EasyChair Preprint

№ 8870

User Space Privileged Function Calls

Nafiseh Moti, Reza Salkhordeh and André Brinkmann

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

September 23, 2022

User Space Fast Privileged Function Calls

Abstract. The operating system's traditional design controls and manages all system resources, which comes at the cost of performance and scalability overhead. The scalability overhead results from the kernel's internal metadata structures and locks primarily designed for sequential access. Additionally, implementing software services and resource management requires compliance with the strict kernel abstractions and programming paradigms that can result in semantic bugs. Although plausible, decoupling from the strict kernel control path and code stack comes at the penalty of losing a higher trust entity to enforce protection separation and protection of user code and data. This paper offers a hardware-assisted method to run confined user-space functions at a higher privilege level. Our method allows the implementation of fined-grained user-level services and protocols without modifying the operating system's protection scheme. This is done by introducing two high-level instructions to the x86 ISA. Our simulation shows that user-level functions that leverage our instructions run in the same order as standard function calls, while the real benefit lies in the flexibility and ability to decouple the protected code from the kernel limitations.

Keywords: Operating system · Protection rings · Privilege separation · user space protection.

1 Introduction

The sub-microsecond latency of new storage technologies and network interfaces accentuates the cost of software overhead previously masked by the cost of slow I/O [25]. The monolithic kernel software stack and its scalability limit are among the main contributors to the software overhead in traditional storage and network services [23, 24]. The cost of kernel software stack stems from its role in applications' control and data path as a centralized resource management entity and, accordingly, the overhead of synchronization, context switch, and data copies [24]. Kernel code contains general-purpose abstractions and sequential data structures containing internal locks that limit the scalability of ever-increasing parallel codes. As a result, Kernel-bypass and decentralized, adhoc user-level services have gained popularity to mitigate these costs [25].

Although bypassing monolithic kernel abstraction for data and control path seems plausible, any implemented resource management needs to enforce permissions and isolation between user processes accessing the data. As a result, control plane management still requires a higher trust entity to ensure protection protocols. The previous kernel-bypass proposals either kept the control plane inside the kernel or radically changed the operating system process isolation schemes.

The former limits the scalability and flexibility of user-level code design, and the latter requires forgoing compatibility with the previous libraries and services.

To implement a secure user-space code in conventional operating systems, many user-level data services such as memcached [12] and in-memory file systems [8, 18] leverage Intel's memory protection keys (MPK) to decouple their control plane from the kernel. Intel MPK offers a set of instructions and register and provides a mechanism for enforcing page-based protections without modification of the page tables and by using unused bits as encryption keys [18]. However, MPK is unaware of the user code and relies on page-level protection, making developing fine-grained protocols complicated. Additionally, access control change in MPK is costly and only supports a maximum of 16 protection domains. Simurgh [16] took a different approach and provided hardware-assisted function level protection in a user-level NVMM (non-volatile main memory) file system and showed how enabling lightweight, safe user-level functions and, as a result decoupling from the kernel virtual file system led to scalability and performance benefits. However, the Simurgh approach did not consider control transfer cases in the CPU, which limits the generality of their approach.

This paper offers a general-purpose hardware-assisted solution that enables running user-enforced protocols and software services implemented as safe functions at a higher CPU privilege level.

Implementing safe functions is enabled through introducing two new instructions `JMPS` (safe jump) and `RETS` (safe return) and one microinstruction `RDEP` that allows identifying whether the running code is safe or not. These instructions provide the means to execute a confined user-level set of functions with higher privilege under special circumstances. Similar to Simurgh, we use predefined page offsets as entries to the safe code pages. Our proposal removes the need for switching to kernel code stack through system calls for protection. Safe functions ease the design of decentralized protocols, conditional data sharing, and permission enforcement tailored to the needs of applications. Our solution requires minimal changes to the kernel and page table entries and is compatible with both `SMEP` and `SMAP` (Supervisor Mode Execution Prevention and Supervisor Mode Access Prevention [9]).

The key challenges of providing a solution to run privileged instructions in the user space are preserving the integrity of the operating system kernel and the system's stability. We address these challenges by proposing changes to the memory and interrupt management of the system and by offering suggestions for the code linking and compilation so that it minimizes the system vulnerability.

Our simulation analysis on Gem5 [4] simulator shows that the safe functions run at the order of regular function calls. However, the actual performance benefit of safe functions lies in the possibility of running decentralized and user-level implemented protocols at a higher privilege level in the context of monolithic kernels while avoiding internal kernel locks and abstractions. This can help with the performance-critical software services that require microsecond latency.

The next section discusses the background and related work. Section 3 describes the safe functions architecture and how we manage control transfer cases

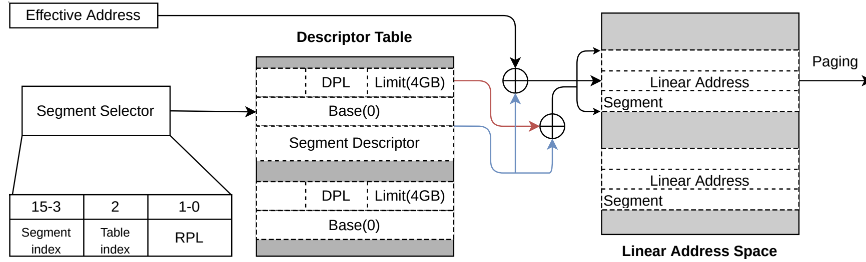


Fig. 1: Segmentation in 64-bit Long mode

in the CPU. In Section 4 we evaluate the safe functions and finally, Section 5 concludes the paper.

2 Background and Related Work

Many research proposals tried to address the limitation of context switches between kernel and user code in data and control path of the system by either defining new protection schemes and blurring the line between kernel and user space or by improving context switch overhead through different methods such as a batching system calls or removing unnecessary checks. However, these methods either need extensive operating system change or do not offer the flexibility and scalability of user-level code [11].

Aside from the scalability penalty, implementing the control path of the application in the kernel restricts the flexibility of the implementation, complicates the code debugging, and might even come at a vulnerability cost due to semantic mismatch and concurrency control. For example, in 2018, a ten years analysis of syscall-specific commits to the Linux kernel showed that 35% of these commits were bug fixes, mostly rising from semantic and concurrency control bugs in memory management [2]. Another study in file system design showed that data structure inconsistencies between the kernel virtual file system(VFS) and file system code in metadata management compose more than 70% of the file system vulnerabilities [6]. These data services, therefore, often take years to enter the Linux kernel upstream path since they need to gain a general user audience and go through the verification process to prevent compromising the whole system's security.

Our goal is to provide a method for developing safe user-level services and to enable running privileged instructions from the user space without affecting the system's stability or changing operating system paradigms. This section first explains the CPU and operating system's privilege level, protection mechanism, and inter-privilege control transfer mechanism. Later we discuss the related work and how our solution can be compared to them.

2.1 Protection Rings and Paging

CPU protection rings are the legacy of segmentation in the system's memory. Before the introduction of virtual memory, paging, and flat memory model, segmentation was the default method to provide memory and inter-process isolation [1, 9]. Each segment has a size of 64KB. Accessing a segment requires that a 16-bit segment selector be loaded into a segment register. The segment selector contains the Requested Privilege Level (RPL) and is used to make requests from a high privilege level on behalf of a lower one [1, 9]. Figure 1 shows how segmentation is being implemented in the X86 processors.

Four protection rings define the system's running state and privilege level. Ring 0 is the highest privilege level used to define the kernel mode. The operating system uses code and data segments to switch between different rings. The minimum access level for these segments is defined by their Descriptor Privilege Level (DPL), while the current privilege level(CPL) defines the privilege level the processor is running in. CPL is changed by setting the DPL value of the code segment(CS). The processor keeps a cached CPL value stored inside the lower two bits of the `%cs` register. The equations $RPL \geq CPL$ and $DPL \geq CPL$ need to be satisfied to access a segment.

Although X86 supports four protection rings, most operating systems only support rings 0 and 3. Ring 0 provides privilege instructions that protect writes to %cs, memory read and writes, I/O port access and control registers, and allows running privilege instructions in CPU. The operating systems' addressing and memory access control methods depend on the CPU's running mode. Most operating systems use the 64-bit protected mode of the x86 architecture. This mode uses a flat memory space, and all segments create the same flat 64-bit linear address space. It supports only flat address spaces with a single code, data, and stack space. In this mode, memory segmentation is mostly disabled; however, it is still used to change between different privilege rings.

For efficiency and to prevent TLB flushes upon system calls and traps, the Linux kernel places the kernel code and data into the address space of every running process. Every x86 page table entry(PTE) contains a user/supervisor(U/S) bit to define kernel pages and prevent unprivileged memory access. For 64-Bit Long mode, there are four levels of page tables used to translate an address. Setting the U/S bit in each PTE level marks all the pages below that level as supervisor. These pages are only accessible in rings 0-2, and any unprivileged access to them results in a general protection exception `(#GP)`.

2.2 System Call and Control Transfer Mechanisms

There are various x86 instructions to pass control over to the kernel (ring 0) and to request services from it. This service interface is implemented inside the kernel by providing different system calls. Each system call is uniquely identified by a number and is used to expose kernel functionalities such as managing interprocess communication and shared resource management.

Control transfer instructions such as `SYSCALL/SYSRET`, `CALL/JMP/RET` and `INTn/IRET` causes the processor to perform privilege check. Fast system call instruction pairs `SYSENTER/SYSEXIT` and `SYSCALL/SYSRET` eliminate unnecessary checks and load pre-determined values inside CS and SS registers [1]. These instructions can enter and exit the kernel code on predefined locations specified by special purpose registers [1, 9] and only after the necessary checks are they allowed to change the privilege level.

2.3 Related Work

The previous kernel bypass research proposals focus on overcoming the performance and scalability limitation of user/level context switch by moving all or part of the data or control path to the user space. In these systems, protection in user space is offered through new hardware-assisted methods or special CPU instructions.

Data-centric [3, 5, 17] operating systems and capability-based designs [19, 22] are two approaches proposed in previous studies to offer complete user space designs and to address the protection and isolation requirements. In these systems, data often carries its permission through universal pointers, and the data access control is checked using hardware-assisted capability design or programming language level bound checks. However, using new operating systems means compromising the compatibility of the existing libraries and applications.

Intel memory protection keys (MPK) offer user-space protection, and it has been used by several user-level services to protect their code and data pages [10,18,21]. However, MPK-based protection lacks flexibility and the protection granularity of general-purpose microservices. Additionally, executing privilege instructions is impossible due to the inability to share user/supervisor pages.
.

Since system calls are an integral part of an operating system to allow user mode to interact with the kernel, there has been a vast body of research on improving them. System calls must constantly change to meet current security requirements as new vulnerabilities are discovered. These changes often come with a performance penalty [2]. Different approaches focus on improving security or performance by providing alternatives to the traditional syscall mechanism. LOTRx86 is a response to the HeartBleed vulnerability [14]. It introduces an additional privilege level, called PrivUser, at ring one and exports these functions using a new call interface. This approach takes 1000-1500 cycles since it affects caching mechanism. FlexSC [20] proposes exception-less system calls by asynchronously handling them. Finally, Privbox [13] runs the syscall-heavy parts of the code in a privileged sandbox. These approaches, however, only focus on solving the overhead of system calls. Simurgh [16] proposed a hardware-assisted method to provide fine-grained security to their user-level NVMM file system. File system functions were implemented in confined protected code pages, and they showed how they could improve the performance and scalability of the file system design. However, unlike this paper, they did not consider different control transfer mechanisms.

3 Safe Functions

Executing safe functions and allowing the user space process to elevate its privilege level without going through the standard kernel entry points requires some measurements to ensure that they comply with operating system security and do not affect the system's stability. These system stability measurements need to be guaranteed: 1) prevent normal functions from accessing the safe data, 2) disallow normal functions to change safe code, and 3) provide a safe means for transitioning privilege from normal to supervised mode. A user application can access the safe data pages exclusively through the safe functions. These functions are exported in a one-time bootstrapping process via a kernel module to the user address space and act as a cross-privilege trampoline. Upon the library's initialization, this module is invoked, and the application, as a result, can continue its safe execution in the user space.

3.1 Safe pages

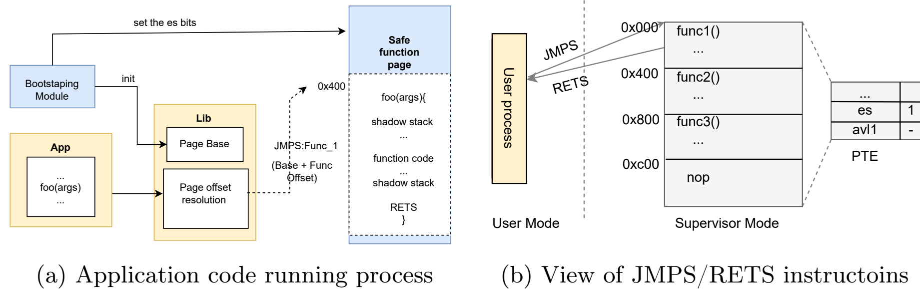
Allowing arbitrary code to be executed with an elevated privilege level causes security risks. An attacker could execute malicious code with an elevated privilege level, thereby compromising the whole system's security. Privilege elevation can only be performed upon jumping to code locations that are stored in safe pages to disallow arbitrary and malicious code to be executed with a higher privilege level. These pages are marked with a special execute safe bit (es-bit). Changing the execute safe bit or the page's content is only possible from within the kernel. We have modified existing page table entries (PTE), and translation lookaside buffer(TLB) check policies to guarantee these protections. Figure2b shows the view of safe pages. Each safe page is divided into four evenly sized sections, resulting in four entry points. Given a page size of 4096 bytes, a page can only be entered on byte 0 (0x000), 1024 (0x400), 2048 (0x800) and 3072 (0xC00). If the virtual address of the destination corresponds to one of these page entry points, the code is allowed to be executed with a higher privilege level. Otherwise, a general protection fault will occur and aborts the execution.

3.2 Privilege escalation/de-escalation

To enable ring elevation and to jump to the safe user space function, we have introduced two new instructions:

- Jump safe(JMPS): This instruction allows to jump through safe functions on safe page entry points, change the call stack pointer to safe pages and continue the execution.
- Return safe(RETS): This instruction returns from the safe execution function and restores the privilege level if required.

Unlike the system call mechanism mentioned in Section 2.1, `JMPS` does not changes the existing user code segment and does not exchange and load the



(a) Application code running process (b) View of JMPS/RETS instructoins

segments. It runs in the context of the user process. Similar to the absolute indirect `JMP/CALL` mechanism, the `JMPS` instruction jumps to a specific code location and saves the next instruction pointer onto the stack before jumping. When running an application in user mode, it has a privilege level of 3. `JMPS` instructions elevate the current privilege level of the running process, so a jump to a predefined user-specific virtual address can be performed. The code continues its execution from this address until a RETS instruction is encountered. This instruction will revert the privilege level to the user level and return the program flow to the next instruction after JMPS. Figure 2b shows the execution mechanism of a safe function.

Nesting `JMPS` calls is enabled by popping the return addresses from the stack on `RETS`. In case of a nested `JMPS/RETS` call, `RETS` must not reset the privilege level. Otherwise, the parent `JMPS` operation would resume execution with a privilege level of 3 instead of 0. Therefore resetting the privilege level is skipped if the es-bit for the corresponding return page is set.

Executing JMPS at intermediate rings If safe functions do not need privilege instructions and only require the safe execution of the user code in a higher privilege level, they can be implemented in the intermediate rings 1 or 2 alternatively. In this case, the control path still needs to pass the CPU segmentation check, and any attempt to execute privileged instructions will result in a general protection fault. The design and overhead of `JMPS` and `RETS` will remain the same in this case.

3.3 Control transfer management

The `JMPS` instruction changes the privilege level without handing over control to the kernel. The current process can therefore be interrupted or run a control transfer instruction. To guarantee system stability in case of an external interrupt, it is necessary to ensure the privilege level is reverted if another process is scheduled. The elevated privilege has to be restored only if the execution is returned to an es-bit protected page. Returning from control transfer instructions does not check any paging metadata. This check can be done using an additional TLB check upon returning from instructions or by introducing a new control register. We took the first approach and modified the control transfer

return instructions to check for the safe bit of the return address before resetting the elevated privilege. To check the return address, we introduce a new micro-operation to set a flag if the es-bit is set in the target address page. This instruction is necessary if we want to restore the elevated privilege while an interrupt or system call occurs in the middle of the safe function execution path.

Handling timer interrupts When an interrupt occurs, the main process information is stored on the stack in an interrupt frame. This interrupt frame includes a copy of the current code segment register, which in turn contains the RPL value of the code segment. When `IRET` is executed, the CPL value is restored using the RPL value inside this interrupt frame. We have changed the `IRET` instruction to check whether the return address is inside a safe page. In this case, the CPL value of 0 is loaded instead of the RPL value.

Handling interrupts The kernel is not involved in the execution path of the safe functions. Therefore CPU is responsible for handling interrupts. If an interrupt happens while a safe function with elevated privilege is running, the privilege level has to revert to the user level before another process is scheduled. Otherwise, a newly scheduled process could run with supervisor privilege.

Handling system calls On `SYSENTER/SYSCALL` the kernel software stack and page tables will be loaded before elevating the privilege; therefore, it is safe to enter the kernel space using kernel entry points in the system call. Upon returning from the system call, we need to check and restore the elevated privilege to the safe execution mode. This can be done by checking the return address of the caller process in the call stack. Consider the scenario that two processes, A and B, should be scheduled. Assuming process A is executing code with an elevated privilege level using `JMPS`. Process B becomes active, and A waits for the execution to resume. If B forces a scheduled event by calling the corresponding syscall, the kernel might return the execution to process A using the `SYSRET` instruction instead of `IRET`. `SYSRET` always restores the user code segment with a DPL value of 3, discarding the elevated privilege level that A held previously. It is necessary to modify `SYSRET` similar to `IRET` to consider this case and restore the privilege level if required. Before loading the user code segment attributes, the instruction must check if the es-bit is set. If this is the case, it loads a DPL value of 0 instead of 3. Writing the attributes will cause the CPL value to be set to the code segment DPL value.

SMAP/SMEP Supervisor Mode Access Prevention (SMAP) is an Intel CPU feature that, if enabled, prevents supervisor mode (Ring 0-2) from accessing user pages. Similarly, if Supervisor mode execution prevention (SMEP) is enabled, the operating system will not be allowed to directly execute application code, even speculatively.

SMEP does not affect safe function since safe function does not execute any code in user pages. Safe functions are also compatible with SMAP as safe, and supervisor data pages are only accessible and modifiable from within the safe functions.

micro-op	param1	param2	param3	description
limm	dest	lmm		Stores the 64 bit immediate lmm into the integer register dest
rdip	dest			Set the dest register to the current value of rip
wrattr	dest	src1		Writes the selector in src1 to the register dest
subi	dest	src1	lmm	Substracts the contents of lmm from src1 and pushes to dest
wrip	src1	src2		Set the rip to the sum of src1 and src2, this causes a macroop branch
addi	dest	src1	lmm	Adds src1 and the immediate lmm and puts the result in dest

Table 1: micro-op symbols used in the code snippets

3.4 Security Consideration

Safe functions provide a way to run privileged instructions from the user space. We guarantee that normal users cannot access the privileged code or the safe data, and we guarantee a safe control transfer between different privilege modes. In addition, we guarantee that a user cannot execute or jump to arbitrary locations inside the kernel. However, our work does not guarantee the security of the user code. Developers need to verify and trust their code shared by different processes.

When executing code using `JMPS`, no branch instructions pointing outside of an es-bit protected page should be performed since the privilege level is not reset in this case. For E.g., while executing privileged code with `JMPS`, a call instruction to a standard user page is performed. In this case, the privilege level is not reset. The code executed by the call instruction is therefore executed with a privilege level of 0. It is the responsibility of the user to prevent this case.

To prevent return-oriented programming (ROP) attack, the return address and the stack must be placed inside safe pages. Storing the stack in a normal user page could enable another thread of the same process to change the return addresses on the stack. In this case, `RETS` would return to the compromised return address and execute the altered program flow. If the return address does not point to an es-bit protected page, `RETS` will reset the privilege level to user level. This minimizes the risk of an unintended privilege escalation but does not solve the problem of the altered program flow. Therefore changing the stack pointer is still necessary. Since `JMPS` to a safe page elevates the privilege, Time Of Check To Time Of Use `(TOCTTOU)` does not happen between the `JMPS` and the beginning of the function.

3.5 Bootstrapping process

For an application to be able to access the safe functions, a one-time bootstrapping process is being enforced that marks the supervisor bit in the safe pages. Bootstrapping module creates a mapping between the user and safe code pages and returns the base address of the safe pages.

3.6 Implementation

We have implemented and prototyped the new instructions in Gem5 [4] simulator. Table1 lists the micro-ops and symbols used in this section.

```
limm t1 , 0x3FF ;checking the entry points
and t0 , reg , t1 , EZF<-1
IF EZF=0 THEN #GP(0)
...
limm t1 , 0x3FF ;checking the es-bit
and t0 , reg , t1, EZF<-1
IF EZF=0 THEN #GP(0) ;issue general protection fault
...
rdip t1 ;saving the return address
stis t1 , ss , [0 , t0 , rsp ] ;Store t1 on the stack
subi rsp , rsp , ssz ;Subtract the stacksize from rsp
...
limm t4 , 4GB ;changing the privilege level and jump
wrattr cs , t4
wripi reg , 0
```

Fig. 3: Jumping to a safe page

Execute safe bit To mark a page as executed safe, the es bit on all four levels of page table entries needs to be set. However, for setting the supervisor bit, only the last level needs to check the bit as the CPU extends supervisor mode to all pages below a page level [1, 9]. To implement es-bit, We have repurposed bit 10 in the page table entry. This bit is one of the bits available to the user. We have modified the TLB check to disallow normal users to change the es bit. Additionally, we have implemented a new micro-op called `rdep`. This instruction reads the es-bit value and sets the `EZF` flag. `JMPS` utilizes this flag to check the safe jump condition.

Implementing JMPS and RETS Figure 3 shows the implementation of `JMPS`. At the beginning, it is checked whether the jump address is a valid offset in a page. This is done by checking whether the lower 12-bits of the address are divisible by 1024 and subsequently sets the EZF flag. The instruction will next check the execute safe bit. This is done through *rdep* micro-operation. If all the checks pass successfully, the instruction needs to save the return address on top of the stack. Nesting `JMPS` calls is, therefore, possible by popping the return addresses from the stack on the `RETS`. Before performing the jump, the instruction loads new attributes with a DPL value of 0 for the current user code segment. By applying these attributes using `wrattr`, the current privilege level is elevated from 3 to 0. At the end of this macro-operation, a branch happens by writing the new instruction pointer. However, in our real hardware test, it seems that Intel processors does not bother to check DPL and RPL selector and only rely on CPL and paging protection mechanism.

The `RETS` instruction does not need any input parameters. Figure 4 shows the Gem5 implementation of the return instruction. The CPU reads the return address from the top of the stack and loads it into t1.

```
ld t1 , ss , [1 , t0 , rsp ]
addi rsp , rsp , dsz
; Check for return to es - bit protected page
rdep t1 , flags =( EZF ,) , atCPL0=True
br label ( " return" ) , flags =( CEZF ,)
limm t4 , 4GB ; Reset the privilege level
wrattr cs , t4 ; Changing the CPL
return:
wripi t1 , 0 ; Jump to the return address .
```

Fig. 4: Returning from JMPS

```
protToVirtFallThrough:
rdep t1 , flags =( EZF,) ,atCPL0=True
limm t5 , 0
br label ("skipDefault") ,flags =( CEZF,)
andi t5 , t2 , 0x3
skipDefault: ...
```

Fig. 5: IRET instruction modification

Interrupt return handling As explained in section 3.3 to restore the privilege level in case of a return from control transfer instructions, we have to modify the interrupt return instruction. We have used the `rdep` micro-op to check whether the es-bit is set for the return address, and in this case, we can safely return to the code execution using elevated privilege. Figure 5 shows the micro-op code that needs to be inserted into the `IRET` instruction. Register t5 stores the temporary value that will be used to restore the CPL. If the es-bit is set for the return location, the default behavior of loading the RPL value into t5 (line 5) is skipped. Register t5 will still hold the value of 0 set in line 3. If the es-bit is not set, line 5 will be executed and override the temporary CPL value of 0 with the cached RPL value. This approach causes an additional performance overhead to the `IRET` instruction. The slowest part of this modification lies within the fact, that the page of the return address has to be loaded into the TLB to check the es-bit. To resume the code execution, this step has to be performed anyways. The time spent on the TLB load operation is therefore negligible.

Linking the library and Alignment Check The safe function code should be compiled as position independent without any external library dependency. Additionally, no branch to the outside of the pages like exception handling should be allowed. All needed functions need to be implemented inside the safe pages. The entry of functions needs to be placed in the predefined page offsets of the .text section as jumping is only allowed to those offsets. In GCC, this can be done

in the application link script by telling the compiler to put each function in a separate section and by adding section attributes to the functions.

4 Evaluation

This section describes the simulation infrastructure, experimental setup, and results. As mentioned in section 3.6 we have implemented our new instructions in Gem5 [4] simulator. We have measured the overhead of the JMPS/RETS instruction and compared it with the Linux system call and Linux fast system call(vDSO call) on the Gem5 simulator using a 1GB DerivO3CPU, 512MB memory. vDSO(virtual dynamic shared library) is a small shared library provided by the kernel and mapped into the address space of the user application. This enables the fast execution of frequent system calls that do not need additional security measures without mode and stack change [7]. Additionally, we measured the overhead of the Linux system call and fast system call on an Intel Xeon Gold 5212 processor running at 2.5GHz. to grasp the real performance measurements of safe function calls. Next, we show the breakdown measurement of safe function and system call on Gem5. The simulator runs a Gentoo distribution on top of the Linux kernel 5.4.55. The kernel module and PTEditor [15] were loaded to the image. The goal of the simulation is to show that the `JMPS` overhead is low compared to a system call.

4.1 Measurements

To measure the overhead of syscall, we have added an empty system call and vDSO call to the kernel. We modified the syscall entries and used m5_reset_stats and m5_dump_stats pseudo instructions and hardcoded the *rdi* and *rsi* registers to minimize the measurement overhead. Each measurement was performed 100 times to account for caching and speculation execution effects. The overhead of vSDO call and safe function were measured between the function calls. Figure6 shows the baseline measurements. On Gem5, `JMPS/RETS` call performs close to vDSO call. Table 2 shows the breakdown of instructions and their overhead run on Gem5 for jump safe compared to system calls. The Gem5 overhead of a vDSO routine, including its return, is 30 cycles. On Xeon CPU, this call takes less than two cycles.

The gem5 implementation `JMPS` checks the `es` bit in the page table, modifies the CPL value of the CPU, and performs the call routine to the predefined safe function page. The `JMPS` and `RETS` combined overhead is 111 cycles and, therefore, in the same order as a standard function call on Gem5.

The syscall overhead is induced not only by the call itself but also by setting up the registers and copying parameters to memory, switching to the kernel context, and locating the corresponding function for the syscall through the dispatching table. In contrast, `JMPS` employs the same technique as a normal function call for passing parameters and does not require a context switch. The user code simply calls safe functions by their addresses and not by a number;

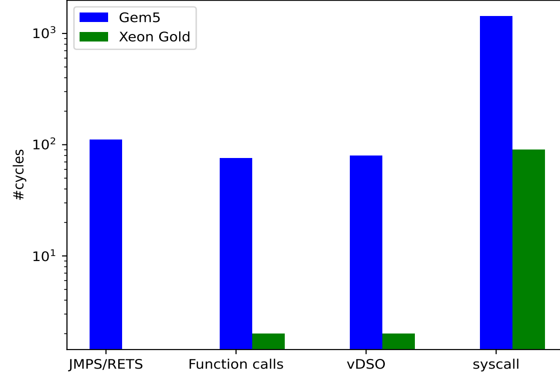


Fig. 6: Cycle count measurement

JMPS/RETS		System call	
Instruction	Time(cycles)	Instruction	Time(cycles)
Checking entries	6	syscall wrapper	241
PTE check	6	swapgs	22
Saving return address	9	Switch to CR3	214
Change CPL	32	construct pt_regs	248
Write instruction pointer	6	do_syscall	226
Read Return Address	6	find return path	10
PTE check	6	sysret	45
Revert CPL	32	USERGS_SYSRET64	400
Write Instruction pointer	6		
total: 111 cycles		total: 1432 cycles	

Table 2: The breakdown of instruction measurements on Gem5 simulator

hence, `JMPS` does not need a dispatching table. Changing the CPL value and writing the return address in the safe stacks are the subset of the syscalls' operations also needed by `JMPS` and take 30 cycles. Additionally, checking the `es` bit and entry points can be made in 6 cycles.

5 Conclusion and future work

In this paper we introduced a new hardware assisted method to execute user level functions in a higher privilege. The safe inter-ring transition was provided by introducing two new instructions to X86 ISA. Our proposed instructions provide a safe jump and return to the confined user verified code pages. Our prototype on Gem5 simulator showed that the safe functions run at the same order of

magnitude as normal function call. We believe that our proposal improves the flexibility and scalability in the implementation of safe user-level and kernel-bypass software services.

References

1. Amd64 architecture programmer's manual (2021)
2. Bagherzadeh, M., Kahani, N., Bezemer, C., Hassan, A.E., Dingel, J., Cordy, J.R.: Analyzing a decade of linux system calls. Empir. Softw. Eng. **23**(3), 1519–1551 (2018)
3. Baumann, A., Barham, P., Dagand, P., Harris, T., Isaacs, R., Peter, S., Roscoe, T., Schüpbach, A., Singhania, A.: The multikernel: a new OS architecture for scalable multicore systems. In: Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14
4. Binkert, N.L., Beckmann, B.M., Black, G., Reinhardt, S.K., Saidi, A.G., Basu, A., Hestness, J., Hower, D., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Altaf, M.S.B., Vaish, N., Hill, M.D., Wood, D.A.: The gem5 simulator. SIGARCH Comput. Archit. News **39**(2), 1–7 (2011)
5. Bittman, D., Alvaro, P., Mehra, P., Long, D.D.E., Miller, E.L.: Twizzler: a data-centric OS for non-volatile memory. In: USENIX Annual Technical Conference (ATC), July 15-17. pp. 65–80 (2020)
6. Cai, M., Huang, H., Huang, J.: Understanding security vulnerabilities in file systems. In: Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys 2019, Hangzhou, China, Augsut 19-20, 2019. pp. 8–15. ACM (2019)
7. Davis, M.: Creating a vdso: the colonel's other chicken. https://dl.acm.org/doi/fullHtml/10.5555/2073763.2073769 (2012)
8. Dong, M., Bu, H., Yi, J., Dong, B., Chen, H.: Performance and protection in the zofs user-space NVM file system. In: Brecht, T., Williamson, C. (eds.) Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019. pp. 478–493. ACM (2019)
9. Guide, P.: Intel® 64 and ia-32 architectures software developer's manual. Volume 4 (2022)
10. Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M.L., Shen, K., Marty, M.: Hodor: Intra-process isolation for high-throughput data plane libraries. In: Malkhi, D., Tsafrir, D. (eds.) 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019. pp. 489–504
11. Kim, T., Peinado, M., Mainar-Ruiz, G.: STEALTHMEM: system-level protection against cache-based side channel attacks in the cloud. In: Kohno, T. (ed.) Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012. pp. 189–204. USENIX Association (2012)
12. Kjellqvist, C., Hedayati, M., Scott, M.L.: Safe, fast sharing of memcached as a protected library. In: Amaral, J.N., John, L.K., Shen, X. (eds.) ICPP 2020: 49th International Conference on Parallel Processing, Edmonton, AB, Canada, August 17-20, 2020. ACM
13. Kuznetsov, D., Morrison, A.: Privbox: Faster system calls through sandboxed privileged execution. In: 2022 USENIX Annual Technical Conference (USENIX ATC)
14. Lee, H., Song, C., Kang, B.B.: Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In: Lie, D., Mannan, M., Backes, M., Wang, X. (eds.) Proceedings of the 2018 ACM SIGSAC Conference on Computer and

Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018. pp. 1441–1454. ACM (2018)
15. Michael Schwarz, Moritz Lipp, C.C.: misc0110/pteditor: A small library to modify all page-table levels of all processes from user space for x86_64 and armv8. https://github.com/misc0110/PTEditor (2018)
16. Moti, N., Schimmelpfennig, F., Salkhordeh, R., Klopp, D., Cortes, T., Rückert, U., Brinkmann, A.: Simurgh: a fully decentralized and secure NVMM user space file system. In: de Supinski, B.R., Hall, M.W., Gamblin, T. (eds.) SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021. pp. 46:1–46:14
17. Narayanan, V., Huang, T., Detweiler, D., Appel, D., Li, Z., Zellweger, G., Burtsev, A.: Redleaf: Isolation and communication in a safe operating system. In: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. pp. 21–39. USENIX Association (2020)
18. Park, S., Lee, S., Xu, W., Moon, H., Kim, T.: libmpk: Software abstraction for intel memory protection keys (intel MPK). In: Malkhi, D., Tsafrir, D. (eds.) 2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019. pp. 241–254. USENIX Association (2019)
19. Shapiro, J.S., Smith, J.M., Farber, D.J.: EROS: a fast capability system. In: Proceedings of the 17th ACM Symposium on Operating System Principles (SOSP), December 12-15. pp. 170–185 (1999)
20. Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., Paek, Y.: HDFI: hardware-assisted data-flow isolation. In: IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, May 22-26. pp. 1–17 (2016)
21. Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N.O., Sammler, M., Druschel, P., Garg, D.: ERIM: secure, efficient in-process isolation with protection keys (MPK). In: 28th USENIX Security Symposium, USENIX Security, Santa Clara, CA, USA, August 14-16. pp. 1221–1238 (2019)
22. Watson, R.N.M., Woodruff, J., Neumann, P.G., Moore, S.W., Anderson, J., Chisnall, D., Dave, N.H., Davis, B., Gudka, K., Laurie, B., Murdoch, S.J., Norton, R.M., Roe, M., Son, S.D., Vadera, M.: CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In: IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, May 17-21. pp. 20–37 (2015)
23. Wippel, H.: Dpdk-based implementation of application-tailored networks on end user nodes. In: 2014 International Conference and Workshop on the Network of the Future, NOF 2014, Paris, France, December 3-5, 2014. pp. 1–5. IEEE (2014)
24. Yang, Z., Harris, J.R., Walker, B., Verkamp, D., Liu, C., Chang, C., Cao, G., Stern, J., Verma, V., Paul, L.E.: SPDK: A development kit to build high performance storage applications. In: IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017. pp. 154–161. IEEE Computer Society (2017)
25. Zhang, I., Raybuck, A., Patel, P., Olynyk, K., Nelson, J., Leija, O.S.N., Martinez, A., Liu, J., Simpson, A.K., Jayakar, S., Penna, P.H., Demoulin, M., Choudhury, P., Badam, A.: The demikernel datapath OS architecture for microsecond-scale datacenter systems. In: van Renesse, R., Zeldovich, N. (eds.) SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021. pp. 195–211 (2021)