect against external and internal malicious actors. Adelie, however, is more general and can be used for any Linux-based system.

Adelie is designed with Linux's source availability in mind. Unlike Shuffler and CodeArmor, both user-space solutions, Adelie avoids binary-level transformation. Also unlike them, Adelie implements 64-bit KASLR while using a zero-copying method for moving data and code, and efficiently keeps track of and unmaps previously used address ranges. Although CodeArmor also employs remapping, it lacks PIC support and consequently restricts ASLR to 32 bits. Adelie is also unique in the way it creates and handles multiple GOT tables. Adelie's GCC plugin greatly reduces the engineering effort by changing kernel modules automatically.

AVAILABILITY

Adelie's latest code is available at https://github.com/adelie-kaslr.

ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers and our shepherd Baris Kasikci for their insightful comments and suggestions, which helped greatly improve this paper.

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

This research is also based upon work supported by the U.S. Office of Naval Research (ONR) under grants N00014-18-1-2022 and N00014-19-1-2493 and U.S. Naval Surface Warfare Center Dahlgren Division/NAVSEA/NEEC under grant N00174-20-1-0009.

Adelie was integrated into an enterprise-level software infrastructure called SAVIOR (Secure Applications in Virtual Instantiations of Roles) system, which was developed as part of the IARPA VirtUE (Virtuous User Environment) program [38]. SAVIOR's source code repository is publicly available [47]. Adelie provides the Amazon ENA driver re-randomization in SAVIOR.

A ARTIFACT APPENDIX

A.1 Abstract

In this Appendix, we provide information about how to deploy Adelie using pre-installed VMs and run the benchmarks. We also describe the hardware and software requirements necessary to run the experiments and reproduce the results presented in Section 5.

Note that the experiments from the paper were run on physical machines, but we adapted the artifact to use VMs to make deployment and testing easier. As a result, we had to use E1000 rather than E1000E as a NIC driver. Also, the NVMe driver runs on top of an emulated rather than a physical device. Our artifact consists of two VMs: (1) the server VM with the modified Linux kernel and modules; (2) the client VM (load generator), where we keep benchmark scripts.

Our work specifically targets x86-64. Consequently, the provided artifact can only be deployed on x86-64 systems.

A.2 Artifact Check-List (Meta-information)

- **Program:** Modified Linux kernel and modules (Adelie). mySQL, Sysbench, Apache benchmarks are also included.
- **Compilation:** Ubuntu 18.04, GCC 8.4.
- **Transformations:** GCC plugins can be used for re-randomization.
- **Binary:** Pre-compiled kernel, kernel modules, and other binaries are provided in the VM images.
- **Run-time environment:** Ubuntu / VirtualBox.
- **Hardware:** Ideally, hardware described in Section 5 or similar.
- **Run-time state:** All VMs must be configured appropriately and placed on the same network. A correct host name, interface name, and other parameters must be specified on the client VM side. See below for more details.
- **Execution:** Automated via provided benchmark scripts.
- **Metrics:** Throughput.
- **Output:** Results are output to `/home/client/benchmark/results` on the client VM. Additionally, the status of re-randomization can be analyzed by running 'dmesg' (see below).
- **Experiments:** mySQL, Apache, IOCTL, Sysbench, etc.
- **How much disk space required (approximately)?:** 200GB (can be much less if not using FileIO).
- **How much time is needed to prepare workflow (approximately)?:** 30 minutes.
- **How much time is needed to complete experiments (approximately)?:** 2 hours (if not running many iterations).
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** The Linux kernel code uses Linux's original (GPLv2-only with the syscall exception) license unless specified otherwise. Other components that are not directly related to the Linux kernel (evaluation scripts, GCC plugins, etc) may be licensed under either the 3-Clause BSD or the GPLv2 license.
- **Archived (provide DOI)?:** 10.5281/zenodo.5831326

A.3 Description

A.3.1 How to Access. The artifact is available at https://doi.org/10.5281/zenodo.5831326.

The artifact contains source code, benchmark scripts, and pre-installed VM images that should be used with VirtualBox. You need to download the client (load generator) VM image (`Client.zip`) and the server VM image (`Adelie.zip`). Adelie's latest source code is also available at https://github.com/adelie-kaslr.

A.3.2 Hardware Dependencies. We recommend to use a system used for the paper evaluation (Xeon Silver 4114 2.20GHz, 96GB of RAM) or a system close to that.

A.3.3 Software Dependencies. We used VirtualBox 6.1.26 to test the provided VM images. It should also be possible to use more recent versions of VirtualBox. VirtualBox must be used for both the client VM and the server VM images. For simplicity, both the client VM and the server VM can be executed on the same machine.

The provided VM images need VirtualBox Extension Pack. In Ubuntu, VirtualBox and VirtualBox Extension Pack can be installed as follows:

```
$ sudo apt-get install virtualbox
$ sudo apt-get install virtualbox-ext-pack
```

A.4 Installation

The installation and compilation process is non-trivial and time-consuming. Although the system was evaluated on physical hardware (which uses E1000E, NVMe), we decided to provide virtual images with similar drivers (E1000, NVMe). Although the results may not necessarily be as precise as in the paper due to virtualization, we went with this approach to simplify deployment.

Adelie's version of the Linux kernel and all other dependencies are installed on the provided VM images. Since the compiled Linux source tree occupies around 20GB, we did not include it into the server VM image. If you wish to compile Adelie's version of the Linux kernel yourself, you can follow the instructions in `/home/asplos22/source/README.md` in the server VM image. We recommend using `/home/asplos22/mnt` for compilation since the root file system does not have sufficient space. Source code is also provided outside of the VM image in `source_code.zip`.

Before importing images into VirtualBox, you need to ensure that both images will use the same NAT network. First, go to `File->Preferences->Network` in VirtualBox. Then, click on the plus sign. Create a new network named "NatNetwork" with CIDR "10.0.2.0/24" and enable DHCP.

Then, import both the server and client VM images into VirtualBox. The easiest way to import VMs is to extract files directly into the `VirtualBox VMs` directory and then double click on the corresponding `.vbox` files. Once imported, VM settings can be adjusted. We recommend increasing the default memory size, especially if you are planning to run the mySQL benchmark. At least 8GB is preferable. You may also adjust the number of virtual CPUs (vCPUs) accordingly. (The default parameters are 1GB of RAM and 1 vCPUs for the client VM, 4GB of RAM and 2 vCPUs for the server VM.)

Note that we already provide sample configuration files along with the VDI images. Please make sure that your configuration uses an E1000-compatible adapter (e.g., Intel 82540EM). The server VM also requires attention when importing storage images. The server VM must contain three storage images: `Adelie_Boot` (the very first SATA image, the `/boot` directory), `Adelie` (the `/` NVMe partition, which is re-randomized), `Adelie_140G` (an additional 140GB-max NVMe partition which is used by FileIO tests and is mounted to `/home/asplos22/mnt`).

When done, you can launch both the client VM and the server VM and log in. **Username** and **password** for the server VM image are `asplos22` and `asplos22` correspondingly. **Username** and **password** for the client VM image are `client` and `client`.

A.5 Experiment Workflow

Running the experiments (and reboots) are automated through a script in the client VM. Please modify the configuration in the client VM (in the home directory):

```
$ vim benchmark/config.py
```

Specifically, change *HOSTNAME* to the IP address of the server (which can be identified by running 'ifconfig' on the server side). Also specify the correct interface name in *ETH_NAME* (which can also be observed in 'ifconfig'). You may also adjust other parameters via the config file, including which benchmarks you want to run (mySQL, Apache, etc.)

To run the benchmarks, use:

```
$ cd benchmark
$ python benchmark.py
```

Once the benchmarks complete, you can optionally generate plots by running:

```
$ python plots.py
```

A.6 Evaluation and Expected Results

Results are output to `/home/client/benchmark/results` on the client VM. Additionally, if plots are generated, they will be located at `/home/client/benchmark/plots`.

The benchmark script on the client VM will automatically reboot the server VM to load the correct version of the kernel. In the paper, we had different Linux kernels (e.g., retpoline enabled, retpoline disabled, etc). The differences are not very significant, thus we only provided one version of Adelie's 5.0.4-KASLR kernel with enabled retpoline and stack randomization. In this version, re-randomization can be enabled or disabled for the modules which were compiled with that option (e.g., e1000, nvme). Regardless of this, **all** modules use the position-independent code model as discussed in the paper. To further reduce the size of the server VM image, we also kept Ubuntu's out-of-the-box Linux kernel as the vanilla kernel.

Two different kernel versions will be evaluated by the benchmark scripts (vanilla and Adelie's KASLR). The 5.0.4-KASLR version will also run tests using re-randomization periods of varying lengths (i.e., 1 ms, 5 ms).

Whenever the 5.0.4-KASLR kernel is loaded and the re-randomization period is non-zero, you should be able to see something like this by running 'dmesg':

```
[  288.737300] Randomize: kthread started
[  289.023680] -----
[  289.023682] Randomized 53 times
[  289.023682] SMR Retire: 106
[  289.023683] SMR Free: 106
[  289.023683] SMR Delta: 0
[  289.023684] Stack Alloc: 530
[  289.023684] Stack Free: 530
[  289.023685] Stack Delta: 0
[  295.023676] -----
[  295.023677] Randomized 2117 times
[  295.023678] SMR Retire: 4234
[  295.023678] SMR Free: 4234
[  295.023679] SMR Delta: 0
[  295.023680] Stack Alloc: 21170
[  295.023680] Stack Free: 21170
[  295.023681] Stack Delta: 0
```

Generally speaking, results corresponding to figures in Section 5 (mySQL, Apache, IOCTL, etc) are expected. The only excluded test is kernbench. It is not fundamentally important, but it would require storing (and compiling) the entire Linux source tree (≈ 20GB) in the server image.

The benchmark scripts take care of re-randomization automatically. However, if desired, re-randomization of modules can also be started independently of the benchmark script (assuming that you boot up the 5.0.4-KASLR kernel). For example, to load the E1000 and NVMe modules with a re-randomization period of 20 ms, type:

```
$ sudo modprobe randmod \
>     module_names=e1000,nvme rand_period=20
```

Similarly, to stop re-randomization, run:

```
$ sudo rmmod randmod
```

A.7 Experiment Customization

We recommend customizing tests in config.py. For example, if you only wish to run Apache, you can simply specify:

```
TESTS_TO_RUN = ["APACHE"]
```

We also recommend reducing the number of iterations for Apache, especially if you need quicker testing. For this, you need to locate `def run_apache_sweep` in `benchmark.py` and change NUM_REQ to 5000 (or use another smaller value).

A.8 Notes

We recommend adjusting the number of vCPUs and increasing RAM size for VMs. Our default configuration for VMs does not use too many cores and memory to be on the safe side for everyone. Please be advised that the output might not exactly match the paper results due to (1) using a machine with different specs than in the paper; (2) running all tests using VMs rather than natively.

If you encounter any errors during tests, try increasing the resources of both the server and client VMs.

If desired, the contents of the VM images can be copied to physical partitions. Adelie can also be deployed to existing Linux installations by compiling from sources (see `source_code.zip` and the provided `README` file).

A.9 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

REFERENCES

[1] Advanced Micro Devices, Inc. 2021. AMD64 Architecture. (2021). https://developer.amd.com/resources/developer-guides-manuals/.

[2] Air Force Research Laboratory AFRL/RIEB. 2021. SecureView. https://www.ainfosec.com/technologies/secureview/.

[3] Apache Software Foundation. 2021. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.2/en/programs/ab.html.

[4] Luiz André Barroso and Urs Hölzle. 2007. The Case for Energy-Proportional Computing. *IEEE Computer* 40 (2007). https://doi.org/10.1109/MC.2007.443

[5] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. 2005. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD) *(SSYM '05)*. USENIX Association, 255–270.

[6] David Bigelow, Thomas Hobson, Robert Rudd, William Streilein, and Hamed Okhravi. 2015. Timely Rerandomization for Mitigating Memory Disclosures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. ACM, New York, NY, USA, 268–279. https://doi.org/10.1145/2810103.2813691

[7] Bill Mertka. 2018. Recent Linux vulnerabilities and the importance of patching. (2018). https://www.servercentral.com/blog/linux-vulnerabilities-importance-patching/.

[8] Andrea Bittau, Adam Belay, Ali Mashtizadeh, David Mazières, and Dan Boneh. 2014. Hacking Blind. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 227–242. https://doi.org/10.1109/SP.2014.22

[9] Tyler Bletsch, Xuxian Jiang, Vince W. Freeh, and Zhenkai Liang. 2011. Jump-oriented Programming: A New Class of Code-reuse Attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (Hong Kong, China) *(ASIACCS '11)*. ACM, New York, NY, USA, 30–40. https://doi.org/10.1145/1966913.1966919

[10] Pat Bohrer, Elmootazbellah N Elnozahy, Tom Keller, Michael Kistler, Charles Lefurgy, Chandler McDowell, and Ram Rajamony. 2002. The case for power management in web servers. In *Power aware computing*. Springer, 261–289. https://doi.org/10.1007/978-1-4757-6217-4_14

[11] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC. In *Proceedings of the 15th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '08)*. ACM, 27–38. https://doi.org/10.1145/1455770.1455776

[12] Canonical Ltd. 2021. Ubuntu Security Features. (2021). https://wiki.ubuntu.com/Security/Features.

[13] Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. 2010. Return-oriented Programming Without Returns. In *Proceedings of the 17th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '10)*. ACM, 559–572. https://doi.org/10.1145/1866307.1866370

[14] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. 2011. Linux Kernel Vulnerabilities: State-of-the-Art Defenses and Open Problems. In *Proceedings of the 2nd Asia-Pacific Workshop on Systems* (Shanghai, China) *(APSys '11)*. ACM, New York, NY, USA, Article 5, 5 pages. https://doi.org/10.1145/2103799.2103805

[15] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. 2005. Non-control-data Attacks Are Realistic Threats. In *Proceedings of the 14th USENIX Security Symposium* (Baltimore, MD) *(SSYM '05)*. USENIX Association, Berkeley, CA, USA, 177–191.

[16] X. Chen, H. Bos, and C. Giuffrida. 2017. CodeArmor: Virtualizing the Code Space to Counter Disclosure Attacks. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 514–529. https://doi.org/10.1109/EuroSP.2017.17

[17] Yue Chen, Zhi Wang, David Whalley, and Long Lu. 2016. Remix: On-demand Live Randomization. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy* (New Orleans, LA, USA) *(CODASPY '16)*. ACM, 50–61. https://doi.org/10.1145/2857705.2857726

[18] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. *SIGOPS Oper. Syst. Rev.* 35, 5 (Oct. 2001), 73–88. https://doi.org/10.1145/502059.502042

[19] John Criswell, Nathan Dautenhahn, and Vikram Adve. 2014. KCoFI: Complete Control-Flow Integrity for Commodity Operating System Kernels. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14)*. IEEE Computer Society, Washington, DC, USA, 292–307. https://doi.org/10.1109/SP.2014.26

[20] Charlie Curtsinger and Emery D. Berger. 2013. STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 219–228. https://doi.org/10.1145/2451116.2451141

[21] CVE. 2021. The number of vulnerabilities. (2021). https://cve.mitre.org.

[22] CVE Details. 2021. Linux Kernel Vulnerabilities. (2021). https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/cvssscoremin-7/cvssscoremax-7.99/Linux-Linux-Kernel.html.

[23] CVE Details. 2021. MacOS X Security Vulnerabilities. (2021). https://www.cvedetails.com/vulnerability-list/vendor_id-49/product_id-156/Apple-Mac-Os-X.html.

[24] CVE Details. 2021. Windows 10 Security Vulnerabilities. (2021). https://www.cvedetails.com/vulnerability-list/vendor_id-26/product_id-32238/Microsoft-Windows-10.htmll.

[25] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. 2014. Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection. In *Proceedings of the 23rd USENIX Security Symposium (Security '14)*. 401–416.

[26] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. 2015. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 901–913. https://doi.org/10.1145/2810103.2813646

[27] Ubuntu Foundation. 2017. Weekly Newsletter, 2017-06-15. (2017). https://lists.ubuntu.com/archives/ubuntu-devel/2017-June/039816.html.

[28] Keir Fraser. 2004. *Practical lock-freedom.* Technical Report UCAM-CL-TR-579. University of Cambridge, Computer Laboratory. http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.pdf.

[29] Mark Gallagher, Lauren Biernacki, Shibo Chen, Zelalem Birhanu Aweke, Salessawi Ferede Yitbarek, Misiker Tadesse Aga, Austin Harris, Zhixing Xu, Baris Kasikci, Valeria Bertacco, Sharad Malik, Mohit Tiwari, and Todd Austin. 2019. Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 469–484. https://doi.org/10.1145/3297858.3304037

[30] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *Proceedings of the 21st USENIX Security Symposium* (Bellevue, WA) *(Security '12)*. USENIX Association, Berkeley, CA, USA, 475–490.

[31] Michael Grace, Zhi Wang, Deepa Srinivasan, Jinku Li, Xuxian Jiang, Zhenkai Liang, and Siarhei Liakh. 2010. Transparent Protection of Commodity OS Kernels Using Hardware Virtualization. In *SecureComm '10*. 162–180. https://doi.org/10.1007/978-3-642-16161-2_10

[32] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. KASLR is Dead: Long Live KASLR. In *Engineering Secure Software and Systems - 9th International Symposium, ESSoS 2017, Proceedings*, Vol. 10379 LNCS. Springer-Verlag Italia, Italy, 161–176. https://doi.org/10.1007/978-3-319-62105-0_11

[33] Kernel Hardening. 2019. PIE support for Linux. (2019). https://www.openwall.com/lists/kernel-hardening/2019/01/31/5.

[34] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. 2007. Performance of memory reclamation for lockless synchronization. *J. Parallel and Distrib. Comput.* 67, 12 (2007), 1270 – 1285. https://doi.org/10.1016/j.jpdc.2007.04.010 Best Paper Awards: 20th International Parallel and Distributed Processing Symposium (IPDPS 2006).

[35] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. 2006. Reorganizing UNIX for Reliability. In *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture* (Shanghai, China) *(ACSAC '06)*. 81–94. https://doi.org/10.1007/11859802_8

[36] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'D My Gadgets Go?. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 571–585. https://doi.org/10.1109/SP.2012.39

[37] Lu H.J., Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger, and Mark Mitchell. 2018. System V Application Binary Interface AMD64 Architecture Processor Supplement Version 1.0. (2018). https://github.com/hjl-tools/x86-psABI/wiki/x86-64-psABI-1.0.pdf.

[38] IARPA. 2021. VirtUE (Virtuous User Environment). https://www.iarpa.gov/index.php/research-programs/virtue.

[39] Intel Corporation. 2021. Intel 64 and IA-32 architectures software developer's manual. (2021). https://software.intel.com/en-us/articles/intel-sdm.

[40] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. (2018). https://spectreattack.com/spectre.pdf.

[41] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent Byunghoon Kang. 2017. Hacking in Darkness: Return-Oriented Programming against Secure Enclaves. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC '17)*. USENIX Association, USA, 523–539.

[42] J. Li, Z. Wang, T. Bletsch, D. Srinivasan, M. Grace, and X. Jiang. 2011. Comprehensive and Efficient Protection of Kernel Control Data. *IEEE Transactions on Information Forensics and Security* 6, 4 (Dec 2011), 1404–1417. https://doi.org/10.1109/TIFS.2011.2159712

[43] Zhuohua Li, Jincheng Wang, Mingshen Sun, and John C.S. Lui. 2019. Securing the Device Drivers of Your Embedded Systems: Framework and Prototype. In *Proceedings of the 14th International Conference on Availability, Reliability and Security* (Canterbury, CA, United Kingdom) *(ARES '19)*. ACM, New York, NY, USA, Article 71, 10 pages. https://doi.org/10.1145/3339252.3340506

[44] LWN.net. 2004. x86 NX support. (2004). https://lwn.net/Articles/87814/.

[45] LWN.net. 2013. Kernel address space layout randomization. (2013). https://lwn.net/Articles/569635/.

[46] LWN.net. 2014. "Strong" stack protection for GCC. (2014). https://lwn.net/Articles/584225/.

[47] Next Century. 2019. SAVIOR (Secure Applications in Virtual Instantiations of Roles). https://github.com/NextCenturyCorporation/VirtUE.

[48] Ruslan Nikolaev and Binoy Ravindran. 2019. Brief Announcement: Hyaline: Fast and Transparent Lock-Free Memory Reclamation. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. ACM, New York, NY, USA, 419–421. https://doi.org/10.1145/3293611.3331575

[49] Ruslan Nikolaev and Binoy Ravindran. 2021. Snapshot-Free, Transparent, and Robust Memory Reclamation for Lock-Free Data Structures. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI '21)*. ACM, New York, NY, USA, 987–1002. https://doi.org/10.1145/3453483.3454090

[50] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. 2011. Faults in Linux: Ten Years Later. *SIGPLAN Not.* 46, 3 (March 2011), 305–318. https://doi.org/10.1145/1961296.1950401

[51] PaX project. 2021. ASLR Design Document. (2021). https://pax.grsecurity.net/docs/aslr.txt.

[52] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2017. kR^X: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proceedings of the 12th European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. ACM, New York, NY, USA, 420–436. https://doi.org/10.1145/3064176.3064216

[53] Aravind Prakash and Heng Yin. 2015. Defeating ROP Through Denial of Stack Pivot. In *Proceedings of the 31st Annual Computer Security Applications Conference* (Los Angeles, CA, USA) *(ACSAC 2015)*. ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/2818000.2818023

[54] Ryan Riley, Xuxian Jiang, and Dongyan Xu. 2008. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection* (Cambridge, MA, USA) *(RAID '08)*. Springer-Verlag, 1–20. https://doi.org/10.1007/978-3-540-87403-4_1

[55] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. 2012. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Trans. Inf. Syst. Secur.* 15, 1, Article 2 (March 2012), 34 pages. https://doi.org/10.1145/2133375.2133377

[56] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. 2009. Surgically Returning to Randomized Lib(C). In *Proceedings of the 2009 Annual Computer Security Applications Conference (ACSAC '09)*. IEEE Computer Society, Washington, DC, USA, 60–69. https://doi.org/10.1109/ACSAC.2009.16

[57] ROPgadget project. 2021. ROPgadget Tool. (2021). https://github.com/JonathanSalwan/ROPgadget.

[58] Joanna Rutkowska and Rafal Wojtczuk. 2010. Qubes OS architecture. Invisible Things Lab Tech Rep. https://www.qubes-os.org/attachment/doc/arch-spec-0.3.pdf

[59] Sascha Schirra. 2021. Ropper. https://github.com/sashs/Ropper.

[60] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. 2011. Q: Exploit Hardening Made Easy. In *Proceedings of the 20th USENIX Security Symposium* (San Francisco, CA) *(SEC '11)*. USENIX Association, Berkeley, CA, USA, 1–16.

[61] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[62] Kevin Z. Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2013. Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13)*. IEEE Computer Society, Washington, DC, USA, 574–588. https://doi.org/10.1109/SP.2013.45

[63] Richard M. Stallman and the GCC Developer Community. 1988-2018. *GNU Compiler Collection Internals*. Free Software Foundation, Inc., Chapter 24, 665–672.

[64] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. 2006. Recovering Device Drivers. *ACM Trans. Comput. Syst.* 24, 4 (Nov. 2006), 333–360. https://doi.org/10.1145/1189256.1189257

[65] SysBench. 2021. A System Performance Benchmark. http://sysbench.sourceforge.net/.

[66] Paul Turner. 2018. Retpoline: a software construct for preventing branch-target-injection. (2018). https://support.google.com/faqs/answer/7625886.

[67] Zhi Wang, Xuxian Jiang, Weidong Cui, and Peng Ning. 2009. Countering Kernel Rootkits with Lightweight Hook Protection. In *Proceedings of the 16th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '09)*. ACM, 545–554. https://doi.org/10.1145/1653662.1653728

[68] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (Raleigh, North Carolina, USA) *(CCS '12)*. ACM, New York, NY, USA, 157–168. https://doi.org/10.1145/2382196.2382216

[69] David Williams-King, Graham Gobieski, Kent Williams-King, James P. Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P. Kemerlis, Junfeng Yang, and William Aiello. 2016. Shuffler: Fast and Deployable Continuous Code Re-randomization. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, Berkeley, CA, USA, 367–382.

[70] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. 2020. Egalito: Layout-Agnostic Binary Recompilation. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*. ACM, New York, NY, USA, 133–147. https://doi.org/10.1145/3373376.3378470

[71] Xen Project. 2021. PCI Passthrough. https://wiki.xenproject.org/wiki/Xen_PCI_Passthrough.

[72] J. Xu, Z. Kalbarczyk, and R. K. Iyer. 2003. Transparent runtime randomization for security. In *Proceedings of the 22nd International Symposium on Reliable Distributed Systems*. 260–269. https://doi.org/10.1109/RELDIS.2003.1238076