Intra-Unikernel Isolation with Intel Memory Protection Keys

Mincheol Sung
Virginia Tech, USA
mincheol@vt.edu

Pierre Olivier*
The University of Manchester, United Kingdom
pierre.olivier@manchester.ac.uk

Stefan Lankes
RWTH Aachen University, Germany
slankes@eonerc.rwth-aachen.de

Binoy Ravindran
Virginia Tech, USA
binoy@vt.edu

Abstract

Unikernels are minimal, single-purpose virtual machines. This new operating system model promises numerous benefits within many application domains in terms of lightweightness, performance, and security. Although the isolation between unikernels is generally recognized as strong, there is no isolation within a unikernel itself. This is due to the use of a single, unprotected address space, a basic principle of unikernels that provide their lightweightness and performance benefits. In this paper, we propose a new design that brings memory isolation inside a unikernel instance while keeping a single address space. We leverage Intel's Memory Protection Key to do so without impacting the lightweightness and performance benefits of unikernels. We implement our isolation scheme within an existing unikernel written in Rust and use it to provide isolation between trusted and untrusted components: we isolate (1) safe kernel code from unsafe kernel code and (2) kernel code from user code. Evaluation shows that our system provides such isolation with very low performance overhead. Notably, the unikernel with our isolation exhibits only 0.6% slowdown on a set of macro-benchmarks.

CCS Concepts: • **Software and its engineering** → *Virtual machines*; *Operating systems*; *Memory management*; • **Security and privacy** → **Virtualization and security**; **Operating systems security**.

Keywords: Unikernels, Memory Protection Keys, Memory Safety

*Part of this work was done while Pierre Olivier was at Virginia Tech.

VEE '20, March 17, 2020, Lausanne, Switzerland

ACM Reference Format:
Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '20), March 17, 2020, Lausanne, Switzerland*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3381052.3381326

1 Introduction

Unikernels have gained attention in the academic research community, offering multiple benefits in terms of improved performance, increased security, reduced costs, etc. As a result, the application domains for these minimal, single-application virtual machines are plentiful, encompassing cloud- and edge-deployed micro-services/SaaS/FaaS-based software [6, 21, 22, 30, 44], server applications [22, 28, 29, 44, 53], NFV [12, 29–31], IoT [12, 15], HPC [25], efficient VM introspection/malware analysis [52], and regular desktop applications [36, 47].

One of the fundamental principles of unikernels is the elimination of the separation between kernel and user parts of the address space. Thus, a unikernel instance running an application alongside the unikernel's Library Operating System (LibOS) possess a *single and unprotected address space*. Because of this particularity, it is possible for unikernels to present interesting opportunities for performance benefits, such as replacing traditional system calls with regular function calls [11, 34] so that significant mode switch overhead [45] can be avoided. However, from a security perspective, a single and unprotected address space makes it so that the entire unikernel must be viewed as a single unit of trust. Subversion of a kernel or application component will result in the subversion of the entire unikernel with serious consequences, such as arbitrary code execution, critical data leaks or tampering, etc.

We argue that the current level of isolation provided by unikernels is too coarse-grained for many scenarios. First, a single application may be made of mutually-untrusting components [2, 49], such as if they came from different sources with variable security coding standards. Second, regarding the LibOS, although some are written in a memory-safe language [9, 24, 29, 51], they generally rely on untrusted

components for low level operations by using a traditional unsafe language [9, 29, 51] or by using `unsafe` code blocks for languages such as Rust. Other unikernel LibOSes are written entirely in an unsafe language [18, 19, 25]. Third, in scenarios where mutually untrusting components belonging to the same application need to be isolated [2, 16, 49], a computing base that is trusted from the tenant point of view has to be established to enforce that isolation. In the current state of the unikernel model, this Trusted Computing Base (TCB) cannot be the guest kernel as it is not itself isolated from the application. This implies falling back on the hypervisor to be this TCB, which is suboptimal from a performance standpoint.

Several isolation mechanisms have been proposed in the past to isolate an application's untrusting components. They operate at various levels: using hardware assisted virtualization [3, 20], running components in different processes/using different page tables [2, 23], or using ISA extensions such as Intel SGX [1, 7]. None of these techniques can be easily applied to unikernels without breaking the single address space principle and thus not only canceling the resulting performance benefits but also introducing non-negligible additional performance overheads in the form of switching costs [49].

In this paper, we propose addressing the aforementioned unikernel security issues by providing intra-unikernel isolation while maintaining the single address space feature of this OS model. To do so, we leverage the Intel Memory Protection Keys (MPK) [10] technology. MPK is a new hardware primitive that provides per-thread permission control over groups of pages in a single address space with negligible switching overhead [35, 49], making it a compelling candidate for use in unikernels.

We identify the various areas composing the address space of a unikernel, i.e. the kernel's safe/unsafe memory regions (static data section, stack, and heap), and the user memory regions (static data section, stack, and heap). Those areas are isolated from each other by using MPK-based mechanisms to enforce per-thread permissions on each memory area. Our design principles are: it should (1) preserve a *single address space*; (2) *isolate* the various areas of the address space; and (3) have *negligible cost.*

We demonstrate our techniques on RustyHermit [24], a unikernel written in Rust. This is done using an efficient isolation method for intra-unikernel components that relies on easy-to-use code annotations made by the unikernel LibOS programmer. On top of that mechanism we implement two isolation policies. First, we isolate safe from unsafe Rust kernel code so as to limit the possibilities for an attacker to exploit a vulnerability in the unsafe kernel code. Second, we re-introduce kernel and user space separation by isolating kernel from user code. This is a basic requirement to implement application components isolation mechanisms that should be enforced by the kernel. Our design allows protecting the trusted components of the kernel from attacks leveraging vulnerabilities in untrusted ones. We also protect kernel space from unauthorized access by subverted user code. Our isolation techniques have a low overhead; in particular, we still maintain the low system call latency feature of unikernels.

2 Background and Motivation

In this section, we first describe the motivation for intra-unikernel isolation. Next, we give some background information about the unikernel we work with in this paper, RustyHermit [24], as well as the programming language it is written in (Rust) and the Intel MPK technology we use to provide isolation.

2.1 Unikernel and Isolation

A unikernel [29] consists of a single application compiled and statically linked with a minimal kernel LibOS. Unikernels are *single purpose*, i.e. one instance corresponds to one guest VM running a single application on top of a hypervisor. A unikernel instance also presents a *single and unprotected address space* shared between the kernel and the application. All the code executes with the highest privilege level (for example, ring 0 in x86-64) and thus there is no memory protection between kernel and user code/data in that address space.

Such a model brings significant benefits in several domains [34], in particular in terms of performance [19, 25, 34]: due to the elimination of kernel/user separation, system calls can be replaced with regular function calls. This significantly reduces system call latency, as there is no longer a costly world switch between privilege levels [25]; expensive operations such as page table switching [34] are eliminated. As a result, unikernels have been shown to outperform traditional OSes in system intensive workloads [11].

However, the lack of isolation within a unikernel (intra-unikernel isolation) raises serious security concerns. Even if it executes a unique application, viewing a unikernel instance as a single and atomic unit of trust is too coarse-grained in current scenarios: a vulnerability in a relatively untrusted/vulnerable application component automatically leads to the attacker taking over the entire system. In a unikernel, this concern also include kernel components, as there is no isolation between kernel and user space. We divide intra-unikernel isolation issues into two categories: (1) the lack of isolation between kernel and user space and (2) the lack of isolation between trusted and untrusted kernel components in memory-safe unikernels.

Lack of Isolation between Kernel and User Space. Modern applications are made of components (such as libraries) having variable degrees of trustworthiness/potential for vulnerabilities, manipulating data with various levels of sensitivity [2]. Without isolation between these, taking over a

vulnerable component gives the attacker control over the entire application, including the sensitive data belonging to other components. Consider, for example, a formally verified cryptographic library [54] and a user-facing HTTP parsing module. The former is unlikely to contain vulnerabilities, but the sensitive data it manipulates (crypto keys) could be leaked through a vulnerability in the latter (such as CVE-2013-2028 in NGINX) when they run in the same application. Another example is an image manipulation library overwriting sensitive function pointers in the Global Offset Table [2]. To provide more security in these scenarios, intra-application solutions have been proposed [2, 16, 49]. They rely on a trusted entity to enforce an isolation policy. Due to the lack of user/kernel separation in unikernels, that entity cannot be the guest kernel as application code can freely access kernel memory. Although the hypervisor could play that role, it would be suboptimal from a performance point of view (more VMEXITs). It would also lead to an increase in the trusted computing base (hypervisor), which is a security concern. In conclusion, to support isolation of components within applications, *it is necessary to bring back user/kernel separation in unikernels.*

Lack of Isolation between Trusted and Untrusted Kernel Components. Several unikernels' OS layers are written in memory safe languages [9, 24, 29, 51]. This offers strong security guarantees compared to unikernels written in unsafe languages such C/C++ [18, 19, 25, 34]. However, even memory-safe unikernels rely on untrusted components to realize the low-level operations that are unavoidable in an OS context: the use of inline assembly and the need to dereference raw pointers. This is realized either with an unsafe language for those components [9, 29, 51] or with the use of unsafe code blocks [24] in a language such as Rust. Once again, without intra-unikernel isolation, a vulnerability in an unsafe kernel component leads to the subversion of the entire system, in effect negating the benefits of using a memory safe language.

In the rest of this paper we focus on RustyHermit [24], a unikernel written in Rust, although our design could easily be adapted to other unikernels that use C for low-level operations. One of the main reasons we chose RustyHermit for our implementation is the fact that, contrary to other memory-safe unikernels, it does not restrict the application code to the same language as the LibOS (such as OCaml for MirageOS, Erlang for LING, and Haskell for HaLVM), which is a significant compatibility advantage.

Listing 1 shows an unsafe code snippet extracted from RustyHermit's source code. These functions manage per-core variables using the GS x86-64 segment register, plus a relative offset depending on the variable. Examples of per-core variables are the CPUID, scheduling data structures, and task state segments. Practical addressing relative to the GS register can only be done using inline assembly, i.e. it

```
impl<T> PerCoreVariableMethods<T> {
    #[inline]
    default unsafe fn get(&self) -> T {
        let value: T;
        asm!("movq %gs:($1), $0"
            : "=r"(value) : "r"(self.offset())
            :: "volatile");
        return value;
    }
    #[inline]
    default unsafe fn set(&self, value: T) {
        asm!("movq $0, %gs:($1)"
            :: "r"(value), "r"(self.offset())
            :: "volatile");
    }
}
```

Listing 1. Per-core variable get/set methods.

should be placed within an unsafe code block. If we assume that, through a bug, the attacker has control over the `self` parameter, then the `set` function can be used to perform arbitrary memory writes (note that `self.offset()` returns a value deterministically computed from the value of `self`). Similarly, if we additionally assume that the attacker can exploit a bug to return the value of the `get` function, then it becomes an arbitrary memory read.

To conclude, in addition to kernel/user separation, there is also the need to bring *isolation between safe and unsafe kernel components* into memory-safe unikernels. Furthermore, *neither type of isolation should come at the cost of degraded performance, nor should they negate the performance benefits of unikernels* such as fast system calls, fast context switches, and the like.

In the rest of this section we give some background information on RustyHermit, kernel development in Rust, and the Intel MPK, the technology used in this paper to provide intra-unikernel isolation.

2.2 Rust

Rust is attracting attention as a system programming language because of the memory safety guarantees provided by its compiler. Furthermore, the absence of a garbage collector allows Rust to avoid much runtime overheads[13]. Instead of collecting unused memory in the runtime, Rust is designed to rely on comprehensive safety checking at compilation time; there are also runtime safety checks when the compile-time checks are not sufficient [24]. The concept of *ownership* ensures that all objects are safely handled with minimal runtime overhead. Thanks to Rust's memory safety and high performance, operating systems like RustyHermit [24], Theseus [5], TockOS [27] and Redox [14] were written in Rust.

Rust basically prohibits dereferencing raw pointers for memory safety. It is, however, inevitable for the kernel to access unchecked raw pointers, such as when accessing the page table. In some cases, the kernel has to call assembly,

such as when executing start-up code directly. To support those cases, Rust also provides an unsafe code region that is not checked by the Rust compiler or runtime. As Rust's memory safety is not guaranteed in the unsafe block by the compiler, developers have to write vulnerability-free codes by themselves.

2.3 RustyHermit

The unikernel RustyHermit is completely written in Rust and does not depend on any C code. One of Rust's major advantages for kernel developers is that it splits the runtime into an operating-system-independent library and an operating-system-dependent library. By implementing Rust's global memory allocator, the *alloc* library, multiple data structures become available and usable in kernel space. These include smart pointers as well as basic data structures like linked lists, binary heaps, ring buffers, and maps. Only a target specification file that specifies processor type, pointer width, etc. is required to compile these libraries. Consequently, kernel developers are able to reuse existing, well-tested code from the Rust community, which simplifies development and increases the robustness of the kernel.

Additionally, RustyHermit is a full 64-bit kernel, supporting x86-64 processors, SIMD instructions like AVX, thread local storage, and symmetric multiprocessing. RustyHermit is completely integrated into the Rust compiler infrastructure. One part of the Rust infrastructure is Cargo, which is Rust's package manager and coordinates the build process of Rust binaries. The main difference from the typical C/C++ build process is that the package manager does not install binaries, headers, static or shared libraries. It instead downloads the source code, compiles it with the same compiler flags, and links it directly to the executable. The Rust community calls such packages *crates*. By fully integrating RustyHermit into the Rust toolchain, cargo can be used to define the dependencies for the application. In principle, every published crate in a repository (e.g., crate.io) can be used to build executables based on the library operating system. The only requirement is that the crate must not directly call the host OS and bypass Rust's standard runtime.

Besides the support for pure Rust binaries, it is also possible to develop C/C++ applications on top of the Rust kernel. For this purpose, the C library *newlib* [32] is used to create the interface between C/C++ applications and the kernel.

In addition, RustyHermit comes with the lightweight hypervisor *uhyve*, which is also completely written in Rust and uses KVM to accelerate the virtualization. RustyHermit is able to delegate operating system services like filesystem access to the host system by hypercalls. The technique is outlined in [34]. RustyHermit is composed of about 20k LoC, including 650 lines of unsafe code [24].

2.4 Intel Memory Protection Keys (MPK)

Intel Memory Protection Keys is a new hardware feature providing per-thread permission control over groups of pages

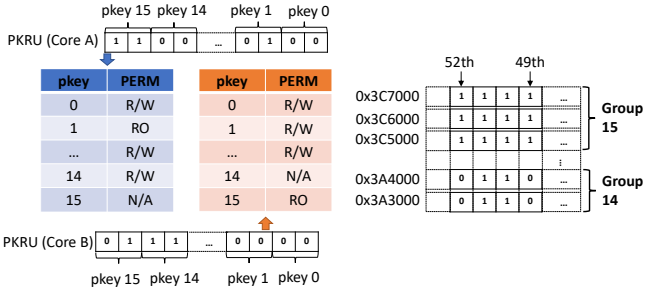


Figure 1. Intel Memory Protection Key.

without requiring modification of page tables at a small performance cost. Four previously-unused bits of each page table entry (the 62nd to the 59th on x86-64) are exploited by MPK [10, 35]. Since MPK exploits four bits of the page table entry, it supports up to 15 protection keys (we opted to reserve key 0).

MPK controls per-thread permission on groups of pages with the notation (WD, AD), where WD is *Write Disable* and AD is *Access Disable*. The possible states are are read/write (0,0), read-only (1,0), or no-access (x,1). Each core has a PKRU register (32 bits) containing a permission value. The value of the PKRU register defines the permission of the thread currently running on that core for each group of pages containing a protection key in their page table entries. Figure 1 illustrates MPK's operation. A thread running on a core *A* has the *no-access* permission on the pages of group 15 and *read-write* on those of group 14. On the other hand, a thread running on core *B can not access* the pages of group 14 and can *only read* the pages of group 15.

Unlike page-table-level permission, MPK provides thread-local memory permission. Furthermore, the cost of switching the PKRU value is quasi-negligible [49]. We believe MPK is most suitable for providing isolation within a unikernel without harming the principle of unikernels.

3 Assumptions and Threat Model

We define a unikernel application to be a collection of software components, i.e. pieces of code. These are compiled and linked together to form a unikernel binary, executed at runtime on top of a hypervisor in a VM representing a unikernel instance. The software components can either be trusted or untrusted. We assume no vulnerability in trusted components, which in practice denotes the use of a memory-safe language or verification techniques for these components. We assume that untrusted components can contain memory vulnerabilities such as buffer overflows, which can be exploited by an attacker aiming at hijacking the unikernel's control flow, leaking or tampering with sensitive data, etc.

We assume a unikernel model in which the LibOS is mainly implemented in a memory-safe language, examples of which include MirageOS [29], RustyHermit [24], LING [9], as well as HaLVM [51]. A unikernel is composed of application and

```
static mut KMSG: KmsgSection = KmsgSection {
  buffer: [0; KMSG_SIZE + 1],
};

pub fn kmsg_write_byte(byte: u8) {
  let index = BUFFER_INDEX.fetch_add(1, SeqCst);
  unsafe {
    let buffer = &mut KMSG.buffer[index % KMSG_SIZE];
    write_byte(buffer, byte);
  }
}
```

Listing 2. Example of unsafe kernel code.

kernel code. In this paper we aim to provide user/kernel separation so we simply see the entire application as an untrusted component, independently of application-specific characteristics such as the language it is written in or the level of skill of the application's programmer. In addition, we divide the kernel code into trusted and untrusted components. Trusted kernel components represent pieces of code written with a memory-safe language, i.e., offering strong security guarantees. Untrusted kernel components correspond to code written either in memory-unsafe languages [9, 29, 51] or in unsafe Rust code blocks [24]. To summarize, a unikernel is composed of (1) untrusted application code, (2) untrusted kernel components, and (3) trusted kernel components.

We assume that there is no vulnerability in the trusted kernel code, as memory safety is also guaranteed by Rust compiler. We trust the hardware to behave correctly and assume that there are no side channels.

4 Design of Intra-unikernel Isolation

This section goes through the design of our intra-unikernel isolation technique. We follow the design objectives: (1) preservation of a *single address space*, (2) isolation of various memory areas, and (3) *negligible cost*.

4.1 Data considered to isolate

We have a general security principle: *untrusted code should access only what it needs to operate correctly*. Listing 2 shows an example of unsafe kernel code in RustyHermit. The function `write_byte` in `kmsg_write_byte` stores the input `byte` on the `KMSG` buffer. As `write_byte` writes the input at the destination decided by a raw pointer, it should be called in an unsafe code block. In this example, `write_byte` accesses the `KMSG` buffer through the local variable `buffer`. Therefore, the call to `write_byte`, the buffer `KMSG`, and the variable `buffer` should all be isolated.

Kernel code is comprised of safe components and unsafe components. Isolating unsafe kernel functions and variables requires separate .data/.bss sections for static data, stacks for function calls, and heaps for dynamic memory allocation. Thus, we create an isolated data section, isolated stack, and isolated heap for the unsafe components. For the user/kernel isolation, we isolate all the sections of user memory by creating another isolated .data/.bss section, isolated stack, and isolated heap for the user application. Figure 2 shows the virtual address space layout of safe sections for the safe kernel components, isolated sections for the unsafe kernel components, and user sections for the user application.

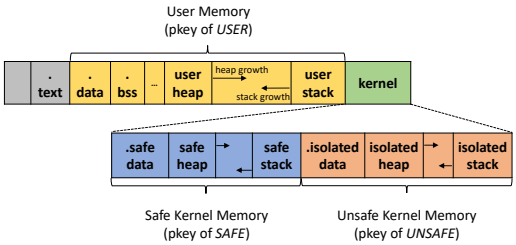


Figure 2. Virtual address space layout for intra-unikernel isolation.

4.2 Isolation with MPK

We leverage Intel MPK for intra-unikernel isolation. As previously described, MPK provides per-thread permissions for groups of pages according to their protection keys (pkeys). We set a pkey of `UNSAFE` on pages of the isolated data section, stack, and heap. On the other hand, pages for the safe kernel memory sections have a pkey of `SAFE` and pkey of `USER` for the user memory.

We switch the current thread's permission for the pkey `SAFE` to "*No Access*" right before calling an unsafe function. Right after the function returns, the permission is switched back to "*Read Write*" to end the isolation. An example in Listing 3 shows that the permission for the `SAFE` memory region is set to *No Access* before executing the unsafe kernel code (raw pointer dereference, unsafe function call, and inline assembly) by setting a value of `0b0...01100` (`SAFE` pkey is 1: 2nd, 3rd bits are set to 1s for *No Access*) in the PKRU register. After the function returns, the permission is set back to *Read-Write* by writing a value of `0b0...00000` to the PKRU. Therefore, MPK prohibits the thread from accessing the `SAFE` memory region (whose PTEs contain the pkey of `SAFE`) while executing the `UNSAFE` function. If a thread executing untrusted code (unsafe kernel or user application code) tries to access the safe memory region, a protection key page fault occurs and terminates process execution.

4.3 Unsafe Kernel Isolation

There are unsafe code blocks containing unsafe function calls, raw pointer dereference operations, and inline assembly in kernel code. Some of them need to access global variables or local variables in the stack frame of their caller function. Therefore, we create separate sections for static data isolation, a stack for unsafe function calls, and a heap for any dynamic memory allocation required by the unsafe functions. Those isolated memory regions are protected from the user application by MPK with the `UNSAFE` pkey.

```
unsafe {                    // pkey of SAFE is 1
    pkey_safe_NO_ACCESS();  // MPK_WRPKRU(0b0...01100)
    *ptr = some_data;       // Raw pointer dereference
    unsafe_function(ptr);   // Unsafe function call
    asm!("NOP"::::);        // Inline Assembly
    pkey_safe_READ_WRITE(); // MPK_WRPKRU(0b0...00000)
}
```

Listing 3. Example of isolating unsafe kernel code: raw pointer dereference, unsafe function call, and inline assembly. `MPK_WRPKRU` writes a value of 32bit on PKRU register.

Static Data Isolation. Unsafe functions in kernel may need to access global variables. We define global variables accessed by the unsafe kernel code as *unsafe global variables*. We place the unsafe global variables into a separate memory section (`.isolated_data` section in Figure 2). On the other hand, global variables that are only used by safe kernel code should be located in the safe data section, which unsafe kernel code is not able to access. We minimize the number of global variables in the unsafe data section by including only those needed, so a compromised thread in the unsafe kernel code can only access a very limited part of memory.

In the real kernel code, there are some global variables that are needed by both safe and unsafe code. We also put those shared global variables in the unsafe data section. As our objective is minimizing the number of global variables accessed by the unsafe code, all the rest of global variables are protected by the unsafe code. Although having a separate .bss section for uninitialized global variables is useful to reduce the size of binary, we keep the variables in the data section to ease design complexity while still attaining the reasonably small size of a unikernel.

Stack Isolation. An unsafe function should not share its function call stack frame with a safe function. We create a separate stack isolated by MPK pkey for unsafe functions, shown as `.isolated_stack` in Figure 2. When an unsafe function is called, we switch the value of the stack pointer register (`%rsp` in `x86-64`) with the address of the isolated stack.

By default, an unsafe function is strictly isolated, so it is unable to access the safe stack frames. In real kernel code, however, an unsafe function may try to access its caller's stack frames through local variables. If the caller is a safe function, the access should be managed carefully. In this case, we only allow accesses to the shared stack frame between the safe caller and the unsafe callee, meaning the unsafe callee function is still not able to access the rest of the caller's stack frames.

Heap Isolation. An isolated heap is required for unsafe code to allocate memory dynamically. We create a separate heap (isolated heap in Figure 2) and a memory allocation function (`unsafe_allocate`) for it. The `unsafe_allocate` function assigns available virtual and physical addresses and maps them while writing the pkey of `UNSAFE` to the corresponding page table entries. Consequently, a thread with inaccessible permissions for the safe memory region can still access the memory allocated by the `unsafe_allocate` function while executing the unsafe code.

4.4 User Application Isolation

The entire user part of the address space is assumed to be untrusted. For that reason, we separate the entire memory of the application from the kernel memory as the traditional monolithic kernel model does. However, separation is done by MPK, for which the domain switch operation, a simple update of the PKRU value, is much faster than traditional user/kernel separation methods involving costly world switch interrupts. Consequently, it fundamentally follows the main principle of unikernels: *a single address space*.

As the entire user application is treated as a set of untrusted components, all the memory sections are separated: .data/.bss, user stack, and user heap (Figure 2). A thread running a user application code should not be able to access either kernel memory regions, safe or unsafe. The border between user and kernel is quite distinct: a thread enters the kernel when system calls are called and exits the kernel when the system calls return.

User application memory also comprises user static data (.data, .bss, etc.), user stack, and user heap like those of the kernel. We can reuse most of the design choices used for the safe/unsafe kernel isolation.

5 Implementation

We implement a prototype on top of RustyHermit to demonstrate our techniques. We can leverage Rust's features such as Rust Macros [38] to provide developers with a convenient way to use our isolation mechanism on the existing kernel source code.

5.1 Protection Keys and MPK Permission

Isolating safe/unsafe and kernel/user memory requires two MPK protection keys. The protection key of 1 is used for the safe kernel memory region permission, while 2 is used for the unsafe kernel memory regions. As the user application is the most untrusted component, it is not protected by any protection key. Table 1 summarizes PKRU values that determine permissions for the groups of pages by the protection keys. A thread running with a PKRU value of `0x00` is the most trusted entity at that point. However, when the thread executes an unsafe kernel code block, its PKRU is set to contain `0xC` (`0b0000_1100`). This PKRU value prohibits the thread from accessing the group of pages of pkey 1, which corresponds to the safe memory regions. In the same way, `0x3C` (`0b0011_1100`) in the PKRU register prevents the thread from accessing both safe (pkey 1) and unsafe (pkey 2) kernel memory regions, providing the isolation of kernel from user memory.

Table 1. PKRU values for memory regions: when a thread executes each code, PKRU is set to the corresponding value. For example, before a thread executes the user code, PKRU is set to contain `0x3C` (`No Access` on both safe and unsafe kernel memory regions) such that access to kernel memory by that thread is prohibited.

Memory Region	Unused 26 bits (pkey 3 ~15)	UNSAFE (pkey 2)	SAFE (pkey 1)	Reserved (pkey 0)	Hex Value
Kernel (safe)	0b00000000000000000000000000	00	00	00	0x00
Kernel (unsafe)	0b00000000000000000000000000	00	11	00	0xC
User	0b00000000000000000000000000	11	11	00	0x3C

5.2 Unsafe Kernel Isolation

Rust unsafe code [39] provides additional features such as raw pointer dereferences, inline assembly, Rust intrinsic functions, and unsafe function calls, as well as the use of static mutable global variables. As the Rust compiler does not guarantee memory safety in the unsafe code blocks, kernel developers should carefully use unsafe code at their own risk. However, all unsafe code can contain potential memory vulnerabilities. Accessing a static mutable global variable, for example, may expose a data race, but does not have memory vulnerabilities.

Rust Macro. Rust macros provide a handy way of reusing multiple lines of code [23]. As explained in Sections 4.2, 4.3, and 4.4, there are several steps involved in safe/unsafe and kernel/user isolation of global and local variables, and functions. All the procedures can be packed into an easy-to-use macro for better *programmability*. Listing 4 provides an example of a macro that isolates an unsafe function introduced in Listing 2. Macro `isolated_function` wraps the unsafe function call and expands to multiple steps that isolate the function. For a global variable accessed by the unsafe function, macro `unsafe_global_var` locates the global variable in the isolated data section.

Isolated Kernel Data Section. We wrote a linker script to specify the isolated data section (labeled `.unsafe_section`) at a certain address. When RustyHermit boots, the pkey of `UNSAFE` is set for the corresponding page table entries of the section. To allocate global variables in the `.isolated_data` section, we leverage Rust's attribute (`#[link_section]`) to dedicate variables to that specific section [37]. To ease use of that attribute, we provide the `unsafe_global_var` macro, which wraps the definition of a global variable with the `#[link_section]` attribute. Developers should explicitly wrap the definition of a global variable that is accessed by unsafe kernel code with the `unsafe_global_var` macro. Listing 4 shows how the global variable `KMSG` is wrapped with the `unsafe_global_var` macro (at line #2). The macro adds the attribute `#[link_section]` before the definition of the target global variable (at line #21).

Isolated Kernel Stack. We create a separate stack with the protection key of `UNSAFE` apart from the stack for safe kernel functions. This isolated stack is used when calling an unsafe kernel function such as `write_byte` in Figure 2. Switching the stack pointer for the unsafe function to use the isolated stack frame can be done by switching the value of `%rsp` register by inline assembly. We provide a macro (`isolate_function`) to expand lines of inline assembly because isolating an unsafe function requires: (1) saving the current stack pointer; (2) switching the stack pointer to the isolated stack; (3) changing MPK permission to *No Access* on the safe kernel memory; (4) calling the unsafe function; (5) restoring the MPK permission to *Read Write* on the safe memory; and (6) restoring the stack pointer to the safe stack.

It only works, however, for an unsafe function that does not need to access its caller's stack frame. Some functions get references to local variables of the caller as function parameters and access them. To cover this case, we also provide a macro (`isolate_function_weak`) with extra steps for sharing the caller's stack frame. The macro that disallows accessing the caller's stack frame is, by contrast, named `isolate_function_strong`. It is also possible that an unsafe function needs to access data in a frame of one of the caller functions (e.g., caller's caller and so on). We provide `share` and `unshare` macros for making local variables in the remote stack frames accessible/inaccessible to the unsafe function.

Placing annotations represents some effort on the programmer side. However, we consider it to be relatively low: in our effort to isolate RustyHermit safe/unsafe code and user/kernel space, less than 2% of the codebase was touched. It is also straightforward: a simple keyword to place. Finally, that process is guided: any overlooked variable will be identified at runtime with a MPK fault.

Isolated Kernel Heap. We create an isolated heap for unsafe functions to allocate memory dynamically. Instead of implementing a new memory allocation function for the isolated heap, we reuse the existing allocation function for the safe kernel heap. The memory allocation function maps a virtual-physical address by writing the physical address and page flags to the corresponding page table entries. The unsafe allocation function additionally sets a protection key of unsafe on the page table entries.

Raw Pointer Accesses, Inline Assembly. Dereferencing raw pointers and using inline assembly allows access to arbitrary locations in memory, so such techniques should be isolated in a way that does not change the stack. We thus

```
/********* Macro usage example *********/
unsafe_global_var!(
static mut KMSG: KmsgSection = KmsgSection {
  buffer: [0; KMSG_SIZE + 1],});

unsafe fn write_byte<T>(buffer: *mut T, byte: T) {
  volatile_store(buffer, byte);
}

pub fn kmsg_write_byte(byte: u8) {
  let index = BUFFER_INDEX.fetch_add(1, SeqCst);
  unsafe {
    let buffer = &mut KMSG.buffer[index % KMSG_SIZE];
    isolate_function!(write_byte(buffer, byte));
  }
}

/********* Macro definition below *********/
macro_rule! unsafe_global_var! {
(static $name:ident: $var_type:ty = $val:expr) => {
    #[link_section = ".unsafe_data"]
    static $name: $var_type = $val;
    };
}

macro_rule! isolate_function {
($f:ident($($x:tt)*)) => {{
    asm!("mov %rsp, $0;" // Store stack pointer
         "mov $1, %rsp;" // Switch to isolated stack
         "mov $2, %eax;" // N/A perm on SAFE memory
         "xor %ecx, %ecx;"
         "xor %edx, %edx;"
         "wrpkru;"       // Write %eax on PKRU
         "lfence"
    : "=r"(current_rsp)
    : "r"(isolated_stack),"r"(UNSAFE_PERMISSION)
    : "eax", "ecx", "edx" : "volatile");

    $f($($x)*);          // Actual function call

    asm!("mov $0, %eax;" // R/W perm on SAFE memory
         "xor %ecx, %ecx;"
         "xor %edx, %edx;"
         "wrpkru;"
         "lfence;"
         "mov $1, %rsp" // Restore stack pointer
    :: "r"(SAFE_PERMISSION),"r"(current_rsp)
    : "eax", "ecx", "edx" : "volatile");
}};
}
```

Listing 4. Isolation of unsafe kernel code using Rust macros. Usage example of the macros in the kernel code and the definitions of the macros.

implemented two macros for developers: `isolation_start` and `isolation_end`. The first macro, `isolation_start`, is used to indicate that the isolation starts, so it switches the MPK permission to *No Access* on the safe memory regions. The other one, `isolation_end`, is used to indicate the end of isolation, and it restores the MPK permission to *Read-Write*. Kernel developers should add `isolation_start` before a raw pointer deference or inline assembly to start isolation and `isolation_end` after them to finish isolation.

Non-isolated Function. There is a small amount of unsafe kernel code that cannot be isolated by our techniques. For example, the spinlock code has a few unsafe functions that are used by both safe and unsafe kernel code. Introducing isolation on the functions may cause deadlock. Functions such as `lgdt` or `load_cs` also cannot be isolated because they are called early in the boot process. We also do not isolate x86 I/O port instructions such as `in` and `out` because these functions manipulate device memory. Functions such as `wrmsr` and `rdmsr` are not isolated because they access machine-specific registers. It is worth noting that all of these unprotected unsafe code blocks are very small, most representing just a few instructions and extremely unlikely to represent vulnerabilities.

5.3 Copy between Safe/Unsafe Kernel Code

RustyHermit requires BIOS and boot loader data to be located in a fixed memory address. Accessing this data is done by unsafe functions because it is accessed via a raw pointer, and this data should also be isolated. However, we cannot apply our isolation mechanism to it, since RustyHermit stores it at a fixed address. To protect the data, we provide a copy mechanism. When a thread accesses the data (e.g., an eight-byte variable in a data structure), only eight bytes are copied to a per-core memory buffer (`unsafe_storage`). The thread then accesses `unsafe_storage` through an unsafe function. If the thread writes new data to `unsafe_storage`, it should be synced so it is copied back to the original data structure. These operations are protected by threads concurrently running on the other cores. This is because `unsafe_storage` is restricted to that core only by using `%gs`-relative addressing (i.e., each core contains a different base address in the `%gs` register).

The memory copy function is itself unsafe because it requires raw pointers for source and destination. We maintain a whitelist of memory addresses to limit arbitrary memory access by the copy function.

In addition to the unikernel-specific areas, per-core data is accessed by the copy mechanism. The per-core data is accessed by the unsafe functions (we introduce `get` and `set` methods presented in Listing 1), so it should be isolated. It is not suitable to locate the per-core data in the isolated data section because per-core data contains important data such as a pointer to the scheduler.

5.4 User Application Isolation

Isolating the user memory region is simpler than the unsafe kernel isolation because the application does not share global

```
pub extern "C" fn sys_rand() -> u32 {
    return kernel_function!(__sys_rand());
}
```

Listing 5. A system call calling an internal function wrapped by the `kernel_function` macro.

variables with the kernel. In consequence, their border is distinct and the MPK permissions should only be switched for system calls.

System Calls. System calls are the gate between user and kernel so MPK permissions and stack should be switched before making a system call and after returning from it. To avoid modifying the Rust standard library, we modified the definition of system call. Each system call calls internal calls (e.g., `sys_rand` calls `__sys_rand` in Listing 5) and the internal function is wrapped with a `kernel_function` macro.

What the `kernel_function` macro does is similar to the `isolate_function` for unsafe kernel isolation. It expands into a few lines of inline assembly, and switches the MPK permission and the stack pointer to the user stack.

6 Security Evaluation

Unikernels such as RustyHermit are still an emerging technology and are not widely used in production. It was thus difficult to find known vulnerabilities we could use to validate our unikernel isolation scheme. As a result, we provide unikernel applications with handcrafted attack scenarios and demonstrate that our isolation technique successfully thwarts those attacks. We present 2 scenarios, respectively demonstrating (1) user vs. kernel space isolation and (2) safe and unsafe kernel code isolation.

User vs. Kernel Space Isolation. In this scenario, we assume the application is external-facing (a web server, for example) and contains a memory corruption-based vulnerability that a remote attacker uses to perform arbitrary memory reads/writes. Examples are CVE-2013-2028 for NGINX and CVE-2014-0226 for Apache. In an unprotected unikernel, due to the lack of user/kernel isolation, the attacker would then be able to use the vulnerability to freely tamper with or leak sensitive kernel data. This could be used to break security mechanisms enforced by the kernel, such as Address Space Layout Randomization (ASLR).

We reproduced this scenario by writing a simple unikernel application that accesses the kernel data segment. In an unprotected unikernel, an attacker could freely read and write kernel data. Our user application isolation scheme can prohibit this attack. As the user application is running with the MPK permission USER, which disallows to access the kernel memory (including the kernel data section). When the write operation is issued, an MPK fault occurs and unikernel execution is terminated. Our system also displays some information about the fault, such as the instruction pointer at the time and the faulty address, in order to help a system administrator investigate the attack.

Unsafe Kernel Isolation. In this scenario, we assume that an attacker is able to hijack the control flow of the unikernel application and divert it to trigger the execution of buggy unsafe kernel code through a system call. Depending on the vulnerability in the kernel code, the attacker could then tamper/leak kernel data, escalate privileges, execute arbitrary code, etc. Examples of vulnerable kernel code called through system calls with specific parameters are numerous, with specific examples being CVE-2013-1763 and CVE-2016-10229.

We reproduced such a scenario by assuming an attacker is able to manipulate the parameters of the per-core kernel variable access methods presented in Listing 1. This would give the attacker arbitrary memory read/write capabilities. The safe/unsafe kernel isolation method we implemented prevents malicious calls to `set`/`get` methods from accessing memory that is not allowed, i.e. the majority of kernel memory. When the unsafe kernel code tries to access the inaccessible memory regions, an MPK fault terminates unikernel execution and provides the instruction pointer at that point as well as the faulty address.

Discussion: Other Attack Scenarios. An attack scenario against our system would be unsafe code tampering with the PKRU. A possible mitigation against such an attack would be to use binary analysis/rewriting to validate/sanitize any use of the WRPKRU instruction, as done in ERIM [49]. An attacker could also try to bypass such checks by using Return Oriented Programming (ROP) to jump to code snippets manipulating the PKRU. A classical mitigation used in all modern systems against ROP is ASLR. Although they are currently not implemented in RustyHermit, both static analysis and ASLR can be integrated without any runtime overhead.

There is also a possibility of information leaks or data-oriented attacks due to unused registers not being saved and scrubbed upon safe/unsafe code switches. We chose not to do so for performance reasons, as it is certain that saving and restoring registers, for example with the `xsaveopt` instruction, will increase the domain switch latency.

7 Performance Evaluation

We conducted a performance evaluation to demonstrate our design principles: providing isolation with minimal overhead. The objective of the performance evaluation is to answer the following questions: First, *what are the overheads of switching across isolated safe and unsafe kernel code and across isolated kernel/user code?* Second, *what is the performance impact of such isolation on real applications?* Third, *how does our scheme perform in a multi-threaded environment?* We chose vanilla RustyHermit as a baseline and compare our prototype against it. Our experimental setup has an Intel Xeon Silver 4110 CPU (2.10GHz, eight physical cores) with 64KB of L1 cache, 1024KB of L2 cache per core, and 11MB of L3 cache. The setup has 192GB of main memory and runs Ubuntu

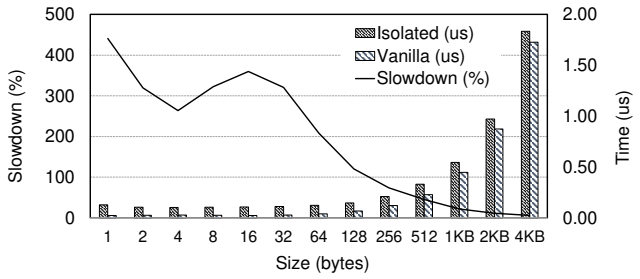


Figure 3. Cost of isolated `write_bytes` call.

18.04 with Linux 4.15 (needed for MPK support). Rust's cargo version is 1.40.0.

7.1 Unsafe Kernel Isolation

In this section, we evaluate the unsafe kernel isolation. We aim to measure the overhead of calling an unsafe kernel function isolated by our techniques. This is because isolating unsafe kernel functions may contain the possible overhead (e.g., MPK permission switching, stack switching, data copying) compared to vanilla ones. We chose examples of some unsafe kernel functions and implemented a micro-benchmark to measure the time cost of the isolated unsafe kernel functions.

Write_bytes. `write_bytes` is an unsafe function writing `byte` to an arbitrary address. We isolate `write_bytes` with `isolate_function_strong` macros and call it one million times, then calculate the time cost of a single function call. The result, presented in Figure 3, contains the total cost of the unsafe function call, composed of: switching the kernel stack and the MPK permission, the actual function call, and restoring the stack and the MPK permission. We change the write size from 1 byte to 4KB. For each size, we iterate one million times and calculate the slowdown caused by the unsafe function isolation. With small writes, the isolated `write_bytes` is four times slower than the vanilla one. This is because the majority of the overhead comes from our isolation mechanism. However, as the write size increases, the cost of calling `write_bytes` dominates the overall cost and the isolation overhead becomes negligible. In particular, our prototype introduces a 6% slowdown when writing 4KB at a time.

Per-core Variable Get and Set Methods. We also evaluated the cost of the `core_id` and `set_core_scheduler` functions to measure the per-core variable get and set methods (`Percore.get` and `Percore.set`). Introduced in Figure 1, `Percore.get` and `Percore.set` could be used as attack vectors to gain arbitrary memory read/write capabilities. Their usage as potential attack vectors means they should be isolated. In addition, they are invoked by kernel functions such as `core_id` and `set_core_scheduler`, which are frequently called in the kernel code. This makes them appealing as candidates for our unsafe kernel isolation as well. To do this, we created a micro-benchmark that iteratively calls `core_id` to invoke `Percore.get` and `set_core_scheduler` to invoke `Percore.set`. We measure the time cost of a hundred million calls, calculate the cost of one function call, and compare the isolated one to the vanilla one. Table 2 shows the results of the experiment. First, we observe the performance difference between `Percore.get` and `Percore.set` on both the isolated and the vanilla benchmarks. `set_core_scheduler` generally costs more than `core_id` because memory reads are faster than writes. When comparing the isolated functions to the vanilla ones, the isolated functions take longer due to the cost of memory copies introduced by the copy mechanism (Section 5.3): it introduces additional memory copy overhead besides the unsafe kernel isolation overhead (MPK permission switching, stack switching). `Percore.get/set` copies the original per-core values to the unsafe memory region, which is followed by the unsafe read/write operation (Listing 1) being performed on the unsafe memory region. Finally, the updated data is copied back to the original per-core data location. This additional overhead explains the performance degradation for the isolated `Percore.get/set` methods.

Table 2. PerCoreVariable `get` and `set` methods called by `core_id` and `set_core_scheduler`, respectively.

Caller function	Unsafe function	Cost (µs)	
		Isolated	Vanilla
`core_id`	`Percore.get`	0.202	0.017
`set_core_scheduler`	`Percore.set`	0.367	0.020

7.2 User Application Isolation

We evaluated user application isolation by measuring the cost of system calls, as they are a bridge between kernel and user space. To do so, we implemented micro-benchmarks written in both Rust and C and compare them. They exhibit null calls and `getpid` calls, the latter involving data copying. In addition to vanilla RustyHermit, we also evaluated system calls in Linux running on KVM. For Linux-KVM, we tested on an Ubuntu 17.10 distribution using Linux 4.13. We compiled all of the code with optimization level 3.

Null System Call. We evaluated a null system call to measure the pure system call latency. This call does nothing other than return, allowing us to measure the pure overhead of our user application isolation mechanism. For Linux, we use the `getpid` system call. We call this null system call a hundred million times and calculate the average cost for one function call. Note that we disabled vDSO for Linux in order to avoid potential user-mode system calls. Figure 4A represents the cost of the null system call in the Rust and C applications. The isolated null system call in the Rust application takes 0.19 µs while the vanilla one takes 0.002 µs. This difference comes from the user application isolation mechanism that we provide. The vanilla system call only has the overhead of function call. However, the isolated system call introduces: (1) accessing the `Task` structure through the per-core scheduler (which can be accessed by `Percore.get` and

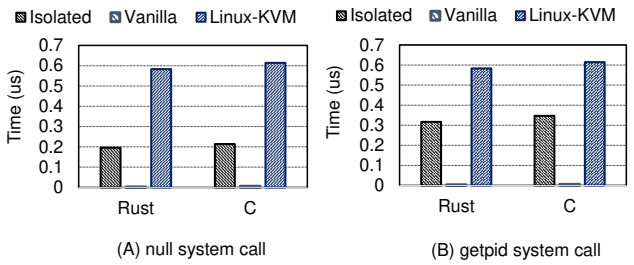
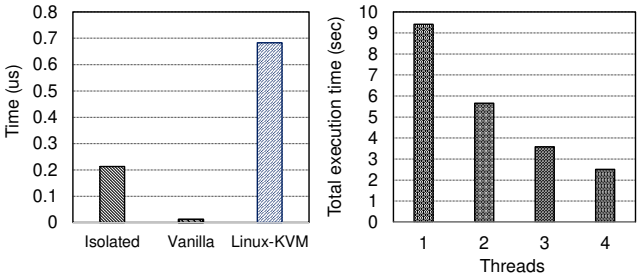


Figure 4. System call evaluation.



(a) Time cost of sbrk **(b)** Multi-threaded getpid

Figure 5. Evaluation of sbrk and multi-threaded getpid.

also introduces the overhead mentioned in Section 7.1) to get the user stack address, (2) switching the MPK permission and stack pointer. Furthermore, the compiler loses optimization possibilities due to the use of the macros that we provide. Nonetheless, the system call isolated by the user application isolation mechanism is approximately three times faster than `getpid` on Linux (0.58 µs). This demonstrates that we can provide isolation while still maintaining the low system call latency feature of unikernels.

In the C application, all of the system call results are a bit slower (the isolated system call takes 0.21 µs, the vanilla version takes 0.005 µs, and the Linux version takes 0.61 µs). The user application isolation overhead still dominates the overall cost of the system call and reduces compiler optimization possibilities.

Getpid. This function is provided for user applications and invokes the `sys_getpid` system call. `sys_getpid` also contains unsafe/safe switches and the copy mechanism used for the per-core data. Thus, the cost of the `getpid` function can represent the overall overhead of the user application isolation mechanism. As in the null system call experiment, we set a micro-benchmark to make the call a hundred million times and calculate the average cost for one function call. Figure 4B presents the results of `getpid` on our prototype, vanilla RustyHermit, and Linux. We tested both Rust and C applications.

In all cases, the system call from the Rust application outperforms that of the C application, as with the null system calls. In addition, the cost gap between Rust and C is similar to that for the null system call. The memory copy overhead is the main factor in the performance degradation of our prototype, as the PID of the task is stored in the `Task` structure that is referenced by the `current` pointer in the per-core scheduler. Accessing the per-core scheduler is performed via `Percore.get`, which introduces the additional memory copy.

With our scheme, the `getpid` system call is still 2x faster than it is on Linux, demonstrating that our technique preserves unikernel benefits.

Sbrk. We measured `sbrk` (only used by C applications) latency for evaluation of the user application isolation. We call `sbrk` with a parameter of 16 (an increment of 16). `sbrk` calls `sys_sbrk`, which does not include expensive per-core variable methods such as `Percore.get` and `Percore.get`. However, our user application isolation introduces the overhead of the MPK switch and the stack switch. Despite this, `sbrk` with our user application isolation still outperforms the Linux one significantly, as shown in Figure 5a.

Multi-threading. To demonstrate that our intra-unikernel isolation method works in multi-threaded environments, we created a Rust benchmark launching up to 8 threads and parallelizing an iteration of ten million `getpid` calls. We could observe that our intra-unikernel worked with multi-threading and scaled with the number of threads (Figure 5b).

7.3 Real Applications

To measure the overall performance impact of our system, we evaluated our prototype with macro-benchmarks. We used memory/compute intensive benchmarks from various suites including NPB [41], PARSEC [4], and Phoenix [40].[1] The results are shown in Figure 6, illustrating that the average slowdown imposed by the intra-unikernel isolation compared with the vanilla unikernel is only 0.6%.

We also counted the number of unsafe/safe switches and user/kernel switches and summarize them in Table 3. Remember that one unsafe function call corresponds to two unsafe/safe switches (from safe to unsafe switch on entry and unsafe to safe switch on return) and one system call corresponds to two user/kernel switches. Especially, phoenix-pca has a total of 27,246 switches and switches at a rate of 1,238 per second, which is system intensive. The evaluation demonstrates that our system introduces negligible performance overhead for real applications.

8 Related Works

Unikernels. Since their invention in 2013 [29], unikernels have grown in popularity in academia. These single-purpose, minimal VMs offer benefits in addition to the already mentioned performance gains. They are lightweight [43], offering subsecond boot time and very low disk/memory footprints. This is due to the simplicity of unikernel LibOSes and the fact that in a unikernel instance, the kernel embeds only what

[1] Note that some applications from these suites are not supported due to the limited compatibility of RustyHermit.

Table 3. Number of unsafe/safe switches and user/kernel switches invoked by benchmarks.

Benchmark	Unsafe/safe switches	User/kernel switches
npb-cg	5218	272
npb-is	4294	106
npb-ep	4370	116
npb-mg	4606	158
phoenix-kmeans	6882	1580
phoenix-pca	19402	7844
whetstone	3758	14
dhrystone	3734	10
livermore	13118	1574
linpack	3878	38

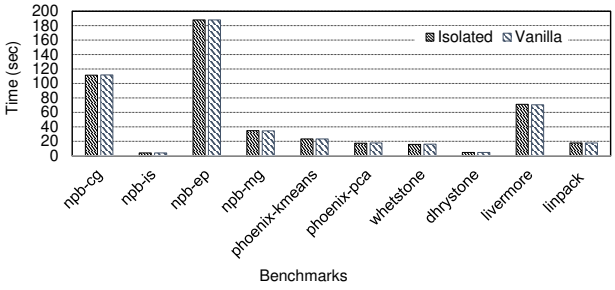


Figure 6. Macro benchmarks.

is needed for the application it runs. Lower footprints translate into cost reductions for the cloud tenant and superior consolidation (increased revenue) for the provider. Fast boot times make unikernels good candidates for scale-out/elastic deployments [33]. The potential application domains for unikernels are plentiful, as listed in the introduction.

The isolation between unikernel instances running on a host is strong as they are virtual machines, and they are considered superior to containers [30] in that regard. However, in this paper we show that the lack of intra-unikernel isolation is a security issue and addressed that concern. To our knowledge, we are the first to propose an intra-unikernel isolation system.

The performance benefits of unikernels come at least partially from the sharing of a single and unprotected address space [19, 25, 34]. That concept was originally pioneered by single-address-space OSes that appeared in the 90s following the appearance of 64-bit virtual addressing, such as Opal [8] or Nemesis [26]. We demonstrated that using a lightweight isolation mechanism such as MPK can bring security benefits while keeping a low latency for system calls.

Although some unikernels such as Rumprun [18], OSv [19], and HermitCore/HermiTux [25, 34] are entirely written in unsafe languages (C/C++), others use memory safe languages. These include MirageOS [29] written in OCaml, LING [9] in Erlang, HaLVM [51] in Haskell, and RustyHermit [24] in Rust. However, even those rely on memory unsafe languages or unsafe code blocks to implement the low level operations that an OS needs to support. Using our isolation scheme, we show that the safe part of the kernel can be isolated from the unsafe regions.

Software Components Isolation. Beyond the traditional user/kernel split, the decomposition of software into trusted and untrusted components has been studied in several past works, at the application [2, 23, 49] and OS [46–48, 50] levels. LibOSes such as Graphene [47, 48] adopt the Exokernel OS model and bring as many kernel components as possible in user space, reducing the size of the interface with the kernel for more isolation. In VPFS [50], the filesystem service is split between two isolated components, a small and trusted computing base performing security-critical operations and an untrusted code base reusing most of the code of an existing legacy file system. In Proxos [46], the system call interface is partitioned into trusted and untrusted operations. Configuration rules allow routing the application's system calls either to a trusted micro kernel or to an untrusted commodity OS. Occlum [42] runs a LibOS within an Intel SGX enclave and offers isolation for multiple tasks inside that enclave by leveraging Intel MPX [17] (deprecated in recent Intel CPUs).

Among the fine-grained isolation works focusing on the application level [2, 23, 49], SandCrust is relatively close to our work. It isolates safe from unsafe Rust code by running unsafe code in a separate process, which is not doable in a unikernel without breaking the single address space principle. To our knowledge, we are the first to apply fine-grained isolation to unikernels. Due to the peculiarities of this OS model, we face specific challenges such, as the need to keep a single address space to preserve a low system call latency and the need to reintroduce user/kernel space isolation.

9 Conclusion

The lack of intra-unikernel isolation is a serious security concern. We designed an isolation scheme for components within a unikernel instance. Relying on the Intel MPK technology allows us to keep the single address space feature of unikernels and thus maintain their performance benefits. We demonstrated an overhead as low as 0.6% for macro-benchmarks. The code is available online at the following URL: https://ssrg-vt.github.io/libhermitMPK.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Nadav Amit, for their insightful comments. Special thanks goes to Chungha Sung, Joshua Bockenek, and other colleagues for their feedback. This work was supported in part by the US Office of Naval Research under grants N00014-18-1-2022, N00014-16-1-2104, and N00014-16-1-2711.

References

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan

O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, Berkeley, CA, USA, 689–703. http://dl.acm.org/citation.cfm?id=3026877.3026930

[2] Julian Bangert, Sergey Bratus, Rebecca Shapiro, Michael E. Locasto, Jason Reeves, Sean W. Smith, and Anna Shubina. 2013. *ELFbac: Using the Loader Format for Intent-Level Semantics and Fine-Grained Protection*. Technical Report TR2013-727. Dartmouth College, Computer Science, Hanover, NH. http://www.cs.dartmouth.edu/reports/TR2013-727.pdf

[3] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. USENIX, Hollywood, CA, 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 72–81.

[5] Kevin Boos and Lin Zhong. 2017. Theseus: A State Spill-free Operating System. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) *(PLOS'17)*. ACM, New York, NY, USA, 29–35. https://doi.org/10.1145/3144555.3144560

[6] Alfred Bratterud, Alf-Andre Walla, Hårek Haugerud, Paal E Engelstad, and Kyrre Begnum. 2015. IncludeOS: A minimal, resource efficient unikernel for cloud services. In *Proceedings of the 7th IEEE International Conference on Cloud Computing Technology and Science (CloudCom 2015)*. IEEE, 250–257.

[7] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. Securekeeper: confidential zookeeper using intel sgx. In *Proceedings of the 17th International Middleware Conference*. ACM, 14.

[8] Jeff Chase, Hank Levy, Miche Baker-Harvey, and Eld Lazowska. 1992. Opal: a single address space system for 64-bit architecture address space. In *[1992] Proceedings Third Workshop on Workstation Operating Systems*. IEEE, 80–85.

[9] Cloudozer LLP. 2017. LING/Erlang on Xen website. http://erlangonxen.org/. Online, accessed 11/20/2017.

[10] Jonathan Corbet. 2015. Memory protection keys. *Linux Weekly News* (2015). https://lwn.net/Articles/643797/.

[11] Glauber Costa and Don Marti. 2014. Redis On OSv. http://blog.osv.io/blog/2014/08/14/redis-memonly/.

[12] Vittorio Cozzolino, Aaron Yi Ding, and Jörg Ott. 2017. FADES: Fine-Grained Edge Offloading with Unikernels. In *Proceedings of the Workshop on Hot Topics in Container Networking and Networked Systems (HotConNet'17)*. ACM, 36–41.

[13] Cody Cutler, M Frans Kaashoek, and Robert T Morris. 2018. The benefits and costs of writing a {POSIX} kernel in a high-level language. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. 89–105.

[14] Developers. 2019. Redox - Your Next(Gen) OS. https://www.redox-os.org.

[15] Bob Duncan, Andreas Happe, and Alfred Bratterud. 2016. Enterprise IoT security and scalability: how unikernels can improve the status Quo. In *IEEE/ACM 9th International Conference on Utility and Cloud Computing (UUC 2016)*. IEEE, 292–297.

[16] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*. 489–504.

[17] Intel. 2013. Introduction to Intel(R) Memory Protection Extensions. https://software.intel.com/en-us/articles/introduction-to-intel-memory-protection-extensions.

[18] Antti Kantee and Justin Cormack. 2014. Rump Kernels No OS? No Problem! *USENIX; login: magazine* (2014).

[19] Avi Kivity, Dor Laor Glauber Costa, and Pekka Enberg. 2014. OS v - Optimizing the Operating System for Virtual Machines. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*. 61.

[20] Koen Koning, Herbert Bos, and Cristiano Giuffrida. 2016. Secure and efficient multi-variant execution using hardware-assisted process virtualization. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 431–442.

[21] Michał Król and Ioannis Psaras. 2017. NFaaS: named function as a service. In *Proceedings of the 4th ACM Conference on Information-Centric Networking*. ACM, 134–144.

[22] Simon Kuenzer, Anton Ivanov, Filipe Manco, Jose Mendes, Yuri Volchkov, Florian Schmidt, Kenichi Yasukata, Michio Honda, and Felipe Huici. 2017. Unikernels Everywhere: The Case for Elastic CDNs. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'17)*. ACM, 15–29.

[23] Benjamin Lamowski, Carsten Weinhold, Adam Lackorzynski, and Hermann Härtig. 2017. Sandcrust: Automatic Sandboxing of Unsafe Components in Rust. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems* (Shanghai, China) *(PLOS'17)*. ACM, New York, NY, USA, 51–57. https://doi.org/10.1145/3144555.3144562

[24] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019. Exploring Rust for Unikernel Development. In *Proceedings of the 10th Workshop on Programming Languages and Operating Systems* (Huntsville, ON, Canada) *(PLOS'19)*. ACM, New York, NY, USA, 8–15. https://doi.org/10.1145/3365137.3365395

[25] Stefan Lankes, Simon Pickartz, and Jens Breitbart. 2016. HermitCore: a unikernel for extreme scale computing. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS 2016)*. ACM.

[26] Ian M. Leslie, Derek McAuley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. 1996. The design and implementation of an operating system to support distributed multimedia applications. *IEEE journal on selected areas in communications* 14, 7 (1996), 1280–1297.

[27] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 234–251. https://doi.org/10.1145/3132747.3132786

[28] Anil Madhavapeddy, Thomas Leonard, Magnus Skjegstad, Thomas Gazagnaire, David Sheets, David J Scott, Richard Mortier, Amir Chaudhry, Balraj Singh, Jon Ludlam, et al. 2015. Jitsu: Just-In-Time Summoning of Unikernels.. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 559–573.

[29] A Madhavapeddy, R Mortier, C Rotsos, DJ Scott, B Singh, T Gazagnaire, S Smith, S Hand, and J Crowcroft. 2013. Unikernels: library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. ACM, 461–472.

[30] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) Than Your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. ACM, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[31] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation* (Seattle, WA) *(NSDI'14)*. USENIX Association, Berkeley, CA, USA, 459–473. http://dl.acm.org/citation.cfm?id=2616448.2616491

[32] Newlib 2017. Newlib Website. https://sourceware.org/newlib/. Online, accessed 12/12/2017.

[33] Vlad Nitu, Pierre Olivier, Alain Tchana, Daniel Chiba, Antonio Barbalace, Daniel Hagimont, and Binoy Ravindran. 2017. Swift Birth and Quick Death: Enabling Fast Parallel Guest Boot and Destruction in the Xen Hypervisor. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (Xi'an, China) *(VEE '17)*. ACM, New York, NY, USA, 1–14. https://doi.org/10.1145/3050748.3050758

[34] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. 2019. A Binary-Compatible Unikernel. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*.

[35] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC19)*. 241–254.

[36] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Newport Beach, California, USA) *(ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1950365.1950399

[37] The Rust Project. 2019. Application Binary Interface - The Rust Reference. https://doc.rust-lang.org/reference/abi.html#the-link_section-attribute.

[38] The Rust Project. 2019. Macros - The Rust Programming Language. https://doc.rust-lang.org/1.29.0/book/first-edition/macros.html.

[39] The Rust Project. 2019. unsafe Rust - The Rust Programming Language. https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html.

[40] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*. Ieee, 13–24.

[41] Sangmin Seo, Gangwon Jo, and Jaejin Lee. 2011. Performance characterization of the NAS Parallel Benchmarks in OpenCL. In *IEEE International Symposium on Workload Characterization (IISWC 2011)*. IEEE, 137–148.

[42] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, and Yubin Xia. 2020. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[43] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. 2019. X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*.

[44] Giuseppe Siracusano, Roberto Bifulco, Simon Kuenzer, Stefano Salsano, Nicola Blefari Melazzi, and Felipe Huici. 2016. On the Fly TCP Acceleration with Miniproxy. In *Proceedings of the 2016 Workshop on Hot topics in Middleboxes and Network Function Virtualization (HotMiddlebox 2016)*. ACM, 44–49.

[45] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-less System Calls. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, Berkeley, CA, USA, 33–46. http://dl.acm.org/citation.cfm?id=1924943.1924946

[46] Richard Ta-Min, Lionel Litty, and David Lie. 2006. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 279–292.

[47] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys'14)*. ACM, 9.

[48] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A practical library OS for unmodified applications on SGX. In *Proceedings of the USENIX Annual Technical Conference (ATC 2017)*. 8.

[49] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Memory Protection Keys. *USENIX Security Symposium* (2019).

[50] Carsten Weinhold and Hermann Härtig. 2008. VPFS: Building a virtual private file system with a small trusted computing base. In *ACM SIGOPS Operating Systems Review*, Vol. 42. ACM, 81–93.

[51] Adam Wick. 2012. The HaLVM: A Simple Platform for Simple Platforms. Xen Summit.

[52] Xen Website. 2018. Google Summer of Code Project, TinyVMI: Porting LibVMI to Mini-OS. https://blog.xenproject.org/2018/09/05/tinyvmi-porting-libvmi-to-mini-os-on-xen-project-hypervisor/, Online, accessed 10/30/2018.

[53] Yiming Zhang, Jon Crowcroft, Dongsheng Li, Chengfen Zhang, Huiba Li, Yaozheng Wang, Kai Yu, Yongqiang Xiong, and Guihai Chen. 2018. KylinX: A Dynamic Library Operating System for Simplified and Efficient Cloud Virtualization. In *Proceedings of the 2018 USENIX Annual Technical Conference*.

[54] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A verified modern cryptographic library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 1789–1806.