kMVX: Detecting Kernel Information Leaks with Multi-variant Execution

Sebastian Österlund*
s.osterlund@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Koen Koning*
koen.koning@vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Pierre Olivier
polivier@vt.edu
Virginia Tech
Blacksburg, Virginia

Antonio Barbalace
abarbala@stevens.edu
Stevens Institute of Technology
Hoboken, New Jersey

Herbert Bos
herbertb@cs.vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Cristiano Giuffrida
giuffrida@cs.vu.nl
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands

Abstract

Kernel information leak vulnerabilities are a major security threat to production systems. Attackers can exploit them to leak confidential information such as cryptographic keys or kernel pointers. Despite efforts by kernel developers and researchers, existing defenses for kernels such as Linux are limited in scope or incur a prohibitive performance overhead.

In this paper, we present kMVX, a comprehensive defense against information leak vulnerabilities in the kernel by running multiple diversified kernel variants simultaneously on the same machine. By constructing these variants in a careful manner, we can ensure they only show divergences when an attacker tries to exploit bugs present in the kernel. By detecting these divergences we can prevent kernel information leaks. Our kMVX design is inspired by multi-variant execution (MVX). Traditional MVX designs cannot be applied to kernels because of their assumptions on the run-time environment. kMVX, on the other hand, can be applied even to commodity kernels. We show our Linux-based prototype provides powerful protection against information leaks at acceptable performance overhead (20–50% in the worst case for popular server applications).

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

*Equal contribution joint first authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASPLOS'19, April 13–17, 2019, Providence, RI, USA
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6240-5/19/04…$15.00
https://doi.org/10.1145/3297858.3304054

Keywords operating systems; security; information leaks; multi-variant exection

ACM Reference Format:
Sebastian Österlund, Koen Koning, Pierre Olivier, Antonio Barbalace, Herbert Bos, and Cristiano Giuffrida. 2019. kMVX: Detecting Kernel Information Leaks with Multi-variant Execution. In *2019 Architectural Support for Programming Languages and Operating Systems (ASPLOS '19), April 13–17, 2019, Providence, RI, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3297858.3304054

1 Introduction

With millions of lines of code, the operating system (OS) is typically one of the most complex pieces of software on a machine. All the research into alternative OS designs and safer languages notwithstanding, monolithic kernels such as Linux, Windows, and BSD (all written in unsafe languages), are still the norm. Unfortunately, such kernels offer a large attack surface. For instance, the Linux kernel contains a long list of exploitable bugs [5, 13, 27, 32, 35, 53], with the most common class of vulnerabilities being that of *information leaks* [27]. Such vulnerabilities allow an unprivileged attacker to extract sensitive information such as crypto keys or pointers from the kernel. Not only do these bugs compromise the confidentiality of the kernel, but they are often critical to subsequent attacks such as privilege escalation [45].

While developers and researchers have proposed numerous defenses and detection tools in response [13–15, 20, 26, 34, 38, 41, 46], their solutions tend to be either too limited in scope and detection capability, or too expensive in terms of overhead. For instance, kernel address space layout randomization (kASLR) in its current form only randomizes the base of the kernel, meaning a single leaked pointer compromises the entire randomization. The built-in KASAN memory error detector [15] can only detect use-after-free and out-of-bounds bugs, at the price of a 4x performance overhead. Meanwhile, kmemcheck [14], another built-in memory checker, can also detect uninitialized reads, but at the cost of several orders of magnitude of overhead. More recent efforts, such as UniSan [26], efficiently mitigate uninitialized reads, but does not cover all the other information leaks.

RIGHTSLINK

In this paper, we present a new technique, called kMVX, which can efficiently detect arbitrary information leaks by running multiple diversified kernels simultaneously on the same machine. These *variants* are constructed in such a way that they exhibit the same behavior in normal circumstances, but show *diverging behavior* if an attacker tries to exploit the system. We achieve this by making particular changes in the memory layout of each variant, which make no difference during benign execution, but do matter for an attacker trying to leak information from the kernel. While the *variant generation* provides the security, our kMVX design also requires components to facilitate running multiple kernels on the same machine and detect the divergences.

The design of kMVX is based on that of multi-variant execution (MVX) [3, 8, 19, 40, 51]. MVX is a user-level defense that runs multiple variants of an application side-by-side and checks their behavior. Traditional MVX synchronization principles are not applicable to kernels, however, as they often lack such clear and strict interfaces, can interact with system resource and hardware directly, and contain many sources of non-determinism (e.g., task scheduling). kMVX addresses this challenge by introducing two synchronization points, called the *I/O sync* and *syscall sync*.

We constructed a kMVX prototype based on Linux, which shows our kMVX design applies to commodity operating systems and is effective at detecting kernel info leaks. Our prototype runs multiple Linux kernels on the same machine while preserving the user space ABI of Linux, allowing existing applications to run unmodified on the system. Experiments show our prototype has at most 20–50% overhead for real-world server applications.

To summarize, our contributions are three-fold:

- A design for kMVX that supports running and monitoring multiple kernel variants on a single system.
- Multiple variant generation strategies applicable to the kernel to stop information leaks.
- An evaluation of a kMVX prototype on Linux that demonstrates its effectiveness and efficiency.

2 Background

OS kernel vulnerabilities Similar to most large projects written in unsafe languages, the Linux kernel contains a large number of bugs, with more being discovered—and even introduced—every month. Despite the numerous efforts to add detection and protection mechanisms, studies show that there is still a wide variety of bugs present in the kernel [5, 26]. For instance, in their 2010 study, Chen *et al.* [5] identified four major classes of exploits in the Linux kernel: memory corruption, policy violation, denial of service, and information leaks. The latter was, and still is, by far the most dominant class of vulnerabilities [26].

This motivates our focus on *information leak* attacks in which attackers abuse such vulnerabilities to induce kernel data to leave the kernel, for instance to be sent over a socket or copied to user memory. A common cause of such info leaks are data structures where some fields are uninitialized before passing it to the network stack or userland. A recent study shows that some of these info leaks are also present in data structures where the compiler inserts padding bytes that are not initialized, both on the stack and the heap [26].

```
struct snd_timer_tread {
   int event; // 4 bytes padding
   struct timespec tstamp;
   unsigned int val; // 4 bytes padding
};
int snd_timer_user_params(...)
{
   struct snd_timer_tread tread;
   tread.event = SNDRV_TIMER_EVENT_EARLY;
   tread.tstamp.tv_sec = 0;
   tread.tstamp.tv_nsec = 0;
   tread.val = 0;
   // Padding uninitialized at this point
   // ...
   copy_to_user(usr_buffer, &tread,
            sizeof(struct snd_timer_tread));
}
```

Listing 1. CVE-2016-4569: The compiler adds several bytes of padding to the `struct snd_timer_tread` in `sound/core/timer.c`. The `copy_to_user` call will leak the uninitialized padding bytes to the user.

As an example, both CVE-2014-1444 and CVE-2016-4569 concern a data structure allocated on the stack where the compiler adds padding, and can thus leak information to user space when called via an `ioctl` syscall (see Listing 1). If a previous stack frame contained sensitive data, such as kernel pointers, these will be copied alongside the data structure. CVE-2013-2237 is an example where an object is allocated on the kernel heap without being fully initialized, and then sent over a socket, again leading to an info leak.

While info leaks themselves may be already harmful, they are often also used for further attacks. For instance, when kASLR is enabled, an attacker typically first has to leak pointers before mounting more complex exploits. An example is CVE-2016-0728, a use-after-free bug in the keyring management. An attacker can force the kernel to de-allocate an object while it still holds pointers to it. The attacker can then allocate a new object (of a different type), which the kernel will interpret as the old object. In this particular case, an attacker can place arbitrary kernel pointers inside the object which the kernel will then call, leading to a privilege escalation. In cases where kASLR is enabled, the attacker first needs to determine the correct pointers for the last step of this attack via an info leak.

RIGHTSLINK

Multi-variant execution kMVX draws from user space MVX, which has been applied to programs ranging from servers to graphical applications [3, 8, 12, 19, 36, 37, 40, 52]. The core idea of MVX is to generate variants that have the same *outside* behavior (or output) given the same inputs in normal situations, but start to diverge when an attacker tries to exploit a vulnerability in the program. A simple example is running the same program twice with address space layout randomization (ASLR) enabled. ASLR variations have no observable effect on the program execution from the outside: when the same input is applied to both variants they will return the same output. However, if the application contains a bug that allows an attacker to trigger a read at a specific memory location, via a pointer, the application may return arbitrary memory content, hence we will observe divergent behavior. To be specific, both variants will receive the same pointer, but due to ASLR the pointer will most likely have different contents in each variant.

The security guarantees of MVX and kMVX systems are determined by the diversification strategies of the *variant generator*. The design of MVX and kMVX allows for variant generation strategies to be swapped and combined, based on the threat model. Schemes with stronger (deterministic) guarantees often require large amounts of resources [19] or introduce compatibility issues [40]. One important insight for MVX and kMVX is that the amount of entropy itself does not provide the security (like it does for ASLR), instead the security comes from the entropy causing *some* divergence in execution for attacks. Creating suitable variant generation strategies for kMVX, that work with the strict requirements of kernel resources and provide full coverage against information leaks, is a key challenge we address in the paper.

When running multiple variants, it is crucial both have the same view of the outside world to avoid benign divergences. For MVX this is relatively easy, since user space programs have a strictly defined I/O interface in the form of syscalls. For instance, both variants should have the same view of time and network traffic, which are both accessed via syscalls. The kernel does not have such an interface, creating a number of challenges for kMVX, which we address in Section 4.

3 Threat model

MVX in general can be applied to a large number of different exploits targeting applications. All the related vulnerabilities can be found in the kernel as well. As kMVX is a new design, for this paper we primarily focus on attacks on a *local* system, where *information is leaked* via bugs in the kernel code.

We assume a *local* attacker already in full control of an unprivileged user space program, who tries to disclose pointers or sensitive information (e.g., cryptographic keys) from the kernel by (repeatedly) interacting with it via syscalls and exploiting info leak vulnerabilities. We focus on exploitation of such vulnerabilities via software bugs and assume orthogonal defenses for other attack vectors such as side channels.

Due to defenses such as kASLR [49], a majority of Linux kernel attacks rely on leaking information at some stage in the attack. By eliminating info leaks, we hinder many other classes of exploits, such as privilege escalation.

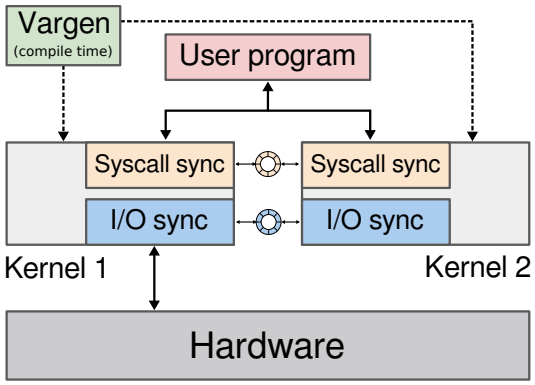

Figure 1. Overview of the kMVX design. Two kernels run on the same hardware by controlling interactions with the hardware via the *I/O sync* component. Interactions with user space go through the *syscall sync* to detect divergences. *Vargen* constructs diversified kernels during compilation.

4 kMVX: Kernel multi-variant execution

Our design of kMVX runs two co-existing OS kernels simultaneously on the same (virtual or physical) hardware. Each kernel is diversified using our *variant generator* (*vargen*) to cause divergence for information leaks. Each kernel includes two components to both keep execution consistent and check for divergences caused by malicious users: the *I/O sync* and *syscall sync*. These two components are similar in spirit to the monitor of traditional MVX. An overview of the components of kMVX can be seen in Figure 1.

The I/O sync part, placed at the hardware-kernel interface, is responsible for preventing spurious divergences by providing a uniform interface to the (shared) hardware. For instance it provides both kernels with the same view of time and the network. Information leaks are detected by divergent behavior at the user space boundary in the syscall sync. Given the local attacker in our threat model, the I/O sync does not need to check for divergences, simplifying the design.

kMVX follows the conventional leader-follower design for MVX [12, 19, 51]. In Figure 1, kernel 1 is the leader and performs actual I/O operations. The other kernel is the follower, which synchronizes with the leader via an in-memory communication channel.

For variant generation, we ensure the virtual memory address spaces of the two kernels do not overlap. In addition, we diversify the different allocators in the kernel and randomize the stack usage with a compiler pass. For instance,

RIGHTSLINK

we modified the `kmalloc` allocator in Linux to be type-based, both to prevent use-after-free attacks but also to generate a different memory layout for objects between the variants. All these diversifications together provide us with a strong defense against info leaks through software bugs.

Although running multiple kernels is also possible via *virtualization*—running each kernel variant in a separate virtual machine—this is not sufficient for kMVX. The level of synchronization required for kMVX requires more in-depth knowledge of the kernel state and its ordering (e.g., scheduling between cores). Similarly the syscall sync, in particular data transfers between user space and the kernel, would require kernel modifications to work properly with a virtual machine monitor (VMM). Overall, the kernels still require significant modifications despite using a VMM, without virtualization giving any significant benefits or portability. As such, we do not use virtualization for our kMVX design.

In kMVX both kernels are partitioned in physical memory, and limited to certain CPU cores. Only the leader has access to all the hardware on the machine, such as the network interface, whereas the followers have to communicate with the leader to interact with the outside world. In our kMVX design, our aim is to place the I/O sync as close to the hardware boundary as possible, maximizing the kernel code executed in each kernel.

4.1 Syscall synchronization

The syscall sync component at the user space boundary is responsible for detecting sensitive information leaking to the attacker. Because of the variant generation, such information will differ when returned to user space, which is then detected. Operating systems have a strict interface for copying data to the user (in Linux there are `copy_to_user`, `put_user`, and the syscall return value) which our syscall sync interposes. Any data copied to the user is placed in the shared ringbuffer to be validated for divergence. If any divergence is detected the faulty bytes are zeroed, to eliminate the info leak without stopping execution (but logging the event). This is especially important for compatibility, since bugs may cause non-malicious info leaks during regular execution.

Because data can only be returned after being checked and (potentially) being zeroed, this effectively enforces lockstep behavior between the variants for copying data to the user. Previous user space MVX designs had a notion of *policies*, where only certain (critical) syscalls are running in lockstep, allowing the user to tune the trade-off between security and performance [12, 19, 51]. Our kMVX design effectively enforces such a lockstep policy since syscalls that copy data to the user are considered *critical syscalls*.

Placing the monitor inside the variants themselves does not compromise the security of our design, since it is mapped at a different location in both kernels (Section 4.3). Any access to this area first requires the attacker to leak its location, which is prevented by the kMVX monitor itself.

4.2 I/O sync

In order to implement kMVX, we need multiple variants of the OS kernel running simultaneously on the same hardware. However, commodity operating systems such as Linux are clearly not intended to run multiple kernel instances. More importantly, when multiple kernel instances natively run on the same hardware, the management of the hardware state, specifically device state—a task performed only by the kernel—must be done carefully. Note that when applying MVX to user space programs, the I/O interface is clearly defined with syscalls. However, for drivers and their interactions with hardware there is generally no such interface [21].

If two kernels share the same set of cores, the execution may need to be serialized, requiring the leader to wait a significant time for the follower to finish before finishing synchronization. Instead we parallelize the implementation, associating a fixed partition of cores to each kernel. This allows both kernels to execute at the same time, greatly reducing the waiting time required for synchronization.

4.3 Variant generation

For our variant generation, we modify existing kernel allocators to yield different usage patterns between variants. In particular, we apply variant generation by partitioning the address space, modifying dynamic allocators such as the SLAB allocator, and changing the format of the stack with a compiler pass. Variant generation is, in some cases, highly dependent on the software. In this section we focus on Linux, but all techniques have more general applicability.

4.3.1 Address space partitioning

Partitioning of the address space is a well-known variant generation scheme for MVX that ensures any pointer can only be valid in at most one variant at a time [8]. Since kernel pointers are a common target for info leaks, making sure these always differ between variants means they are impossible to leak, since that immediately causes a divergence.

Userspace programs are characterized by a very large address space: the OS provides programs with a virtual address space that can span up to 128 TB on x86-64. Our experiments show that less than 1% of that is virtually mapped in a process. Moreover, all regions mapped into userland (e.g., stack, heap, libraries, binary) are very small on their own and can be arbitrarily placed anywhere in the address space during runtime. The kernel address space layout, on the other hand, is strictly defined and determined at compile time.

Luckily, because virtual address ranges are much larger than the actual physical memory-addressable resources of most computers, the 128 TB of kernel virtual address space on x86-64 can be split into two separate partitions without significant drawbacks, enabling address space partitioning of the kernel memory. By doing so, we ensure that any leaked pointer will differ and trigger detection. For partial pointer

RIGHTSLINK

leaks, where only some of the bytes of a pointer are leaked, we add additional entropy on each variants relying on the existing kASLR mechanisms. Recall that the amount of entropy is not very important, since we just needs *some* variation.

4.3.2 Dynamic allocation

OS kernels such as Linux generally offer several ways of dynamically allocating memory. For Linux, of particular interest are the SLAB allocator and `vmalloc`. With `vmalloc` it is possible to allocate one or more pages, which are virtually but not physically consecutive, and allocations are always rounded to a multiple of the page size. The SLAB allocators, on the other hand, reserve a number of pages for a SLAB cache, and can then hand out objects of a smaller size from that cache. Each cache contains objects of a single size. Most OS kernels contain a similar SLAB allocator, including Linux, Solaris, FreeBSD, and NetBSD.

To provide variation in these dynamic allocators, we make sure one variant will follow different allocation patterns than the other. In particular, for the `vmalloc` allocator, we simply add guard pages around each allocation. Since `vmalloc` is rarely used and its allocations are not physically contiguous, this adds minimal overhead.

The pressure and constraints on the SLAB allocator are much higher, and therefore it requires a different scheme to provide diversity. In our design we change one variant to have a type-based SLAB allocator. While Linux partially provides this interface, allocations made with `kmalloc` still use (untyped) generic caches. Furthermore, Linux by default merges all caches with the same size. By making the SLAB allocator type-based, we not only get type-safety, but we also get different allocation patterns for every cache, since each data structure now has its own cache.

4.3.3 Stack frames

Since a lot of info leaks originate from the kernel stack it is important to provide variation there as well. The layout of the stack is determined by the compiler, and cannot be changed after compilation. We identify two types of info leaks on the stack: an overflow into another variable on the stack or an uninitialized read on the stack (Fig. 2a). For each of these we propose lightweight variation techniques using a compiler pass, leading to a different layout, causing different behavior for both classes of errors.

To protect against out-of-bound reads on the stack we change the kernel stack frame layout: by modifying the order or size of every stack variable an over-read will end up reading a different variable in each variant. For uninitialized reads, the read will instead read contents that were on the stack before from an old stack frame, as shown in Figure 2a. These contents were placed there by a previous function call, from another path in the call-chain. To protect against such errors, we randomize the distance between stack frames as shown in Figure 2b. In each variant the stack frame of the function containing the uninitialized read will end up in a different location. Moreover, its offset relative to the old stack frame (`func2` in Figure 2) will be different. While both variants might read sensitive data on their own, the chance that they both read the same sensitive data is marginal.

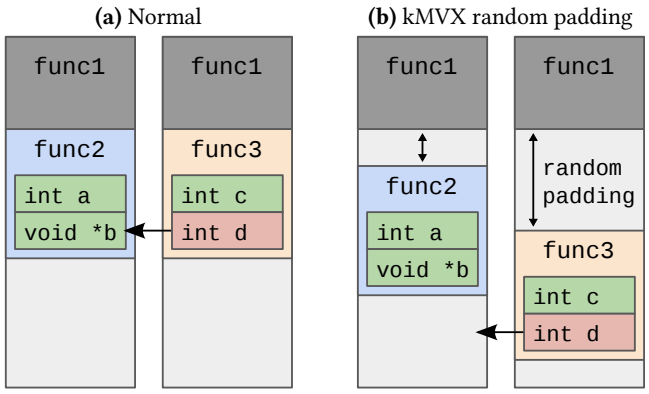

Figure 2. The layout of the stack with different stack frames per function (stack grows downward). This shows the effect of uninitialized reads with and without stack frame padding. In both cases `func1` first calls `func2`, then `func3`. In `func3` there is an uninitialized read on the variable `d`, which would read the value that was previously there, the pointer `b`, but with random padding between frames the value will differ.

5 Implementation

As previously discussed, kMVX requires four building blocks: 1) multiple kernel instances, 2) variant generation, 3) syscall sync, and 4) I/O sync. In this section we will explain in detail how we modified the Linux kernel to implement these.

Popular operating systems capable of running **multiple kernel instances** include Barrelfish [2] and Amoeba [30]. For compatibility with existing software and to show that our design can be applied to production-use operating systems, we implemented a prototype of the kMVX design for Linux, on top of FT-Linux [25]. FT-Linux allows multiple Linux kernel instances to run on the same multicore machine for the purpose of fault tolerance, but provides none of the variant generation and synchronization functionalities part of our kMVX design. FT-Linux is based on the Linux 3.2.12 kernel and targets the Intel x86-64 architecture. We firmly believe that our design can be applied to other OS kernels and enabled on other ISAs as well.

In kMVX the hardware resources of the machine are split into two partitions: each kernel instance gets a set of cores and its own reserved memory. We run all unprivileged user space processes in a special kernel namespace with syscall and I/O sync enabled. When a process is started in this namespace, the same process is also created on the follower, having the same PID, file descriptors, and environment.

RIGHTSLINK

Contrary to the design shown in Figure 1, in our prototype each kernel runs a separate version of the user program for each kernel variant. Most system processes run only once on a machine: only unprivileged applications run on both variants. Running not only the kernel but also applications twice is easier to support, requiring less communication between the kernels, and most importantly eliminates sharing of possibly critical data structures between variants. By strictly separating the memory we reduce the possibility of circumventing kMVX by leaking shared data. In other words, running two separate versions of the user program introduces more variance in the memory layout of the kernel and allows the variants to have strictly separated memory. Similarly, by running the application on the follower, we also introduce variance with scheduling, which may be a source of bugs (e.g., race conditions). Note that replicating the user space application is an implementation detail that can be changed, as it is not fundamental to our design.

5.1 Variant generation

5.1.1 Address space partitioning

We constructed an address space partitioning mechanism, which allows us to linearly split the physical memory space of one machine between different kernels. This feature implicitly enables the Linux kernel to deflate and load its image at any physical address. We can freely set these addresses with kernel boot parameters.

For the virtual addresses, the Linux kernel divides its address space into several regions (direct physical mappings, vmalloc, virtual memory map, kernel text, and modules). To implement address space partitioning, we logically halved the size of each virtual region and assigned the first half to the leader and the second half to the follower, thus keeping the original base address for the leader, and assigned a new one to the follower, as shown in Figure 3b. This has the side effect of reducing the maximum amount of supported physical memory to 32 TB. For this purpose, we had to slightly modify the kernel to ensure each memory region could be freely relocated in the virtual address space. On recent Linux versions, most memory regions are subject to kASLR already, enforcing the memory regions to be in a predefined order and size. We modified the kernel to ensure physical-to-virtual offsets, order, and size can be freely changed and randomized at compile time for all the virtual memory regions, addressing the relocation problem to support address space partitioning.

5.1.2 Stack and heap variation

Besides *address space partitioning*, we implemented all vargen techniques described in Section 4.3 for our prototype, providing us with a high level of security against various classes of vulnerabilities. Note that we can defend against more classes of vulnerabilities by adding more variation techniques. In this paper we focus on a few stack- and heap

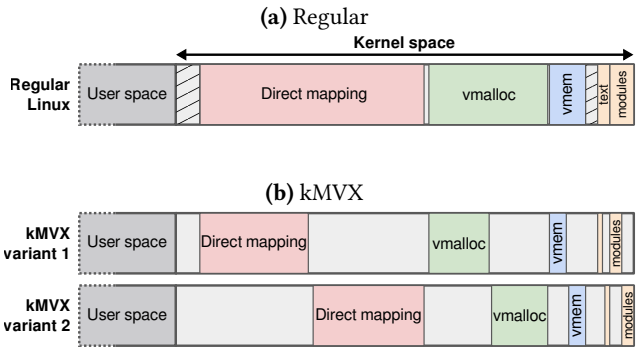

Figure 3. Regular kernel virtual address space, and the kMVX partitioned non-overlapping address space.

variation techniques. Other memory layout randomization techniques, such as struct field order randomization [11], have successfully been applied to Linux [43].

For our prototype we modified the SLUB allocator (used for `kmalloc` and `kmem_cache_alloc`) to be type-based. For the untyped `kmalloc` allocator, we assign types based on the call site, looking at the return addresses as unique type signature [1, 48]. For dynamically sized objects such as strings, we still use the old `kmalloc` (size-based) caches. Changing the allocator introduces variance in the heap layout, providing (probabilistic) protection against heap-based vulnerabilities, including use-after-free vulnerabilities.

For the stack variation we modified the GCC compiler (version 6.0) to add random padding to stack frames, and reverse the order of variables within stack frames. These variations have minimal impact on performance, while providing kMVX with strong protection against stack-based info leaks, as detailed in Section 4.3.3.

5.2 Syscall sync

Our kMVX prototype synchronizes on every `copy_to_user` and `put_user` call. When the leader encounters a synchronization point, the contents of the copied data is sent from the primary to the follower using a fast inter-kernel communication channel. Each message sent between the variants is given a unique id, based on the current process id, syscall id, and a message counter, allowing us to match a certain sync call that originates from the same syscalls on both variants.

Upon entering the `copy_to_user`-call, the follower reads the data from the channel and does a byte-by-byte comparison of the buffers. If the expected message is not available in the channel, to avoid busy waiting, we let the kernel schedule another kernel thread when waiting for a message. We made the scheduler kMVX-aware, allowing it to prioritize tasks with available sync messages. If the follower does not detect a divergence in the message, it sends the leader an acknowledgement message, allowing both variants to continue execution. If, however, the follower detects a divergence, it

RIGHTSLINK

sets the offending bytes to a zero-value, logs the divergence, and sends the modified buffer to the leader. The leader, finally copies either its own buffer in case of an acknowledgement or the modified buffer in case of a divergence.

In our experiments, zeroing the data does not cause any conflicts and allows us to prevent possible info leaks without halting the user-space application. Also note that zeroing the data may only cause a conflict affecting the user space application; the kernels do not diverge or crash due to a mismatch between the leader and follower.

I/O-intensive workloads might generate a lot of communication between variants, both for I/O- and syscall sync, potentially slowing down kMVX. Hence, it is vital to implement an efficient communication channel for our *sync points*. Since the messages for kMVX are synchronous in nature, we opted for an efficient communication layer based on a lockless shared-memory hash table. We also implemented a shared-memory allocator, to reduce the number of memory writes when transferring messages between the variants.

5.3 I/O sync

Kernel operations that interact with hardware devices cannot be performed by both kernels simultaneously. In cases such as *network I/O*, we let the leader communicate with the hardware, after which the result is *replayed* to the follower. This approach is comparable to replaying non-idempotent syscalls in existing MVX systems, but applied to operations inside the kernel. When the leader executes these replayed calls, it copies the results to a shared buffer from where the follower can read the result using a similar communication channel as *syscall sync*. Note that contrary to syscall sync, I/O sync does *not need to be synchronous*; the leader does not need to wait for an acknowledgement from the follower.

kMVX currently contains two different I/O sync modes: *low-level* and *high-level* sync. In the *low-level* sync component kMVX replays results of `in(b,w,l)`, `read(b,w,l,q)`, and `memcpy_fromio` calls, which are used to interact with hardware. This *low-level* mechanism is not used for all drivers, only the ones whitelisted by kMVX. By synchronizing these reads we maintain a consitent state for simple devices such as the real-time clock without special modifications. More complex device I/O, such as DMA, cannot be transparently supported currently, requiring modification in drivers to be kMVX-aware. For compatibility reasons, these kinds of I/O effects are replayed at a *higher* (i.e., subsystem) level. kMVX provides a wrapper mechanism which makes it easy to replay subsystem calls with a few lines of modifications.

5.3.1 Networking

Since the networking stack is highly state-dependent it requires heavy modifications to keep synchronized. For most of the networking, in our prototype, we opt to replicate the calls at a higher level, manually marshalling the effects of the syscall from the leader to the follower, while keeping a minimal shared state to reduce communication.

For example, to keep the result of the `select` call consistent across variants, we have to replay the resulting sets of the select call from the leader to the follower. Nevertheless, we leave most error handling up to the individual variants. For example checking for invalid file descriptors is left to the individual variants. This approach gives us a balance between full state-replication and just forwarding the results of the syscall from the leader to the follower.

As another example, in the `socket` syscall we synchronize the sets of open file descriptors between variants, mainly to keep the file descriptors consistent, but also to allow for error handling in the follower without communicating with the leader. When the `read` syscall is called on a socket file descriptor, the leader has to replay the data obtained from the lower levels of the network stack to the follower, thus requiring an I/O sync. On the other hand, if `read` is called on an invalid file descriptor, there is no need for an I/O sync, as both variants know about which file descriptors are valid.

In the future I/O sync can be changed to happen at a lower level in the networking stack, but this requires more complex logic to be kept consistent. Currently our prototype only supports the `select` API for network I/O multiplexing. As such, we do not support replaying of the `poll` and `epoll` interface, however, these interfaces could be added in a similar fashion to the `select` syscall.

5.3.2 Disk I/O

The current implementation runs different file systems for each replica. The leader kernel accesses the physical disk, while the replica uses a `tmpfs` file system. Using different drivers is a source of variance, enhancing security as the execution paths are very different. Since copy-to-user calls are implemented at a higher level (i.e., not in lower-level drivers) the output is still the same. For kMVX we implemented a generic *character device* that can replay an arbitrary device. For example, to eliminate divergence in the variants, we apply this character device to `/dev/(u)random`, replaying the value from the leader to the follower.

Furthermore, a number of syscalls may cause divergence between variants, due to nondeterministic behavior. By replaying these syscalls from the leader to the follower, we keep the variants consistent. For example syscalls which may return different values if the variants are even slightly out of sync, such as `gettimeofday`, are replayed from the leader to the follower, while syscalls that do not need to be replayed are executed locally on both variants. In fact, only a *small subset* of all syscalls are replayed. For example, we modify the `getpid` syscall to return a namespace-unique identifier that is the same across replicated processes in the variants, avoiding extra communication overhead. Our current prototype replays syscalls related to *time* (`time` and

RIGHTSLINK

`gettimeofday`) and *networking* (in total 15 syscalls were modified for replaying).

5.3.3 Nondeterminism

A source of nondeterminism in the kernel, besides hardware I/O, is kernel thread scheduling. Since the scheduling order inside the kernel is not visible to user space, it does not affect kMVX. The order of system calls in user space might, in the case of multithreaded applications, cause divergence. kMVX supports deterministic Pthread replication by forcing a global deterministic order on the system calls for a process. By using a modified Pthreads library (loaded using `LD_PRELOAD`) that overrides functions, such as `mutex_lock` and `mutex_trylock`, enabling deterministic ordering of syscalls through a special system call. We refer the reader to the FT-Linux paper [25] for a more thorough explanation of thread replication.

Likewise, interrupts are another source of nondeterminism. While these sources of nondeterminism affect the control-flow in the kernel [31], the interaction with user space remains unchanged. The only exception being signals. In order to keep the variants consistent, signals caused by the user (e.g., using the `kill` syscall) are delayed until after the next synchronization point. Likewise, signals originating from other processes are also delayed is a similar fashion.

Asynchronous signals originating from interrupts, such as `SIGINT` caused by the keyboard, are a source of randomness, as they can be caused at any moment in the execution. In our prototype we delay dispatching this signal until the next syscall, as to have a coherent state between variants. We then initiate the same signal on the follower using a callback function through an asynchronous messaging layer. Handling asynchronous signals in MVX is an open problem, with several solutions having been proposed [39, 40]. Our standard `put_user()` and `copy_to_user()` modifications check signal data copied to user space for consistency across variants. We had to modify `copy_siginfo_to_user()` to not synchronize pointer fields (which will obviously be different across variants due to our variation techniques).

6 Evaluation

6.1 Performance evaluation

Experimental setup We evaluate the performance of our kMVX prototype using a number of *micro* benchmarks as well as a number of real-world application *macro* benchmarks on a 4-core Intel i7-3770 with 16 GB of RAM. All benchmarks compare the performance of kMVX against a vanilla Linux 3.2.12 kernel. When running kMVX, we split the hardware into two partitions: each variant is limited to *2 cores* and *8 GB of RAM*. This partitioning accurately reflects how deploying kMVX on existing systems affects the performance without adding additional resources. The baseline uses *4 cores* and *16 GB of RAM*. We disable the vDSO for both the baseline and for the kMVX kernels. For the real-world server applications we benchmark the in-memory database redis-4.0.6 and three web servers: nginx-1.10.1, lighttpd-1.4.48, and a mongoose-6.10-based web server. We also benchmark the performance of multithreading in kMVX using the parallel compression utility *pbzip2*. Since our prototype implementation runs two kernels and two copies of the user space application, kMVX uses roughly twice as much memory as a vanilla Linux kernel.

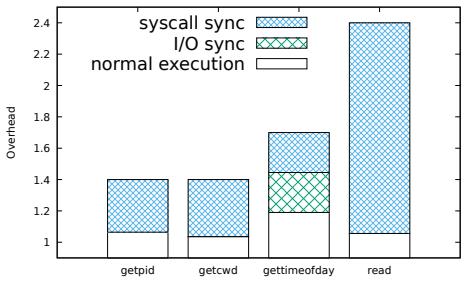

Figure 4. Microbenchmarks. Relative performance overhead of a selected number of syscalls. For the read syscall we read a 512 KB file.

	base	kMVX		UniSan	KASAN
	μs				
null call	0.04	0.06	(50.0%)	0%	5%
null I/O	0.08	0.14	(75.0%)	*	49%
stat	0.34	0.45	(32.4%)	2.7%	640%
open/ close	0.73	0.92	(26.0%)	-4.2%	1300%
select TCP	2.06	2.47	(19.9%)	0%	59%
signal install	0.11	0.15	(36.4%)	0%	10%
signal handle	0.74	2.41	(226%)	3.4%	311%
fork proc	67.1	101.31	(50.1%)	0%	299%
exec proc	204.0	275.11	(34.9%)	0.7%	163%
sh proc	483.0	737.60	(52.7%)	*	50%
pipe latency	1.77	2.05	(15.8%)	2.4%	41%
prot fault	0.208	†0.209	(0.4%)	2.4%	26%
TCP latency	12.0	21.10	(75.8%)	6%	250%
	MB/s				
Pipe bw	3005	2100	(30.1%)	0.2%	27%
TCP bw	3330	1641	(50.7%)	-0.1%	60%

Table 1. LMBench results. †At the moment our prototype does not accurately synchronize all signal handling events between the kernels. To prevent false positives, we disable kMVX sync for asynchronous signals. *Not reported in [26].

Microbenchmarks Figure 4 presents the overhead incurred by our kMVX prototype by comparing the median of the number of cycles taken per syscall measured from user space relative to a vanilla Linux kernel. Each of the selected syscalls is executed 10,000 times in a loop.

In Figure 4, the overhead of the microbenchmarks is split into three components: overhead of the normal execution of the syscall, the I/O sync, and the syscall sync. The syscall sync includes the overhead for communicating return values

RIGHTSLINK

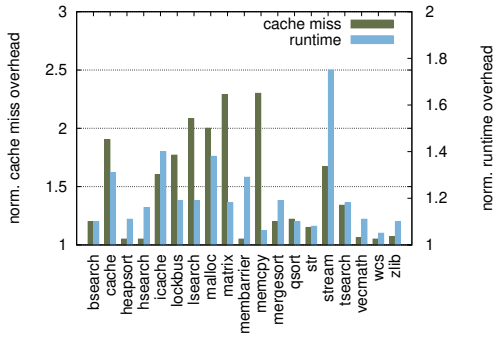

Figure 5. stress-ng. We measure the performance of kMVX on user space benchmarks making use of few syscalls. We show how cache performance is affected by kMVX. *stress-ng* is configured to use two workers.

to other variants, waiting for the other variant, and checking if they match, whereas I/O sync overhead is the cost of replaying certain logic in the follower.

As shown in Figure 4, simple syscalls such as **`getpid`** and **`getcwd`**, which copy a small amount of data to user space, incur a overhead of about 40%. The overhead mostly comes from *syscall sync* between kernels. While **`gettimeofday`** also copies a small amount of data, this syscall needs to be replayed from the leader to the follower (I/O sync), in order to keep the view of time between the variants consistent to prevent spurious divergences. On the other hand, the **`read`** syscall returns a large buffer to user space via copy-to-user. kMVX splits larger buffers into several cache-aligned blocks to optimize throughput. In Figure 4 we can see that the **`read`** system call has a higher overhead than other syscalls due to the larger buffers being copied for syscall sync.

We also benchmark kMVX using the LMBench suite of microbenchmarks, as presented in Table 1. Results show that the overhead is generally more prominent for I/O-intensive benchmarks. In general, small syscalls with a low duration have a higher overhead, since a relatively larger part of the syscall is spent on syscall sync. The select TCP benchmark has a lower overhead compared to other benchmarks due to the fact that it is replayed, thus the leader does not need to wait for an acknowledgment from the follower. Note that LMBench stresses the system calls (where a large part of the overhead for kMVX comes from). As such, the overhead is higher than for a majority of real-world applications, and shows the worst-case for kMVX.

Since running multiple kernels introduces more memory accesses, we also benchmark how kMVX affects cache hit/miss rates compared to a vanilla Linux kernel. For this purpose, we use the *stress-ng* [44] benchmarking suite to measure cache performance and to show how kMVX behaves for computationally and memory intensive applications (i.e., applications with relatively few syscalls).

As can be seen in Fig. 5, the cache miss rate is significantly higher for kMVX than for a vanilla Linux kernel. For applications that are computationally heavy, rather than memory heavy, we generally see a modest 22% runtime overhead. For more memory intensive applications, we see that the overhead is proportional to the increase in cache miss rates in most cases. For example, the **`cache`**, **`matrix`**, and **`stream`** benchmarks in the *stress-ng* suite are memory intensive, getting a higher cache miss rate when running on kMVX, degrading their runtime performance. Some benchmarks (such as **`icache`**, **`malloc`**, and **`membarrier`**) perform many syscalls, and as such kMVX takes an significant performance hit, as shown in earlier microbenchmarks.

Some benchmarks, such as **`zlib`**, **`wcs`**, **`vecmath`**, see a modest increase in cache misses. The data accessed by these benchmarks usually is available in L2, giving us a cache hit. Since L2 cache is partitioned per-core, and the cores are not shared between variants, we speculate that the L2 miss rate is not impacted as much in these kinds of benchmarks. On the other hand, the **`stream`** benchmark shows a significant higher cache miss rate. This benchmarks allocates buffers at least 4 times the size of the L2 cache, and repeatedly performs operations on these buffers. The **`stream`** benchmark, thus relies on L3 cache, which is shared by both kernels in our setup, showing a much higher cache miss rate than other benchmarks. Architectures with non-inclusive caches may show different characteristics for these benchmarks.

Macrobenchmarks To evaluate the real-world overhead of our kMVX prototype, we benchmark a number of server applications. Servers incur the most overhead from kMVX since they are particularly I/O-intensive, their primary bottleneck being networking or the disk. Syscalls like **`read`** occur frequently in such applications and require expensive checks on the copy-to-user calls, as shown by our microbenchmarks. Furthermore, network syscalls have to be replayed, adding more overhead for such applications.

Figure 6 shows the overheads for the *mongoose*, *nginx*, and *lighttpd* web servers for a varying number of concurrent connections. We benchmark kMVX using ApacheBench (with keepalive flag, loading a 512 byte page) over a 1 gigabit network connection (the client running a Intel i7-7800X with 24GB RAM), however, in this setup we could not saturate the system fully for some benchmarks. For example, as can be seen in Figure 6 when using a 1 gigabit network connection the overhead for *mongoose* becomes 3%, as the network request latency is the major bottleneck for the single-threaded server. For *nginx* and *lighttpd* we see an overhead of at most 27% on the gigabit connection.

To be able to fully saturate the system, we also benchmark using the loopback interface, achieving the highest possible throughput. In this setup ApacheBench itself only runs on the leader kernel, whereas the server application utilizes both variants. Since using the loopback interface eliminates

RIGHTSLINK

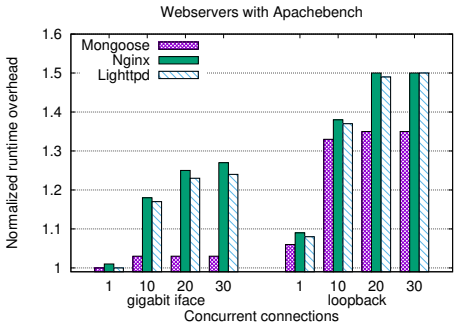

Figure 6. Web servers benchmarked with ApacheBench on a 1 gigabit link and the loopback interface. All web servers use the `select` API. Nginx has 1 worker process.

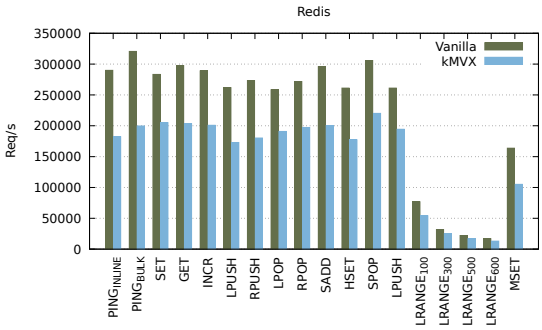

Figure 7. Redis. Geomean overhead of ~1.43x for the *redis* in-memory database using redis-benchmark.

any overhead from the kernel drivers this represents a worst-case scenario. The overhead via the loopback interface for *mongoose* is about 35% when saturated (i.e., more than 10 concurrent connections). For the better-performing servers, *nginx* and *lighttpd*, we see a higher overhead of about 50% when saturated (around 20 concurrent connections).

Similarly, the average runtime performance overhead is 43% when we stress the server with the `redis-benchmark` suite (Figure 7). In some parts of the suite, such as the `SET` and `LPOP` tests, the performance overhead is in the range of 26%. The relatively better performance for these tests can be explained by the fact that a larger part of these operations are performed in user space. The impact of syscall sync on the memory-intensive `SET` operation is relatively smaller than for other syscall-bound operations, such as `PING`.

Besides I/O heavy applications, such as the servers discussed earlier, we also show the performance of kMVX on the multithreaded *pbzip2*, which is computationally/ memory-heavy, rather than I/O-heavy. In this benchmark, we compare the data throughput of kMVX against vanilla Linux in a *multithreaded* file compression benchmark. *pbzip2* is an application that uses *Pthreads* for parallel file compression. pbzip2 uses one producer thread, reading the file from the

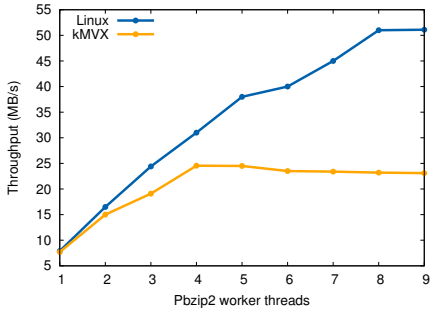

Figure 8. pbzip2. Benchmark of multithreaded bzip2 compression of a 1 GB file of random data using different number of worker threads, using the default block size of 900K. We show the throughput in MB/s of vanilla Linux and kMVX. Note: when kMVX is enabled, we only have 2 cores available.

disk; a number of *worker* threads to compress the file in parallel; and one consumer thread, combining the output from the worker threads into one file. In Fig. 8, we see that the throughput in kMVX when using 1 worker thread is about 11% lower than for a vanilla Linux kernel. A large part of this degradation is due to the overhead incurred by the syscall sync. In the case of *pbzip2*, there are no significant syscalls in the run of the program besides the `read`, `write` and `futex` syscalls. Enforcing a total order on the syscalls in the various threads is, thus, relatively cheap.

When partitioning the hardware for kMVX we give up half of the cores usually available to user space applications. As such, the performance with more than 4 worker threads is, naturally, lower for kMVX. If we consider only the performance up to 4 worker threads, the geometric mean of the run-time overhead is 16%. We note a slight decrease in performance when using more threads in kMVX, caused by the overhead incurred by keeping a global order on the system calls over the various threads.

Note that the overhead of 16% (when considering up to 4 workers) is significantly lower than the 30%+ observed for syscall-intensive applications, such as servers. Most of the time in *pbzip2* is spent in user space, making the syscall sync have less of an impact on the overall performance.

Comparison with related work Our prototype shows an overhead similar to that of traditional user space MVX systems. It is, however, a debatable comparison, since kMVX operates at the kernel level. Instead, we compare our prototype to other kernel defenses focusing on information leaks. For instance, KASAN and kmemcheck provide some degree of memory safety in the Linux kernel, but do not cover all classes of info leaks and have respectively 4x and multiple orders of magnitude overhead. Recent publications on mitigating uninitialized read vulnerabilities, UniSan [26] and SafeInit [28], are the closest to kMVX. Note that UniSan and SafeInit only prevent leaks for uninitialized data. kMVX

RIGHTSLINK

presents a more general defense against information leaks, both for uninitialized data and for other vulnerabilities, such as buffer overreads of initialized data. We would like to point out that the approach introduced with kMVX can also be applied to prevent other types of exploits besides information leaks. For example, MVX has previously been applied to preventing ROP-based attacks [50]. In this paper, our variation techniques are focused on preventing *information leaks*.

Existing solutions for preventing information leaks in the kernel, focusing only on uninitialized reads, have a very low performance overhead. SafeInit and UniSan have an average overhead of below 5% for typical server applications. Note that UniSan benchmarks their web server on a 1 gigabit connection. In the same setup kMVX has a overhead of at most 27%. As a comparison, we included the *LMBench* numbers from the UniSan paper in Table 1. While the UniSan numbers show much less overhead than kMVX, UniSan can only prevent uninitialized reads, which constitute only 60% of all info leaks [26]. kMVX provides *stronger security guarantees* by covering other classes of info leaks not detectable by previous defenses. On the other end of the spectrum, regarding memory safety in the kernel, we find KASAN [15]. KASAN offers detection of various memory errors, such as buffer overreads, use-after-free, use-after-return. Note that KASAN is intended as a debugging tool for memory errors, rather than a defense mechanism against information leaks, and as such the overhead is, understandingly, significant.

6.2 Security analysis

For our security analysis, we identify two primary targets that attackers want to leak from the kernel: kernel *pointers* for further attacks and *sensitive data* such as crypto keys. kMVX applies address space partitioning to its variants, meaning *every* pointer will differ between variants. When a valid kernel pointer is copied to user space, the corresponding pointer will be different in the other variant, leading to a *detected divergence*.

For other sensitive data, we have to consider where it is stored and leaked from. For instance, such data might be located on the kernel stack or heap, and in each case the guarantees depend on the variation in those areas.

For the *heap*, our design provides strong guarantees against both spatial and temporal memory error exploits. Because of the type-based SLAB allocator, both temporal and spatial attacks are already more limited [1, 48]. Having allocators with different behaviors and allocation patterns between variants is highly likely to eliminate the residual attack surface. For the stack, data can be leaked via temporal issues, such as an uninitialized read, or a spatial error such as a buffer overflow. The former is stopped by having different reuse patterns for each variant, using random stack frame padding. Spatial errors are stopped by the stack frame padding or the variable order randomization, causing data to be misaligned in both variants, triggering divergence and detection.

Type of leak	# CVEs			kMVX	UniSan	KASAN
use-after-free	15	0	0	15	0	15
uninitialized read	49	4	1	54	54	0
out-of-bounds read	1	1	0	1	0	1
other	8	0	10	12	0	0
Total	89			82	54	16
	H	S	O			

Table 2. Linux kernel 2017 info disclosure CVEs categorized by type and origin of leaked information. Info leak origin H: heap, S: stack, O: other.

To concretely demonstrate the effectiveness of kMVX, we analyzed all the Linux kernel information disclosure vulnerabilities (CVEs) from 2017 [33]. Similarly to prior work in the area [26, 35], we split the types of information leak vulnerabilities into four categories: (1) *uninitialized read*, (2) *use-after-free*, (3) *out-of-bounds read*, and (4) *other*. The *other* category contains vulnerabilities that leak information using unconventional means, either through a faulty check or by writing sensitive information to a location accessible by an unprivileged user (e.g., system logs). To show that our variation techniques are sufficient, we also identify the origin of the leaked data for each vulnerability: heap (*H*), stack (*S*), and other (*O*). Our analysis shows that our variation techniques are sufficient in preventing 92% of info leak CVEs published in 2017. An overview of the analysis of all these CVEs is published as part of the source code of kMVX.

To showcase how kMVX stops vulnerabilities, we randomly picked three CVEs concerning info leaks in the Linux kernel. *CVE-2016-4569* leaks information from the kernel *stack* due to *reading uninitialized data*. In our tests, we were able to reliably detect that information was leaked as our stack variation techniques between the variants make the diversified stacks leak different data. Similarly, due to the variations introduced by the different heap allocator implementations, *CVE-2013-2237*, which leaks information from the kernel *heap* using uninitialized reads, is also detected. In short, kMVX can reliably detect both heap- and stack-based uninitialized reads, as well as *out-of-bounds reads*, due to the diversified memory layouts introduced by our vargen.

In *CVE-2016-0728*, an attacker can exploit an *use-after-free* bug to forge an object at the location of a previously freed object. Since these objects are of different types, the type-based SLAB allocator in the follower places this object in a different bucket, leading to detectable divergent behavior between the variants.

Finally, we also prevent *other* possible leaks of kernel pointers that do not originate from memory errors, but instead are accidentally leaked due to programmer errors. For example, until recently, the `wchan` field of `/proc/PID/stat`

RIGHTSLINK

could be used to leak an absolute kernel address to unprivileged user space [29]. When an attacker reads such a file, kMVX will detect the divergence and zero out the differences between the variants, eliminating such info leaks.

The one out-of-bounds read vulnerability from the 2017 info leak CVEs (see Table 2) that kMVX does not detect is due to a remote attacker leaking information using a bug in a USB driver. As we focus on a local attacker in our threat model, we currently do not cover this vulnerability. The remaining 2017 CVEs that kMVX is not able to detect are either present in the *bootloader* (e.g., CVE-2017-0455) or are caused by timing side channels (e.g., CVE-2015-2877).

We have shown that kMVX can detect and prevent numerous types of vulnerabilities due to our selected variation techniques. By adding more variation techniques, other types of vulnerabilities can be thwarted in the future. To our knowledge, kMVX is the *first* defense to deterministically prevent any leakage of *kernel pointers* and provide strong (probabilistic) guarantees for preventing leakage of non-pointer data.

7 Related work

Kernel defenses Since Linux is a complex and integral part of the software stack, many kernel defenses have been developed over the years. For instance, kmemcheck [14] provides comprehensive memory safety (use-after-free, out-of-bounds, and uninitialized reads) but at the cost of orders of magnitude overhead. KASAN [15] focuses on only use-after-free and out-of-bounds bugs by applying ASan [42] to the kernel, but cannot detect all information leaks and still incurs 4x overhead. The use of annotations and static analysis with Coccinelle [16], Sparse [17], and GCC plugins [47] can detect potentially problematic code patterns statically. The PaX team introduced features to, for instance, force zero-initialization of `__user` structs (STRUCTLEAK), stack frame clearing during syscalls (STACKLEAK), harden `copy_to_user` (USERCOPY), and secure deallocation (SANITIZE) [46].

UniSan [26] and SafeInit [28] can prevent kernel info leaks due to reading uninitialized variables, by using static analysis and forcing initialization of variables that might leak to the user. Most of these approaches only address a single source of info leaks or incur significant runtime overhead, whereas our kMVX design addresses all info leaks at a much smaller cost. The SPLIT KERNEL [22] hardens the kernel with several (expensive) hardening techniques, but also maintains non-hardened copies of each function, allowing an untrusted process to run under the hardened kernel whereas trusted processes can still use the more efficient kernel functions.

MVX In 2006, Cox *et al.* [8] proposed a design, based on N-version programming [6], where variants were automatically generated, and where the monitoring and synchronization happens at the syscall level, laying the foundation of modern MVX research. Variant generation is based on the field of (automated) software diversity [24]. Proposed variant generation include random code insertion [9], system call randomization [7], instruction set tagging [8], address space partitioning [8], heap- and stack layout randomization [3, 40], and non-overlapping offset spaces [19]. Our variant generation for kMVX takes inspiration from these and implements effective and efficient techniques suited for the kernel.

Monitoring for user space MVX is done primarily through the syscall interface, with older solutions using more coarse-grained alternatives [3, 8]. Three approaches exist: in-kernel monitoring (high performance but intrusive) [8, 51], monitoring by an external application (for instance using the `ptrace` debugging facility of the kernel, which is safe but slow) [4, 18, 40, 52] and in-process monitoring, which is efficient but requires a careful design in security applications [12, 23, 36, 37, 51]. An exception here is Detile [10], which instead synchronizes at the byte-code instruction level to protect scripting engines. Our kMVX design also works at a different boundary than syscalls, requiring a new design. It can be compared with both the in-kernel and in-process monitoring, and can be considered more intrusive than traditional MVX monitors.

8 Conclusion

In this paper, we have presented kMVX—the first design that runs multiple diversified kernel variants simultaneously on the same machine to prevent information leaks. We have discussed a prototype implementation of kMVX, which provides efficient info-leak detection while preserving the Linux ABI, allowing us to run regular Linux binaries without changes or recompilation. We have demonstrated a number of variation techniques specific to kernels and identified the main obstacles in keeping the variants from diverging. We have also shown our prototype supports a number of popular server applications with a overhead of 20%–50%. On less I/O-intensive applications, such as *Pbzip2* and the *stress-ng* benchmark, the overhead is generally below 20%. The source code for kMVX is freely available at https://github.com/vusec/kmvx.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback. This work was supported by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 786669 (ReAct) and No. 825377 (UNICORE), by Cisco Systems, Inc. through grant #1138109, and by the Netherlands Organisation for Scientific Research through grants NWO 639.023.309 VICI "Dowsing" and NWO 639.021.753 VENI "PantaRhei". This paper reflects only the authors' view. The funding agencies are not responsible for any use that may be made of the information it contains.

RIGHTSLINK

References

[1] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *USENIX Security*.

[2] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*.

[3] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *PLDI*.

[4] D. Bruschi, L. Cavallaro, and A. Lanzi. 2007. Diversified Process Replicae for Defeating Memory Error Exploits. In *IPCCC*.

[5] 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 *ApSys*.

[6] Liming Chen and Algirdas Avizienis. 1978. N-Version programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *FTCS*.

[7] Monica Chew and Dawn Song. 2002. *Mitigating buffer overflows by operating system randomization.* Technical Report.

[8] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. 2006. N-variant Systems: A Secretless Framework for Security Through Diversity. In *USENIX Security*.

[9] S. Forrest, A. Somayaji, and D. Ackley. 1997. Building Diverse Computer Systems. In *HotOS*.

[10] Robert Gawlik, Philipp Koppe, Benjamin Kollenda, Andre Pawlowski, Behrad Garmany, and Thorsten Holz. 2016. Detile: Fine-Grained Information Leak Detection in Script Engines. In *DIMVA*.

[11] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *USENIX Security*.

[12] Petr Hosek and Cristian Cadar. 2015. VARAN the Unbelievable: An Efficient N-version Execution Framework. In *ASPLOS*.

[13] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking Kernel Isolation. In *USENIX Security*.

[14] The Linux Kernel. 2007. Getting started with kmemcheck. https://www.kernel.org/doc/Documentation/dev-tools/kmemcheck.rst. Accessed: 2018-01-22.

[15] The Linux Kernel. 2015. The Kernel Address Sanitizer (KASAN). https://www.kernel.org/doc/Documentation/dev-tools/kasan.rst. Accessed: 2018-01-22.

[16] The kernel development community. 2018. Coccinelle. https://static.lwn.net/kerneldoc/dev-tools/coccinelle.html. Accessed: 2018-01-22.

[17] The kernel development community. 2018. Sparse. https://static.lwn.net/kerneldoc/dev-tools/sparse.html. Accessed: 2018-01-22.

[18] Dohyeong Kim, Yonghwi Kwon, William N. Sumner, Xiangyu Zhang, and Dongyan Xu. 2015. Dual Execution for On the Fly Fine Grained Execution Comparison. In *ASPLOS*.

[19] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and Efficient Multi-Variant Execution Using Hardware-Assisted Process Virtualization. In *DSN*.

[20] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *EuroSys*.

[21] Greg Kroah-Hartman. 2018. The Linux Kernel Driver Interface. https://www.kernel.org/doc/Documentation/process/stable-api-nonsense.rst. Accessed: 2018-01-22.

[22] Anil Kurmus and Robby Zippel. 2014. A Tale of Two Kernels: Towards Ending Kernel Hardening Wars with Split Kernel. In *CCS*.

[23] Yonghwi Kwon, Dohyeong Kim, William Nick Sumner, Kyungtae Kim, Brendan Saltaformaggio, Xiangyu Zhang, and Dongyan Xu. 2016. LDX: Causality Inference by Lightweight Dual Execution. In *ASPLOS*.

[24] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *S&P*.

[25] Giuliano Losa, Antonio Barbalace, Yuzhong Wen, Marina Sadini, Ho-Ren Chuang, and Binoy Ravindran. 2017. Transparent Fault-Tolerance using Intra-Machine Full-Software-Stack Replication on Commodity Multicore Hardware. In *ICDCS*.

[26] Kangjie Lu, Chengyu Song, Taesoo Kim, and Wenke Lee. 2016. UniSan: Proactive Kernel Memory Initialization to Eliminate Data Leakages. In *CCS*.

[27] Kangjie Lu, Marie-Therese Walter, David Pfaff, Stefan Nürnberger, Wenke Lee, and Michael Backes. 2017. Unleashing Use-Before-Initialization Vulnerabilities in the Linux Kernel Using Targeted Stack Spraying. In *NDSS*.

[28] Alyssa Milburn, Herbert Bos, and Cristiano Giuffrida. 2017. SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities. In *NDSS*.

[29] Ingo Molnar. 2015. fs/proc, core/debug: Don't expose absolute kernel addresses via wchan. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=b2f73922d119686323f14fbbe46587f863852328. Accessed: 2018-01-30.

[30] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. 1990. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer* 23, 5 (May 1990).

[31] Peter Okech, Nicholas Mc Guire, Christof Fetzer, and William Okelo-Odongo. 2013. Investigating execution path non-determinism in the Linux kernel. In *OSADL*.

[32] Serkan Özkan. 2017. CVE Details Linux Kernel. https://www.cvedetails.com/product/47/Linux-Linux-Kernel.html?vendor_id=33. Accessed: 2018-01-22.

[33] Serkan Özkan. 2017. Linux Kernel : Security Vulnerabilities Published In 2017 (Gain Information). https://www.cvedetails.com/vulnerability-list.php?vendor_id=33&product_id=47&opginf=1&year=2017. Accessed: 2018-04-30.

[34] Jianfeng Pan, Guanglu Yan, and Xiaocao Fan. 2017. Digtool: A Virtualization-Based Framework for Detecting Kernel Vulnerabilities. In *USENIX Security*.

[35] S. Peiró, M. Muñoz, M. Masmano, and A. Crespo. 2014. Detecting Stack based kernel Information leaks. In *CISIS*.

[36] Luıs Pina, Anastasios Andronidis, Michael Hicks, and Cristian Cadar. 2019. MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution. In *ASPLOS*.

[37] Luıs Pina, Daniel Grumberg, Anastasios Andronidis, and Cristian Cadar. 2017. A DSL Approach to Reconcile Equivalent Divergent Program Executions. In *USENIX ATC*.

[38] 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 *EuroSys*.

[39] Babak Salamat. 2009. *Multi-Variant Execution: Run-Time Defense against Malicious Code Injection Attacks DISSERTATION.* Ph.D. Dissertation. University of California, Irvine.

[40] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 2009. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-space. In *EuroSys*.

[41] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. 2017. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security*.

[42] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*.

[43] Dannie M Stanley, Dongyan Xu, and Eugene H Spafford. 2013. Improved kernel security through memory layout randomization. In *IPCCC*.

RIGHTSLINK

[44] stress-ng team. 2018. stress-ng: a tool to load and stress a computer system. http://kernel.ubuntu.com/~cking/stress-ng/. Accessed: 2018-04-30.
[45] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*.
[46] The PaX Team. 2017. Grsecurity and PaX Configuration Options. https://en.wikibooks.org/wiki/Grsecurity/Appendix/Grsecurity_and_PaX_Configuration_Options. Accessed: 2018-01-22.
[47] The PaX Team. 2018. PaX - gcc plugins galore. https://pax.grsecurity.net/docs/PaXTeam-H2HC13-PaX-gcc-plugins.pdf. Accessed: 2018-01-22.
[48] Erik van der Kouwe, Taddeus Kroes, Chris Ouwehand, Herbert Bos, and Cristiano Giuffrida. 2018. Type-After-Type: Practical and Complete Type-Safe Memory Reuse. In *ACSAC*.
[49] Maxime Villard. 2017. Kernel address space layout randomization. https://blog.netbsd.org/tnf/entry/kernel_aslr_on_amd64. Accessed: 2018-02-05.
[50] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter. 2016. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE TDSC* 13, 4 (2016).
[51] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. 2016. Secure and Efficient Application Monitoring and Replication. In *USENIX ATC*.
[52] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. 2013. GHUMVEE: Efficient, Effective, and Flexible Replication. In *FPS*.
[53] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. 2015. From Collision To Exploitation: Unleashing Use-After-Free Vulnerabilities in Linux Kernel. In *CCS*.

RIGHTSLINK