RESEARCH

Open Access

# Iso-UniK: lightweight multi-process unikernel through memory protection keys

Guanyu Li, Dong Du and Yubin Xia*

**Abstract**

Unikernel, specializing a minimalistic *libOS* with an application, is an attractive design for cloud computing. However, the Achilles' heel of unikernel is the lack of multi-process support, which makes it less flexible and applicable. Many applications rely on the process abstraction to isolate different components. For example, Apache with the multi-processing module isolates a request handler in a process to guarantee security. Prior art tackles the problem by simulating multi-process with multiple unikernels, which is incompatible with existing cloud providers and also introduces high overhead. This paper proposes Iso-UniK, a new unikernel design enabling multi-task applications with the support of both functionality and isolation. Iso-UniK leverages a recent hardware feature, named Intel Memory Protection Key (Intel MPK), to provide lightweight and efficient isolation for multi-process in unikernel. Our design has three benefits compared with previous approaches. First, Iso-UniK does not need hypervisor support and is thus compatible with existing cloud computing platforms; second, Iso-UniK promises fast system calls with only 45 cycles; last, a process can be isolated with a flexible configuration. We have implemented a prototype based on OSv, a unikernel system supporting unmodified applications. Iso-UniK can achieve fast *fork* operation with only 66$\mu$s for multi-process applications. Our evaluation shows that the isolation and multi-process support in Iso-UniK will not damage the applications' performance.

**Keywords:** Unikernel, Multi-process, Intel MPK, Isolation

## Introduction

Virtualization is the base of widely-used cloud computing. The abstraction of virtual machine supports unmodified OS and applications, but also introduces limitations, including low resource utilization (Amit and Wei 2018) and unfair scheduling (Kashyap et al. 2018). It has been proposed to leverage unikernel, a virtualized library operating system (libOS), to mitigate these issues. With specialization and single address space, unikernels promise performance improvement as well as smaller trusted computing base (TCB).

However, the lack of multi-process support in unikernel makes it less flexible and applicable for many applications. It has been identified as the main roadblock towards a widespread use of unikernels (Zhang et al. 2018; Tsai et al. 2014). Multi-process applications usually leverage the process isolation to protect sensitive data. For example, Apache (Apache http server project) web server can allow a worker process to handle the request from a remote client. Without the process isolation, memory disclosure bugs like HeartBleed (The heartbleed bug) can leak secrets in the memory. However, supporting the isolation using protection mode like Linux usually causes unacceptable overhead to the unikernel applications. Previous work like Kylinx (Zhang et al. 2018) and Graphene (Tsai et al. 2014) uses multiple instances of unikernel to simulate multi-process and implements inter-process communication (IPC) among these instances with the help of hypervisor or host OS. However, such solution may not scale well and may lead to higher overhead of performance, as shown in Fig. 1.

*Correspondence: xiayubin@sjtu.edu.cn
The institute of parallel and distributed systems (IPADS), Shanghai Jiao Tong University, Dongchuan Road, Shanghai, China

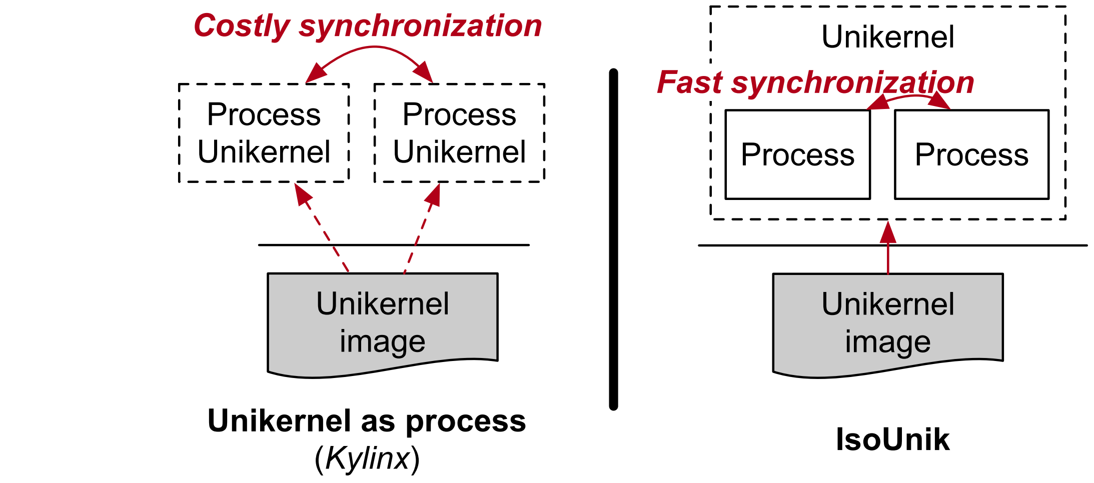

**Fig. 1** Design comparision. Prior solution uses multi-unikernel to simulate multiple processes, which leads to higher overhead

In the paper, we propose Iso-UniK, a multi-process unikernel design with lightweight isolation. We leverage a recent hardware feature, Memory Protection Keys (MPK) (Intel® 64 and IA-32 architectures software developer's manual) from Intel, to provide lightweight protection between kernel and user applications. As we know, this is the first work to leverage MPK for isolation in kernel mode. The design of Iso-UniK is based on two novel techniques: *reverse priority isolation* and *two gates protection*. Further, the design is general and does not depends on the MPK feature.

We have implemented a prototype based on OSv (Kivity et al. 2014), an open-sourced unikernel systems supporting unmodified applications. We implement the multi-process support, and apply our lightweight in-kernel protection to ensure the isolation between different processes. Our evaluation shows that Iso-UniK introduces negligible performance overhead, about 90 cycles slower for system calls compared with OSv.

The main contributions of this paper are as follows:

- A detailed analysis of the requirements for multi-task compatibility in unikernels.
- A general design of Iso-UniK that provides secure multi-task support for unikernels.
- An implementation of Iso-UniK based on a state-of-the-art unikernel system, OSv.
- An evaluation with both micro-benchmarks and real-world applications demonstrating the efficiency and practicability of Iso-UniK.

The rest of the paper is structured as follows. We study the necessity of multi-process support to motivate our design ("Background and motivation" section). Afterward, we introduce the design of multi-process in the unikernel and how we integrate the lightweight isolation into the unikernel design ("Design" section). We present our design of in-kernel isolation using different hardware protection methods ("Kernel data in user-space for data protection" and "Inner kernel design for privileged code" sections). The prototype implementation is based on OSv ("Implementation" section). Last, we evaluate the performance of Iso-UniK ("Evaluation" section), discuss related work ("Related work" section) and conclude our work ("Conclusion" section).

## Background and motivation

### Background

Researchers have proposed several lightweight virtualization systems (Belay et al. 2012; Manco et al. 2017; Madhavapeddy et al. 2013; Kivity et al. 2014) to provide both strong isolation and good performance (resource utilization). The basic idea of the lightweight virtualization systems is to reduce unnecessary software layers in the virtualized environment and eliminate the semantic gaps between host kernel and virtualized applications.

Process-abstraction virtualization systems like Dune (Belay et al. 2012) and gVisor (gVisor) are the extreme cases of the idea, where the host OS provides the system call interfaces to the virtualized processes. Typically, a process-abstraction virtualization system runs an unmodified application in user mode of the guest, while an intercept kernel (e.g., libDune in Dune and Sentry in gVisor) runs in kernel mode of the guest. The intercept kernel can handle system calls or redirect them to the host kernel. Process abstraction for the guest environment can help the host OS manage the resource more efficiently, and provides defense-in-depth security benefit (as in gVisor). Meanwhile, by supporting OCI (Open Container Initiative) specification, process-abstraction virtualization systems can be easily integrated with frameworks like

Docker and Kubernetes, and thus can be used by existing cloud platforms.

However, process-abstraction virtualization systems share the same attack surface, Linux syscall interfaces, and is highly dependent on the host OS. Unikernel is another solution for lightweight virtualization. LightVM (Manco et al. 2017) leverages the small memory footprint of unikernel to achieve fast instantiation. Unikernels like Mirage (Madhavapeddy et al. 2013) and OSv (Kivity et al. 2014) provide better performance by specializing the operating system for applications and removing the isolation between kernel and applications. Compared with other virtualization system design, unikernel systems enjoy the benefit of flexible resource management, as they have a minimal memory footprint and the applications can customize the kernel (in the guest) behaviors.

### Multi-process support is necessary

In this section, we emphasize the necessity of supporting multi-process in unikernels. We have two specific reasons; first, many legacy applications rely on multi-process support for compatibility; second, applications rely on the multi-process isolation for its security.

**Multi-process for Compatibility.** One important reason for using multi-process instead of multi-thread is to be compatible with non-thread-safe libraries. Take Apache as an example. Apache provides Multi-Processing Module (MPM) (Apache MPM prefork) to implement a non-threaded, pre-forking multi-process web server. MPM guarantees the correctness of the sites that using non-thread-safe libraries, specifical libraries with non-reentrant functions.

**Multi-process for Security.** Cloud servers are usually deployed with applications that touch critical secrets, like the cryptographic keys for SSH connection, the personal privacy data and even the passwords of users. However, such secrets can be disclosed due to memory disclosure vulnerabilities in the application. HeartBleed (CVE-2014-0160) (The heartbleed bug), as one of the most notorious vulnerabilities, can be leveraged by attackers to read up to 64KB memory data in a process. This has already lead to practical attacks like stealing private keys and session keys of a cloud server. Although many sandbox approaches (Yee et al. 2009; Sehr et al. 2010; Liu et al. 2015) have been proposed, multi-process isolation provided by the kernel is still the most widely used and efficient approach.

### Isolation for unikernel

With the significance of supporting multi-process in unikernels, researchers have already proposed designs to meet the need. Graphene (Tsai et al. 2014) treats an instance of unikernel as a process. Graphene runs instances on host Linux within a picoprocess, and implements multi-process feature with the help of Linux. The cost of this solution is that the isolation is not as good as virtual machine based on hypervisor. Kylinx (Zhang et al. 2018) uses multiple virtual machines to implements multi-process abstraction.

These proposed unikernel designs are not practical for commercial cloud platforms. For example, instantiating multiple virtual machines in the AWS (AWS) to simulate a multi-process applications costs a lot.

### Intel MPK

Intel MPK (Memory Protection Keys) (Intel® 64 and IA-32 architectures software developer's manual) is a keys-based permission control for memory isolation of userspace. Each page can be set to group with the group number in the page table, from 0 to 15. The register, PKRU, contains two AD/WD bits for each group, and AD bit is for access to the page while WD bit is for write permission to the page. PKRU provides a thread-local control and can be modified with the instruction *wrpkru*.

There are already some studies focused on applying Intel MPK on application memory protection, like Libmpk (Park et al. 2019) and ERIM (Vahldiek-Oberwagner et al. 2019). But there are few works on MPK for kernel memory protection. This paper is intended to fill this gap.

### Goals

Although applications rely on compatibility and isolation of multi-process abstraction, a practical and efficient multi-process unikernel design is still missing. Thus, we propose Iso-UniK, which has the following goals,

- The multi-process applications should be running in the kernel mode to retain the performance and flexibility of unikernels.
- The isolation of the multi-process design should be flexible and configurable by the user.
- The cross-isolation communication should be efficient enough to handle most of the practical applications.
- Do not rely on hypervisors or cloud providers to be compatible with commercial cloud platforms.

## Design

### Threat model

In this paper, we provide a multi-process like model for Iso-UniK. Each process is treated as a sandbox process. Applications run as sandbox processes in Iso-UniK. We assume attackers are able to invade an application sandbox process through remote attack methods. Therefore, the control flow and all code running inside a sandbox process are untrusted. We assume the hypervisor and the Iso-UniK itself is trusted.

Iso-UniK ensures that the behavior of an untrusted sandbox process would not interfere the kernel and other sandbox processes in the same Iso-UniK system. The access and write to kernel data are banned, and the invocations to kernel services are intercepted with security checks.

Iso-UniK does not protect application from data attacks as we think it is the duty of application. Data-only attack (Vogl et al. 2014; Hu et al. 2016) and side-channel attacks like Meltdown (Lipp et al. 2018) are beyond the scope of Iso-UniK and can be addressed by orthogonal work (Hua et al. 2018).

## Overview

In Iso-UniK, each process is treated as a standalone *sandbox*. Like traditional multi-process design in the monolithic kernel (e.g., Linux), Iso-UniK uses separate page tables from different sandboxes. However, Iso-UniK does not isolate the application from the shared kernel using protection mode, but using Intel Memory protection Keys ("MPK for kernel mode isolation" section) with the inner/outer kernel design ("Inner kernel design for privileged code" section).

### *Programming model*

Existing applications used to using multi-process to isolate vulnerable codes/request handling into a sandbox. Iso-UniK provides a compatible programming model with multi-process based sandbox model. The APIs provided by Iso-UniK are shown in Table 1. The *SANDBOX_FORK* has very similar semantics like the standard *fork* in Linux, but with an additional configuration argument. The configuration argument tells the kernel how to configure the sandbox. An example code of using Iso-UniK to provide sandbox is shown in Listing 1, The main process will create a sandbox using *SANDBOX_FORK*, which forks the current process in a sandbox environment. The return value is the *sandbox-id*. Like *fork* in Linux, this API will return two values, 0 for the in-sandbox environment and a non-zero *sandbox-id* for the parent process.

### *Sandbox configuration*

To provide better flexibility, a *sandbox configuration* is used in Iso-UniK. We provide two kinds of isolation requirements in the configuration: 1) Interface isolation and 2) Memory copying isolation. *Interface isolation* is a black-list approach, to list the denied interfaces for the sandbox in the configuration. Although in unikernel, applications use function call for kernel services instead of system call, there is still a line of interface between kernel and applications. These interfaces can be very large in OSv, for example, the POSIX APIs. It is better to use a black-list approach. While zero-copying of data between kernel and user achieves better performance in unikernels, it leads to potential TOCTTOU (time of check to time of use). *Memory copying isolation* is to tell the kernel whether it should copy or not for the sandboxed codes.

**Listing 1** Sandbox API using Iso-UniK in unikernels.

```
int main(){
  ... //initialization work
  int sandbox_id = sandbox_fork(config);
  if (sandbox_id == 0){
    //in-sandbox handler
    int ret = enter_handler();
    exit(ret);
  }
  wait_sandbox(sandbox_id);
  ...
}
```

## Multi-process based sandbox model

OSv adopts the multi-thread design with single address space, which is not enough for multi-process. In order to support multi-process feature, we need to redesign the memory management of unikernel.

### *Page table*

There is only one page table in the world, which is not conducive to the switching of process address space. For the isolation between the sandboxes, we provide a local page table for each sandbox as shown in Fig. 2. We add the support by 1) adding an address space management in the unikernel, instead of using a global page table; 2) extending the thread abstraction to the sandbox process abstraction.

### *Address space*

The data of the sandbox should be independent of each other, but the data of the kernel should be shared between the sandboxes, such as free page list and scheduler. So,

**Table 1** Sandbox API by Iso-UniK

| API | API Arguments | Description |
|---|---|---|
| *SANDBOX_FORK* | *sandbox configuration* | Fork current process in a sandbox configured with the arguement. |
| *SANDBOX_WAIT* | *sandbox-id* | Wait the sandbox *sandbox-id* to finish. |
| *SANDBOX_CREATE* | *sandbox configuration* | Create a sandbox according to the argument, and ready to run. |
| *SANDBOX_DESTROY* | *sandbox-id* | Destroy the sandbox *sandbox-id* |
| *SANDBOX_RUN* | *sandbox-id* | Run the sandbox *sandbox-id*. |
| *SANDBOX_PIPE* | *sandbox-id, message* | Send/Recevie a message to/from a sandbox. |

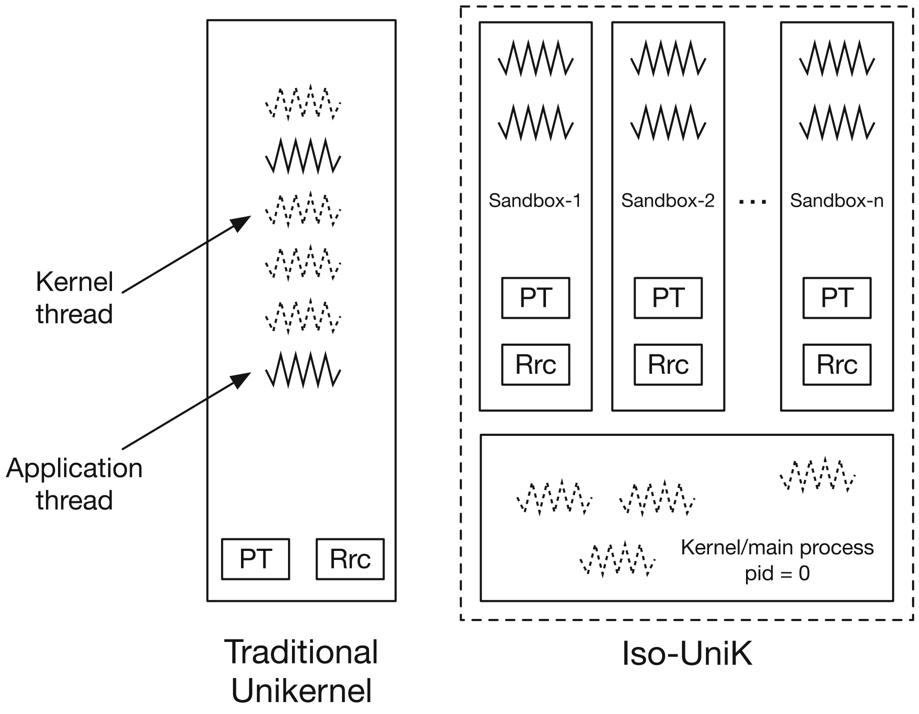

**Fig. 2** Isolation between Sandboxes. Uses page table for inter-sandbox isolation. "PT" means page table, "Rrc" means resources

we divided the single address space of unikernel into the shared space for kernel data and the unshared application space for sandbox process data. At the same time, we found that the data in shared space is critical and affects the running of unikernel and all sandboxes, and needs to be protected from malicious sandbox tampering. Such address space separation is also beneficial for the following isolation work.

As the applications running in the most privileged mode in unikernel, it is a common sense that they are hard to be restricted. For example, codes in the Ring-0 mode can even write a page without write-permission when the write-protection flag is not set.

#### *Restrict kernel behavior with MPK*

The Intel MPK (Memory Protection Keys) is a hardware feature to provide a lightweight memory isolation mechanism for user space. While MPK is originally proposed for user space, it can also be used to restrict the kernel's behavior.

### MPK for kernel mode isolation

MPK is configured by the register PKRU (Protection Key Rights register for User pages). PKRU, a 32-bit register, contains 16 pairs of permission controls bits include "AD" bit for access disable and "WD" bit for write disable. Each user-space page table entry contains a key from 0 to 15, and the corresponding PKRU bits pair will control the access to the page. The MPK does not work when a code in kernel mode privilege accesses a kernel-space memory. However, When *CR4.PKE* is set (Intel® 64 and IA-32 architectures software developer's manual) and PKRU right bits are set disabled, access will be denied when the target memory is in user-space (e.g., user bit in page table entry is set 1) even the code running in kernel mode privilege, as shown in Fig. 3. In the figure, we only configure the first MPK region with "no-read/write", and set all the memory in the page table with the region. Although this permission check will be ignored on any kernel-space memory access from the kernel code, it will work when the kernel code tries to access the memory in user-space. But, MPK does not stop instruction fetch from user-space memory, which means that the program can jump to the protected code in user-space and execute.

PKRU can only be modified using the instruction *wrpkru*, which writes the value of *%rax* into PKRU when *%ecx* and *%edx* are both 0.

#### *Kernel data in user-space for data protection*

As the application located in the kernel-mode in unikernel, we leverage MPK to protect the kernel data by putting it in user-space, configure the page table entries with MPK key 1, and make any memory access from the kernel mode to the protected data forbidden by setting PKRU bits. As shown in Fig. 4, Iso-UniK uses the first PKRU region,

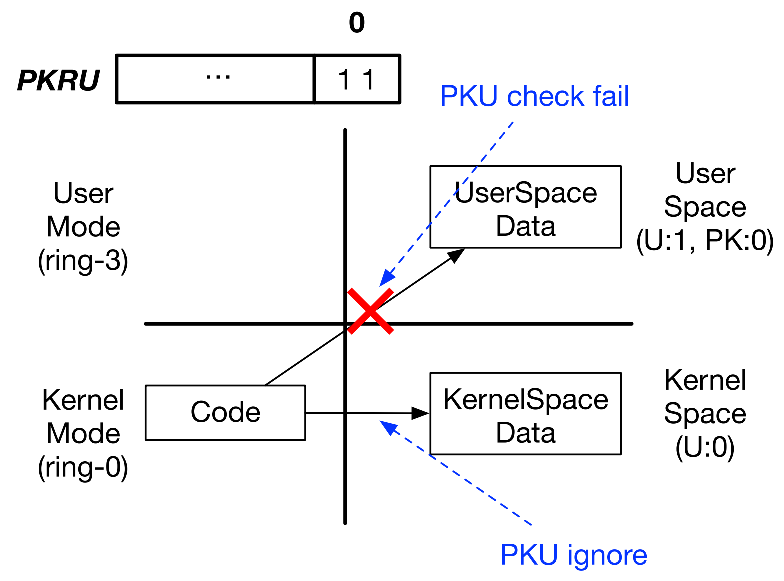

**Fig. 3** MPK Protection. MPK will check the kernel memory accesses on user pages. "U" means user bit and "PK" means MPK protect key index in page table entry

which is set to "no read/write" when application code is running, and switch to "read/write" when the control flow turns to kernel side. In the page table, the memory of protected kernel data will be set as user-space pages, "U:1" in the Figure, to activate MPK checks.

Iso-UniK first introduces the *MPK Gate*, which is proposed to protect the kernel data from an untrusted in-kernel application. Whenever the application sandbox needs to call kernel functions, it first calls the *MPK Gate*, then calls the responding functions. The gate code, shown in Listing 2, will update the PKRU registers to allow the kernel to access its data. The PKRU will be set to protect kernel data again when returning to application code. And Iso-UniK will ensures only *MPK Gate* has the instruction to update PKRU registers.

And there is still a problem that application thread stack is mapped as application data. When a thread call kernel functions, its stack is still writable for other thread of the same process and the control flow may be taken. This is solved by assign a new stack protected by MPK

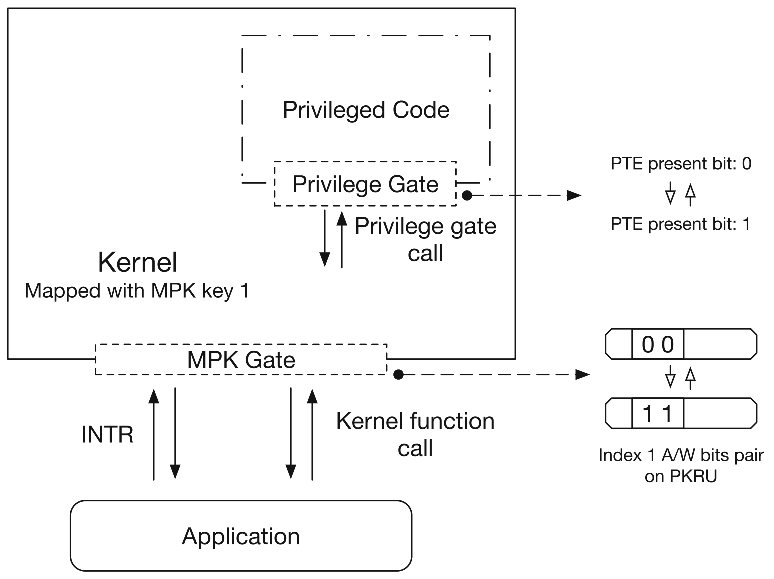

**Fig. 4** Two Gates design. "P" means present bit, "U" means user bit in page table entry

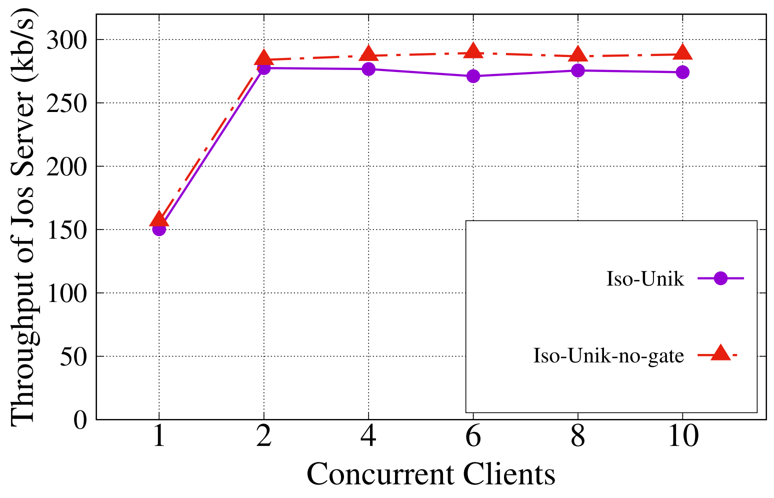

**Fig. 5** The throughput (in kbytes/second) of JOS web server and different concurrent clients

to each application thread. The protected stack is set as same as kernel data. When *MPK Gate* is called, the thread will switch to the protected stack to ensure control flow integrity.

### Inner kernel design for privileged code

Using MPK is not enough since an untrusted application sandbox can still directly jump into any part of the kernel to execute privileged codes, like switching the page table or changing the IDTR register, aiming to take over the control of our unikernel.

**Listing 2** MPK Gate code

```
    cli
;Disable interrupt
    push  %rax,%rbx,%rcx
;Save regs
    xor   %ecx, %ecx
    mov   %ecx, %edx
    mov   Get_MPK_Permission, %eax
    wrpkru
        ;Writes the value of %EAX into
PKRU
    pop   %rax,%rbx,%rcx
;Restore regs
    cmp   $FN_MAX, %rax
;Check funtion id
    ja    abort
    mov PerThreadProtectedStack,%rsp
                    ;Change to protected
                     stack
    sti
;Enable interrupt
    call  *FN_Table(,%rax,8)
                              ;Jump to
function
```

To defend such attacks, Iso-UniK uses an inner kernel and outer kernel design. Specifically, we isolate the privileged codes in the inner kernel, and protect them by unmapping them in the page table, e.g., "P:0" shown in Fig. 4. When kernel needs to execute the privileged codes, it must go through a gate, called *Privilege Gate*, shown in Listing 3, which will map the privileged codes into the page table and then execute the codes. Privileged codes will be unmapped again before return to the normal kernel.

**Listing 3** Privilege Gate code

```
void inner_gate_call(uint call_id,
          uint64_t arg1, ...)
  {
       irq_disable();
// Disable interrupts
       map_privileged_code();
       TLB_flush();
       if(!is_rsp_in_kernel_stack())
           abort();


/// abort if not from kernel stack
       switch(call_id){
           case WRITE_CR3:
             ...
           case SET_IDT:
             ...
           ...
       }
       unmap_privileged_code();
       TLB_flush();
       irq_enable();
// Enable interrupts
  }
```

Iso-UniK will ensures there are no privileged codes in the address space except the *Privilege Gate*. Although the *Privilege Gate* and *MPK Gate* introduces some overhead, our evaluation ("Applications" section) shows the cost of isolation is acceptable for real applications.

## Implementation

We describe our specific implementation of the Iso-UniK in the section. Specifically, we implement the prototype of Iso-UniK on OSv, with about 4700 LoC C++ modifications and 100 LoC assembly code. We choose OSv because it provides other Linux APIs support for applications except the multi-process feature.

## Multi-process functionality support

OSv, as a unikernel, adopts a single address space design with one global page table, which is sufficient for running multi-thread programs. However, this design does not meet the requirements of multi-process feature which is needed by many real applications. To this end, We add sandbox process abstraction to OSv.

### *Address space*

OSv does not have the multi address spaces semantics, and it has only one global page table.

Iso-UniK adds “local_cr3”, which stands for base address of a page table, in the sandbox process struct to support the process abstraction. And threads of the same process share one “local_cr3”. Iso-UniK also modifies the scheduler for the switch of page tables between sandbox processes.

In OSv, the kernel and application use the same memory allocation mechanism, and the address space is not separated. Iso-UniK separates the address space of OSv into two parts, shared space which is shared among sandbox processes and application space which is private for a sandbox process. Shared space is set with MPK key 1 in page table entries to be protected. This work is done by provide a new memory allocator for application with the lower address space, and modify application heap, application stack, and ELF&library load address to the lower address space. And we implement the *sandbox APIs* introduced in “Overview” section using our sandbox process abstraction.

There are other resources that should be unique to each process, such as file descriptor table, which in OSv is also a global one for all threads. They are also be added to sandbox process abstraction and be switched when process is switched.

### *SANDBOX_FORK implementation*

*Fork* is an operation whereby a process creates a copy of itself. Many wellknown applications, like Nginx (Nginx), use *fork* to create new processes. Programmers use the return value of *fork* to distinguish between parent and child processes, and then let them do different things. In OSv, we implement a *fork* like API *SANDBOX_FORK* for applications.

When a process call *SANDBOX_FORK*, it first pushes some registers on the stack, saving the context. Then, a new thread is created and assigned to a new process struct with new pid. The parent process turns to sleep, and the thread of the child process is awakened, copies the page table of the parent process, sets unshared application space to copy-on-write. Other resources are also copied for new process. Child process then restores the registers and return address from stack, sets *%rax* to 0 which is the return value of *SANDBOX_FORK*, and jumps to the return address of *SANDBOX_FORK*. Father process then wakes and returns with the pid of child process.

## Multi-process security support

### *System configuration*

To use MPK, we must set the PKE bit. To ensure that the paging is enabled, we set the PG and PE bits of %CR0, the PAE bit of %CR4, LME bit of EFER MSR. To map specific pages unwritable, we set the WP bit of %CR0.

We then ensure that the application’s code segment is set to read-only, preventing runtime modifications. The kernel’s pages are protected by MPK and cannot be modified while the application is running. Except for the pages of the kernel and the application’s code, the rest of the pages are set to be nonexecutable.

### *Outer kernel with MPK gate*

If an application thread needs to call a kernel function, it must first modify PKRU to get access to kernel data. As *wrpkru* is a critical instruction, Iso-UniK wraps *wrpkru* with *MPK Gate* to reduce the attack interface. When an application needs to call an important kernel function, it will first store the *function id* in a register such as *%rax* before calling *MPK Gate*. We replace the function calls with *function id*s and *MPK Gate* calls in the compilation process of application without modification to applications. A read-only table records the mapping from function id to function address. In addition to kernel function calls, interrupt handler is also protected with *MPK Gate*. In *MPK Gate*, Iso-UniK sets the PKRU bits, looks up the function table, switches to protected stack, then finally calls the corresponding function. It takes about 45 CPU cycles to finish this process. PKRU permissions can only be modified by the instruction *wrpkru*. We can analyze the application to ensure that the *wrpkru* instruction is not used inside application using binary check.

To support unmodified applications, Iso-UniK replaces kernel function calls with MPK Gate calls during compilation. The compilation process typically consists of four stages: preprocessing, compilation, assembly, and linking. Between the compilation stage and assembly stage of the application, Iso-UniK gets the assembly code and replace kernel invocations with call MPK Gate. A function ID is passed by a register to MPK Gate. This design improves the compatibility of Iso-UniK.

### *Identify the inner kernel*

Iso-UniK uses *Privilege Gate* to prevent application from jumping and executing privileged instructions. Due to the significant overhead caused by mapping and unmapping, we must try to delineate the scope of code that *Privilege Gate* protects.

Modification of critical registers may cause protection failures. We analyze the assembly code and inline assembly code of OSv, and move the modification code of key registers into the protection scope of *Privilege Gate*, such as read or write the control registers, SIDT(set interrupt descriptor table), etc. We unmap the pages that contain these instructions and map them only when *Privilege Gate* invoked. The unremovable privileged code that runs during OSv boot will be unmapped before the application runs. Although performance is affected, since these registers are not often used, the overhead is acceptable.

### *Binary check on the applications*

We first remove the privileged code outside of OSv *Privilege Gate*, replaced by *privilege_gate_call()* and *call id*. The application and the required dynamic link library binaries are scanned before loading to ensure that they do not contain privileged code.

We currently summarize the privileged codes that need to be quarantined, some of which are listed in Table 2. In addition, since the design of the *Privilege Gate* is very scalable, privileged code can be added as configured without effort.

## Evaluation

In the evaluation, we try to answer these four questions:

- *Question-1*: How efficient Iso-UniK supports multi-process operations?
- *Question-2*: How Iso-UniK influences unikernel applications' performance?
- *Question-3*: Can Iso-UniK defend attacks from the applications?
- *Question-4*: Is it easy to port Iso-UniK for other unikernel systems?

**Table 2** Some of the privileged instructions

| Privileged Instructions | Influences |
|---|---|
| wrpkru | Modify PKRU for A/W permissions |
| mov %REG, %CR0 | Modify flags such as WP or PG |
| mov %REG, %CR3 | Modify the base of page table |
| mov %REG, %CR4 | Modify flags like PKE |
| sidt | Modify interrupt descriptor handler |
| wrmsr | Modify model specific bit like NX |

And it is easy to configure other instructions. Except *wepkru* is used by *MPK Gate*, other instructions should be protected by the *Privilege Gate* and unmapped in most of the time

### Evaluation environment

We use an x86-64 machine with an 40-core Intel(R) Xeon(R) Gold 6138 CPU (2.00GHz), 128GB memory and a 160GB SSD. The host OS is Ubuntu 18.04 with Linux Kernel 4.15.7. The prototype of Iso-UniK is implemented based on OSv unikernel (Kivity et al. 2014) of git commit #64dfbcdd, with about 4700 LoC C++ modifications and 100 LoC assembly code. For experiments, we assigned 4 vCPUs (virtual CPUs) and 4GB memory for all the evaluated systems used in following tests. We use Qemu 2.11.1 with KVM as the hypervisor.

### Microbenchmark

We present several microbenchmarks to show the performance of Iso-UniK.

**Methodology.** We measured the performance of basic system calls and multi-process interfaces of Iso-UniK. The baseline systems are OSv (commit 64dfbcdd) and Ubuntu 16.04 (with Linux kernel 4.9.75) running in qemu with KVM. We also present a performance-optimized version of Iso-UniK, Iso-UniK-no-gate, which does not include the two gates isolation. We compare Iso-UniK and Iso-UniK-no-gate to show the performance impacts of the two gates isolation methods.

We use LMBench to evaluate the system call latency, and evaluate the latency multi-process interfaces by continuously invoking it.

**Results.** Table 3 shows the mean latencies of several typical system calls. Compare with OSv, Iso-UniK provides a multi-process program model, and the overhead is brought by page table switch and the two gates. As shown in *Null* syscall testing, the *MPK Gate* introduces some overhead compared to OSv, but is still 5x faster than Linux kernel. Compare with Linux kernel, Our *SANDBOX_FORK* and *SANDBOX_EXIT* can be 2x faster, and also have a better performance at the latency of pipe. The latency of *SANDBOX_FORK* (without Exit, not list in the table) of Iso-Unik is 66μs with gate and 59μs without gates. And in *Open*&*Close* testing, the unikernels and Linux kernel are all tested with ZFS, which is the primary file system supported by OSv. This shows that the implementation of file system in OSv is not as perfect as Linux kernel. Iso-UniK also introduces some overhead to *Open*&*Close* compared with origin OSv.

And the overhead of *Privilege Gate* is more expensive than *MPK Gate* when privileged instructions (like write *%CR3* to switch page table) is invoked. Under *Multi-Process* tests, Iso-UniK-no-gate has a much better performance than Iso-Unik, and is closed to origin OSv in *pipe* testing. But in *Open*&*Close* testing, the overhead of isolation is acceptable.

Our design is more attractive when compared to other hypervisor-based or host-based designs. According to the paper (Zhang et al. 2018), the latency of *fork()* in Kylinx

**Table 3** The mean latency (in µs or microseconds) of some important multi-process interfaces and syscalls evaluating with lmbench

| Systems | Multi-Process | | | Syscall | |
|---|---|---|---|---|---|
| | **Fork&Exit** | **Pipe** | **Ctx** | **Null** | **Open&Close** |
| Linux Kernel | 217.8 | 10.46 | 7.794 | 0.1974 | 1.717* |
| OSv | - | 3.48 | - | 0.0013 | 4.560 |
| Iso-UniK-no-gate | 65.8 | 4.32 | 4.256 | 0.0014 | 5.107 |
| Iso-UniK | 103.1 | 9.64 | 7.746 | 0.0425 | 5.181 |

"Iso-UniK-no-gate" means disable the two isolation methods. "ctx" means context switch *: The "Open&close" is tested under Linux 4.4.0 with ZFS

takes about 1.3 ms, slower than Ubuntu (1.0ms in their paper). The latency of *pipe()* in Kylinx is similar to Ubuntu (55µs versus 54µs). The latency of Graphene in the test *fork*&*exit* is 463µs (Tsai et al. 2014), much slower than Linux (67µs in their paper). The latency of IPC (*msgsnd* and *msgrcv*) in Graphene is about 5.0x slower than Linux. As Iso-UniK performs better than Ubuntu, we believe Iso-UniK works better than the designs mentioned before.

## Applications

We port two applications, JOS web server and Nginx, to Iso-UniK to show the performance on real-world applications. Iso-UniK can run unmodified Nginx directly, which proves the compatibility.

**JOS web server.** In order to show the functionality of multi-process feature and the overhead introduced by isolation methods in Iso-UniK, we evaluate the JOS web server(Jos Tiny Server) in Iso-UniK, which utilizes multi-process feature such as *fork* to handle HTTP requests. JOS web server is a tiny web server for lab in MIT OS lessons, which has a simple and clear workflow for testing performance of multi-process feature. The main process of server is always listening and establishes connections with clients. Once it accepts a request, the main process will fork a child process to actually handle the request (such as read file, send contents), and itself will wait to answer next request. We use benchmark tool ab(Apache HTTP server benchmarking tool) to evaluate the performance of this server on Iso-UniK and Linux kernel. The test file is an index html file of about 124 bytes. And the request repeats for 600 times.

The result is shown in Fig. 5. The isolation methods result in about 2.5% to 6.3% lower throughput when comparing Iso-Unik and Iso-Unik-no-gate, which is acceptable.

**Nginx.** Nginx (Nginx) is a well-known and second most widely used web server around the world. It can be used as a reverse proxy, load balancer, mail proxy, HTTP cache, etc. In order to show that the multi-process model of Iso-UniK can help applications take advantage of multi-core to improve performance, we run Nginx on Iso-UniK. The version of Nginx is 1.12.2 and the performance is tested using the ab benchmark with 6000 requests. The test file is the default index html file of Nginx, about 612 bytes. Each unikernel is assigned with 6 vCPUs.

In the original OSv, after the removal of codes related to multi-process features, Nginx can run as one process, accepting and responding to requests. In Iso-UniK, through *fork()*, multiple Nginx processes can work parallelly, and they can run on different virtual CPU cores, accepting and responding to requests at the same time. The result is shown in Fig. 6. When the client concurrency is 2, the throughput in the original OSv is higher, because the pressure of requests is not heavy and Iso-UniK introduces some overhead. As the number of client concurrency increases, the throughput of Iso-UniK rises faster with the benefit of multi-core. Compared with the single-process Nginx in the original OSv, Iso-UniK is about 1.15x to 1.17x faster. And when client concurrency is large, the throughput of Iso-UniK is only 0.4% less than Iso-UniK-no-gate.

Figure 7 shows the throughput of different numbers of Nginx processes running in Iso-UniK with 60 concurrent clients. As the number of Nginx processes increases, the throughput of the entire system increases. The multi-process model introduced by Iso-UniK helps applications to get better performance with security.

## Security analysis

We analyze some of the methods that malicious applications might use to attack kernel critical data or other sandboxes.

**Kernel Data Exposure Elimination.** We evaluate the data protection by scan critical memory region. We write a tool to dump the memory region of the kernel page pool as an application. And we allocate a page in the kernel page pool to represent some secret and important data. As shown in Fig. 8, because the whole unikernel runs in the kernel mode, it is easy for application to reach the data of kernel, include some secret and critical data. And if we enable the *MPK Gate* isolation method, the region of the kernel page pool is hidden and safe. This shows that the *MPK Gate* helps to protecte the kernel data.

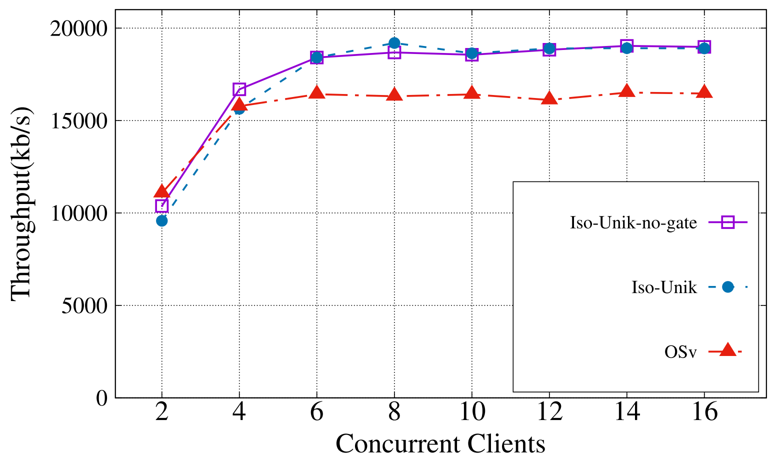

**Fig. 6** The throughput (in kbytes/second) of Nginx server. Two processes in Iso-UniK, and one process in origin OSv

**Arbitrary Directly Jump.** MPK is not designed to block instruction fetches, application can directly jump to kernel-space code. However, the attacker cannot modify the data in kernel-space code as MPK still prevent the access and write. And there is no privileged code outside *Privilege Gate* for attacker to exploit. Therefore, the malicious application sandbox process cannot affect other sandboxes.

**Directly Jump to *MPK Gate* Code.** Malicious applications can jump directly to the wrpkru instruction in outgate to get MPK permissions. But outgate will continue to check the function id in the register and can only continue to jump to the entry of the finite corresponding kernel functions. Malicious apps are not free to change the control flow.

**Directly Jump to wrpkru before application initialization.** We need to use *wrpkru* to set data in shared space Unreadable and unwritable. And we immediately check the value of %rax, and it should be the 1 at corresponding AD and WD bit of PKRU. If application tries to set %rax with a value jump to *wrpkru* to write PKRU and get MPK permissions, it will be forbidden by Iso-UniK.

**Directly Jump to Map Code in *Privilege Gate*.** The page table page is protected by MPK, and is free from the malicious applications when directly jump to map code in *Privilege Gate*.

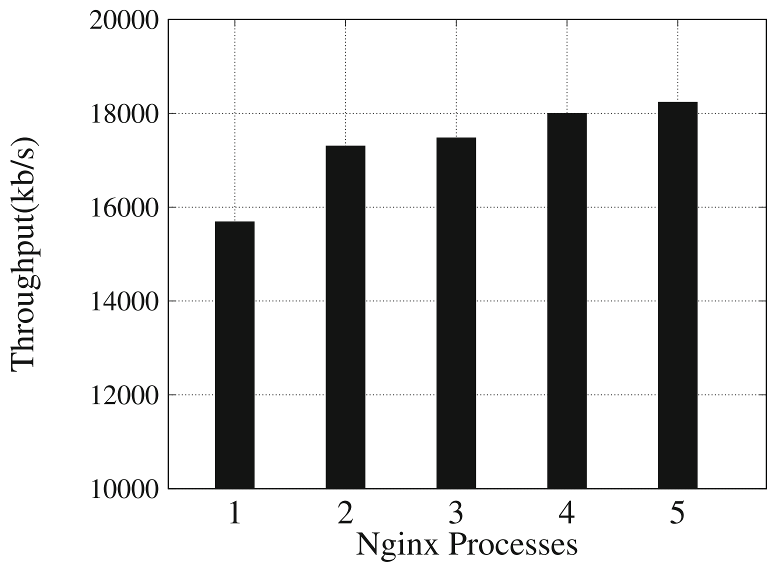

**Fig. 7** The throughput (in kbytes/second) of Nginx server with different numbers of processes

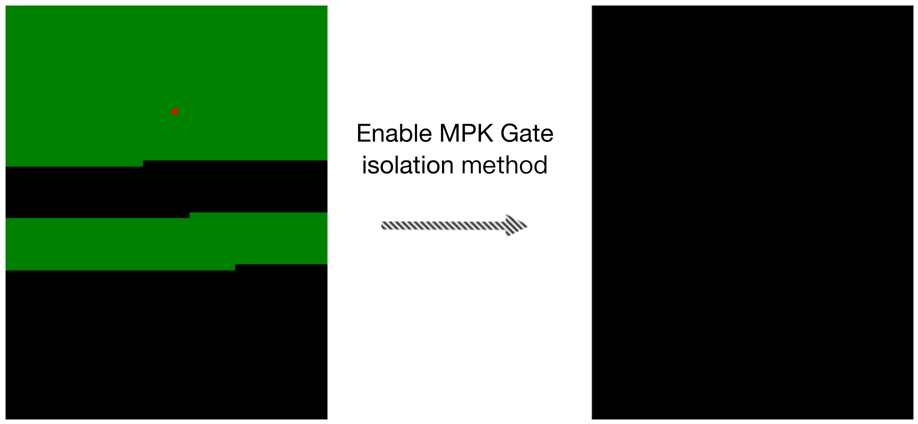

**Fig. 8** The memory layout for kernel page pool from the view of application. Green for accessible area. Black for unaccessible or invalid area. Red for the pages containing secret

**Side-channel Attacks.** Side-channel attacks like Meltdown (Lipp et al. 2018) are beyond the scope of this paper. Other orthogonal works like EPTI (Hua et al. 2018) can help to solve the problem with software solution with some overhead. And hardware manufacturers also plan to address these issues at the hardware level, such as improving speculative execution mechanisms.

## Lessons learned

It is straightforward to support multi-tasking on a mature unikernel systems like OSv. In this section, we present several challenges we met and how we solved them , then we discuss about the generality of the design.

**Stack Switch in *MPK Gate*.** If an application thread needs to call a kernel function, it first calls the *MPK Gate* to modify PKRU for permission of kernel data. As the stack of thread is exposed to other thread of the same process, it needs to change to a protected stack in *MPK Gate*, to guarantee the control flow of the thread with MPK permission. It is implemented as modifying the stack register *%RSP*.

One of the problems is that the arguments passed to the function are pushed on the stack, and if the stack is changed, the arguments may be lost. However, The modern Intel x86-64 architecture provides more registers, and some of the registers are used to store function arguments to accelerate the function calls. We compile Iso-UniK with GCC version 5.4.0, which uses 6 specific registers(*%RDI, %RSI, %RDX, %RCX, %R8, %R9*) to pass up to 6 arguments. And the arguments of Iso-UniK kernel functions are no more than 6. All the arguments of kernel functions used by Nginx, are passed in registers, and avoid the problem of arguments lost when stack switches.

Another conclusion is that, when a kernel function is finished, the stack of kernel function is destroyed and nothing is stored on the protected stack. So it is only need to provide a fixed protected stack for each thread, and switch to the fixed stack each time *MPK Gate* is called.

**Fork Implementation Details.** In Iso-UniK, when a process calls *fork()*, a new process should be created as a copy of the calling process. There will be problems if father process copies the page table for new process. Father is still running, and the page table is not stable as the stack may change. So we first create a new thread, pause the father process, then use the new thread to copy the page table of father. After copy, the new thread switch to the copied page table and become the thread of child process.

During the page table copy, *Copy-on-Write* is considered as an efficient method to reduce time and memory usage. However, in OSv, there is no support for *Copy-on-Write* feature. So Iso-UniK adds the feature to the unikernel. Specifically, Iso-UniK adds a check of the Copy-on-Write bit in page table entry in page fault handler. When the Copy-on-Write bit is found when page fault happens, a copy of the old page will be made and be filled to the page table.

When other work is done, the child thread directly jumps to the return address of *fork()* and runs just like the father with a different return value of 0. The return value can be modified by set *%RAX*, and the return address can be found by *%RSP*.

If child process runs on the same CPU processor, it will be long before father process wakes as child process needs to consume some time. This results in a long *fork()* latency. And if Iso-UniK is assigned multiple processors, it is better to assign the child process to another CPU processor to reduce the latency.

**Inter-Process Communication Support.** In OSv, *pipe()* is implemented as to write/read a memory region in kernel shared space. The region is abstracted as a file with file descriptor. During *fork()*, the file descriptor table is copied, and the memory region is shared between page tables of processes. So Iso-UniK supports *pipe()* as an inter-process communication between father and child processes.

**Generality.** It is easy to port Iso-Unik to other unikernel systems. For the unikernels based on virtualization platforms like KVM, unikernel itself plays the role of memory and resource manager. It is therefore possible to add mechanisms that Iso-UniK utilizes to support the multi-process feature, such as multiple page tables, separation of address space, mechanisms of fork and IPC. And the isolation method introduced by Iso-UniK needs the MPK hardware feature provided by Intel x86-64 architecture. But similar hardware features can also be found on other architectures, like the Domain Access Control on ARM. So it is possible to provide efficient isolation methods for unikernels running on different architectures.

## Related work

**Multi-process Feature in Unikernel.** Prior art has already explored the multi-process support in unikernels. Graphene (Tsai et al. 2014) treats an instance of unikernel as a process. Graphene runs instances on host Linux within a picoprocess, and implements the multi-process feature with the help of picoprocess of Linux. The cost of this solution is that the isolation is not as good as the virtual machine. Kylinx (Zhang et al. 2018) treats a single VM as a process and uses the method of VM-fork to solve the multi-process problem. It modifies Xen to offer fork and IPC. This solution is not suitable for existing cloud vendors as it requires modifications on the hypervisor. Iso-UniK is the first work to support multi-process with high-security assurance using MPK, and is compatible with existing cloud providers.

**Isolation Using Hardware Isolation.** Most of prior systems (Dautenhahn et al. 2015; Shi et al. 2017; Hua et al. 2018; Liu et al. 2015; Li et al. 2019; Hua et al. 2017) rely on privilege modes or address space (e.g., page table or extended page table) for isolation. Nested Kernel (Dautenhahn et al. 2015) proposes a nested kernel architecture, and uses a map and unmap method to protect privileged instructions in kernel mode. Deconstructing Xen (Shi et al. 2017) applies the Nested Kernel architecture in Xen hypervisor.

MPK is a new hardware feature, which is proposed for user-space isolation. There are some works (Park et al. 2019; Vahldiek-Oberwagner et al. 2019) are using MPK feature for application isolation. Libmpk (Park et al. 2019) provides a library with more semantic-gap-mitigated and scalable abstraction with Intel MPK. ERIM (Vahldiek-Oberwagner et al. 2019) helps application protect sensitive data from other untrusted components of the application using MPK without requiring control-flow integrity. Iso-UniK has a similar MPK gate design with ERIM, but we modify the compilation process to insert MPK gate without modifying application codes. Moreover, all the related systems can not leverage MPK for kernel-space isolation.

**Efficient Inter-process Communication.** L4 microkernel (Klein et al. 2009) proposes direct process switch to boost the IPC performance. Moreover, recently efforts on hardware-software co-design to optimizing IPC performance, including CrossOver (Li et al. 2015), Codom (Vilanova et al. 2014), SkyBridge (Mi et al. 2019) and XPC (Du et al. 2019), can significantly reduce the IPC latency. Crossover and SkyBridge leverage a hardware virtualization feature, VMFUNC, which enables a virtual machine to directly switch its EPT (extended page table) without trapping to the hypervisor. XPC proposes two hardware extensions, direct switch and relay segment, to achieve fast domain switching and zero-copying data transfer. The design of Iso-UniK is compatible with existing IPC model, thus is easy to adopt optimizations proposed by these systems.

## Conclusion

The lack of multi-process feature in unikernel makes it less flexible and applicable for nowadays applications. Meanwhile, the isolation between processes is a necessary feature to ensure the security of applications like Nginx. This paper proposes Iso-UniK, a multi-process sandbox model for unikernels. Iso-UniK is compatible with existing multi-process interfaces, has high-security assurances through Intel MPK, and does not need any modifications in the hypervisor. Iso-UniK uses a novel *two-gates configurable isolation* technique to balance the performance and security, and is also the first work to leverage Intel MPK for kernel-mode isolation. The overhead introduced by two gates isolation is only about 2.5% to 6.3% in Tiny Server, 0.4% in Nginx.

**Acknowledgements**
Not applicable.

**Authors' contributions**
The fist author conceived the idea of the study and finish the implementation. The first and second author wrote the paper. All authors discussed the results and revised the final manuscript. All authors read and approved the final manuscript.

**Funding**
Sponsored by Program of Shanghai Academic/Technology Research Leader (No.19XD1401700).

**Availability of data and materials**
Not applicable.

**Competing interests**
The authors declare that they have no competing interests.

Received: 4 October 2019 Accepted: 27 December 2019
Published online: 07 May 2020

**References**

Amit N, Wei M (2018) The design and implementation of hyperupcalls. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA. pp 97–112. https://www.usenix.org/conference/atc18/presentation/amit

Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html. Accessed Dec 2019

Apache http server project. https://httpd.apache.org/. Accessed Dec 2019

Apache MPM prefork. https://httpd.apache.org/docs/2.4/mod/prefork.html. Accessed Dec 2019

AWS AmazonWebServices. https://aws.amazon.com/?. Accessed Dec 2019

Belay A, Bittau A, Mashtizadeh A, Terei D, Mazières D, Kozyrakis C (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. pp 335–348. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

Dautenhahn N, Kasampalis T, Dietz W, Criswell J, Adve V (2015) Nested kernel: An operating system architecture for intra-kernel privilege separation. In: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15. Association for Computing Machinery, New York, NY, USA. pp 191–206. https://doi.org/10.1145/2694344.2694386

Du D, Hua Z, Xia Y, Zang B, Chen H (2019) Xpc: Architectural support for secure and efficient cross process call. In: Proceedings of the 46th International Symposium on Computer Architecture, ISCA '19. Association for Computing Machinery, New York, NY, USA. pp 671–684. https://doi.org/10.1145/3307650.3322218

gVisor. https://gvisor.dev/. Accessed Dec 2019

Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z (2016) Data-oriented programming: On the expressiveness of non-control data attacks. In: 2016 IEEE Symposium on Security and Privacy (SP). pp 969–986. https://doi.org/10.1109/SP.2016.62

Hua Z, Du D, Xia Y, Chen H, Zang B (2018) EPTI: Efficient defence against meltdown attack for unpatched vms. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA. pp 255–266. https://www.usenix.org/conference/atc18/presentation/hua

Hua Z, Gu J, Xia Y, Chen H, Zang B, Guan H (2017) vtz: Virtualizing ARM trustzone. In: 26th USENIX Security Symposium (USENIX Security 17). USENIX Association, Vancouver, BC. pp 541–556. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/hua

Intel® 64 and IA-32 architectures software developer's manual. https://software.intel.com/en-us/articles/intel-sdm. Accessed Dec 2019, Published November 11, 2019

Jos Tiny Server. https://pdos.csail.mit.edu/6.828/2014/labs/lab6/. Accessed Dec 2019

Kashyap S, Min C, Kim T (2018) Scaling guest OS critical sections with ecs. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA. pp 159–172. https://www.usenix.org/conference/atc18/presentation/kashyap

Kivity A, Laor D, Costa G, Enberg P, Har'El N, Marti D, Zolotarov V (2014) Osv—optimizing the operating system for virtual machines. In: 2014 USENIX Annual Technical Conference (USENIX ATC 14). USENIX Association, Philadelphia, PA. pp 61–72. https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity

Klein G, Elphinstone K, Heiser G, Andronick J, Cock D, Derrin P, Elkaduwe D, Engelhardt K, Kolanski R, Norrish M, et al. (2009) Sel4: Formal verification of an os kernel. In: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, SOSP '09. Association for Computing Machinery, New York, NY, USA. pp 207–220. https://doi.org/10.1145/1629575.1629596

Li W, Xia Y, Chen H, Zang B, Guan H (2015) Reducing world switches in virtualized environment with flexible cross-world calls. In: Proceedings of the 42nd Annual International Symposium on Computer Architecture, ISCA '15. Association for Computing Machinery, New York, NY, USA. pp 375–387. https://doi.org/10.1145/2749469.2750406

Li W, Xia Y, Lu L, Chen H, Zang B (2019) Teev: Virtualizing trusted execution environments on mobile platforms. In: Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE 2019. Association for Computing Machinery, New York, NY, USA. pp 2–16. https://doi.org/10.1145/3313808.3313810

Lipp M, Schwarz M, Gruss D, Prescher T, Haas W, Fogh A, Horn J, Mangard S, Kocher P, Genkin D, Yarom Y, Hamburg M (2018) Meltdown: Reading kernel memory from user space. In: 27th USENIX Security Symposium (USENIX Security 18). USENIX Association, Baltimore, MD. pp 973–990. https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

Liu Y, Zhou T, Chen K, Chen H, Xia Y (2015) Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, CCS '15. Association for Computing Machinery, New York, NY, USA. pp 1607–1619. https://doi.org/10.1145/2810103.2813690

Madhavapeddy A, Mortier R, Rotsos C, Scott D, Singh B, Gazagnaire T, Smith S, Hand S, Crowcroft J (2013) Unikernels: Library operating systems for the cloud, vol. 41. Association for Computing Machinery, New York, NY, USA. pp 461–472. https://doi.org/10.1145/2490301.2451167

Manco F, Lupu C, Schmidt F, Mendes J, Kuenzer S, Sati S, Yasukata K, Raiciu C, Huici F (2017) My vm is lighter (and safer) than your container. In: Proceedings of the 26th Symposium on Operating Systems Principles, SOSP '17. Association for Computing Machinery, New York, NY, USA. pp 218–233. https://doi.org/10.1145/3132747.3132763

Mi Z, Li D, Yang Z, Wang X, Chen H (2019) Skybridge: Fast and secure inter-process communication for microkernels. In: Proceedings of the Fourteenth EuroSys Conference 2019, EuroSys '19. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3302424.3303946

Nginx. http://nginx.org/en/. Accessed Dec 2019

Open Container Initiative. https://www.opencontainers.org/about. Accessed Dec 2019

Park S, Lee S, Xu W, Moon H, Kim T (2019) libmpk: Software abstraction for intel memory protection keys (intel MPK). In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). USENIX Association, Renton, WA. pp 241–254. https://www.usenix.org/conference/atc19/presentation/park-soyeon

Sehr D, Muth R, Biffle C, Khimenko V, Pasko E, Schimpf K, Yee B, Chen B (2010) Adapting software fault isolation to contemporary cpu architectures. In: Proceedings of the 19th USENIX Conference on Security, USENIX Security'10. USENIX Association, USA. p 1

Shi L, Wu Y, Xia Y, Dautenhahn N, Chen H, Zang B, Guan H, Li J (2017) Deconstructing xen. In: 24th Annual Network and Distributed System Security Symposium,(NDSS'17), San Diego, CA, USA. The Internet Society, Reston, Virginia, U.S. https://doi.org/10.14722/ndss.2017.23455

The heartbleed bug. http://heartbleed.com/. Accessed Dec 2019

Tsai C-C, Arora KS, Bandi N, Jain B, Jannen W, John J, Kalodner HA, Kulkarni V, Oliveira D, Porter DE (2014) Cooperation and security isolation of library oses for multi-process applications. In: Proceedings of the Ninth European Conference on Computer Systems, EuroSys '14. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2592798.2592812

Vahldiek-Oberwagner A, Elnikety E, Duarte NO, Sammler M, Druschel P, Garg D (2019) ERIM: Secure, efficient in-process isolation with protection keys (MPK). In: 28th USENIX Security Symposium (USENIX Security 19). USENIX Association, Santa Clara, CA. pp 1221–1238. https://www.usenix.org/conference/usenixsecurity19/presentation/vahldiek-oberwagner

Vilanova L, Ben-Yehuda M, Navarro N, Etsion Y, Valero M (2014) Codoms: Protecting software with code-centric memory domains. In: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA). pp 469–480. https://doi.org/10.1109/ISCA.2014.6853202

Vogl S, Pfoh J, Kittel T, Eckert C (2014) Persistent data-only malware: Function hooks without code. The Internet Society, Reston, Virginia, U.S. https://doi.org/10.14722/ndss.2014.23019

Yee B, Sehr D, Dardyk G, Chen JB, Muth R, Ormandy T, Okasaka S, Narula N, Fullagar N (2009) Native client: A sandbox for portable, untrusted x86 native code. In: 2009 30th IEEE Symposium on Security and Privacy. pp 79–93. https://doi.org/10.1109/SP.2009.25

Zhang Y, Crowcroft J, Li D, Zhang C, Li H, Wang Y, Yu K, Xiong Y, Chen G (2018) Kylinx: A dynamic library operating system for simplified and efficient cloud virtualization. In: 2018 USENIX Annual Technical Conference (USENIX ATC 18). USENIX Association, Boston, MA. pp 173–186. https://www.usenix.org/conference/atc18/presentation/zhang-yiming

## Publisher's Note