IEEE-SEM, Volume 9, Issue 7, July-2021
ISSN 2320-9151

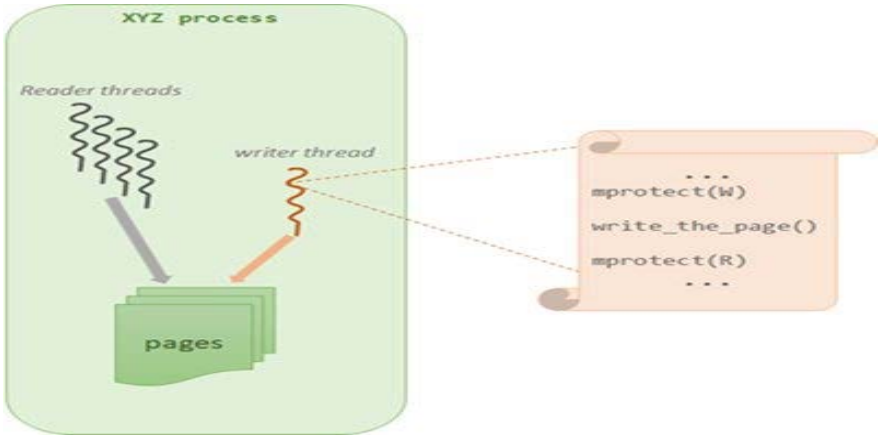
Memory Protection using Intel's Memory Protection

Vishal Bhandari, Archana Patil

Abstract-Out of the five main tasks of operating system i.e. Hardware abstraction, file management, memory management, process management and process scheduling, Memory protection is an important part of memory management . The OS restricts one process from accessing the memory of another process but it doesn't provide security between memory access of threads of same process[6]. This paper proposes a new hardware feature provided by Intel i.e. Memory Protection Keys. This feature enables us to provide memory protection between threads[2]. It uses the previously unused bits of page table[2]. The important benefits of MPK are : performance ,group-wise control , per-thread view protection techniques[1].

I. INTRODUCTION

Memory protection is a technique to control memory corruption on a computer[6]. Many modern OS provide inbuilt memory protection. Memory protection's main aim is to prevent one process from accessing memory of other process[6] .Thus a process having a bug does not access address space of other processes or the OS itself. Protection to a specified area may consist of read/write access to a area or a attempt to execute it. If a process tries to access unauthorized memory it generates SEGFAULT or can lead to abnormal termination of process[6]. Operating systems provide an isolation between different processes and their memory by working in tandem with MMU (widely also known as virtual memory and address space techniques). Where a virtual address space is available for each process (memory isolation) that means any changes by a process will not be visible to other processes in the system (except in specific cases). By these techniques, Intrinsically we get memory isolation between processes. But a common question which comes out is what isolation is present within process? and that is Page based protection mechanism Page based Protection : A process's virtual address space consists of memory pages (contiguous memory chunks of some fixed size), and each page has protection flags (in both page table and TLB) which determines the kind of access allowed to this page at a given point in time (Read, Write Execute)[6]. These flags can be modified with the mprotect() system call[6]. The arguments to mprotect are an address of a memory region, the dimensions of the region, and a group of protection flags[7]. The address of the region must be aligned to the system's page size, and the length of the region must be in the multiple of page size[7]. A typical use-case is to protect memory pages in a process from accidental coding errors. Basically when a page is allocated it is marked as Read-Only and the writer thread would legitimately call mprotect() system call to change the corresponding page's permission before touching it and will change it back to Read-Only after it is done with writing[7]. As mentioned earlier, mprotect() syscall is used to update the protection flags but one of biggest disadvantages is that it does not perform well where protected pages are very frequently updated. Because mprotect() syscall would require a context switch and Flushing of TLB to reload the updated protection bits[1]. It performs even badly if protected memory regions are not contiguous (e.g. Sparse pages)[1]. Given this limitation, developers would not enable this feature in performance sensitive code paths. and hence it would be difficult to detect and fix memory corruption issues in production.



Previously used approaches certainly will be helpful to catch memory corruption issues in in-house testing but given that performance overhead these may add, we may not be able to enable them in production code path. In general the iea is to be able to catch all the memory corruption issues in in-house testing but we would not always be. Given that MPK is a hardware feature and provides a mechanism to change the permission of multiple pages with a single instruction, We can keep this feature always ON. In this approach we would create a Memory Protection Key and then assign this key to all the pages. Now that pages are tagged with a key, any thread which wishes to access these pages need to first grab the corresponding key and then proceed to write the page. The basic idea is to change the code that legitimately accesses or modify these pages to grab the right key before reading or writing and to drop the key after the job is done. The assumption being, any thread that unintentionally corrupts memory would most likely not have bothered to grab the key associated with the page, and will therefore be caught red handed.

II. MPK EXPLAINED

Intel Memory Protection Keys (Intel MPK) , does not require a kernel space process to change permission of page group[1]. A user process can change permission. The important benefits of using MPK are:

- performance
- group-wise control
- per-thread view protection techniques[1]

The first benefit is that MPK uses a PKRU i.e. Protection Key Rights Register to provide no-access, read-only or read/write to pages of particular group. A process which needs to access the page only needs to execute WPKRU, which is non-privileged to update the PKRU, which is faster than traditional methods and does not require context switching and TLB flushing[1]. The effective permission of a particular page is the intersection of page-table permission and PKRU. The second benefit is that as we use the previously unused 4 bits of PTE we can create 16 page groups. All pages having a same key assigned to it belong to same group. It allows user to change permissions of all pages in a group according to the data stored[1]. The third benefit of MPK is that it provides separate PKRU register for each thread.Hence the permissions of each thread are different[1]. It ensures that two threads accessing same address have different permissions and the permission of one thread does not affect other threads permission.

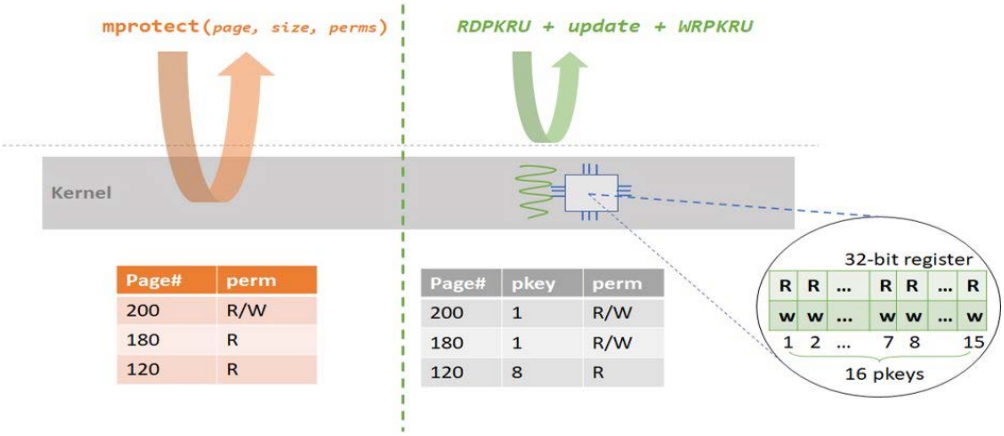


Fig. 2. Difference between MPK and mprotect

A. Hardware Primitives

Intel MPK makes it possible to change permission of pages of a particular group having a same protection key associated by just changing PKRU instead of changing permission for individual pages. Protection key field in page table entry MPK uses previously unused 4 bits of page table to store the page key[6]. Thus we have a provision to create 16 groups of pages i.e. 2**4 groups[6]. MPK provides a single key to every page in memory and multiple pages can have same key associated share same group. Previously a privileged instruction was used to change permission in PTE but the Linux Kernel(after version 4.6) provides pkey-mprotect(),a new system call which can be used to change or assign key to memory pages[6]. Protection key rights register (PKRU) MPK provides PKRU register to manage permissions of each page group. The access permission are defined as no-access, read only or read/write[6]. The value of (Ad,W,D) determines the thread's permission for given group:

00 : read/write
01 : read only
1,x : No access

Per-thread view is provided to each thread by maintaining separate PKRU for each thread[6]. Instruction set MPK has introduced new instructions which are used to manage PKRU.

• We can update the permission in PKRU using instruction WRPKRU.
• We can retrieve the available permissions using the RDPKRU instruction.

The registers EAX,ECX,EDX are used as input by WRPKRU[6]. The new permissions are stored in EAX register which are to be overwritten in PKRU[6]. The registers ECX and EDX are filled with zeros. The same registers are also used by RDPKRU[6]. The EAX register gets the current permissions available in PKRU and overwrites the EDX register with zeros. For proper functioning of RDPKRU the ECX register should also contain zeros. Note: Actual usage of EDX and ECX in not mentioned.[6]

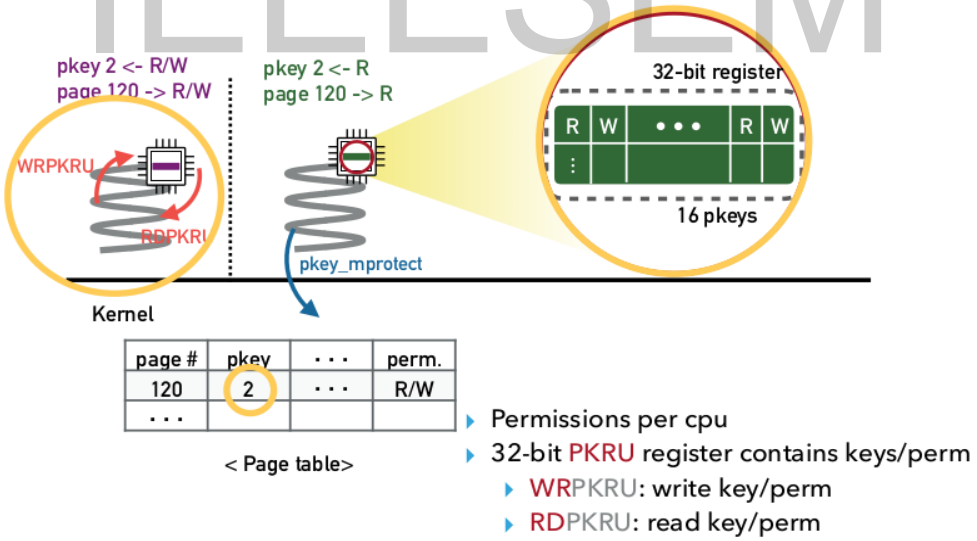


Fig. 3. Underline Implementation

B. Kernel Integration and Standard APIs

The Linux Kernels after version 4.6 has support for MPK[6]. Also glibc after version 2.27 supports MPK[6]. The focus of these kernels is to manage protection and assignment of these keys to particular pages[6]. They provide additional 3 system call :

pkey-free(), pkey-mprotect() and pkey-alloc().

The behavior of previous mprotect() system call is also changed to support execute only memory. Two additional instructions pkey-get and pkey-set are provided by glibc to retrieve and update permissions of protection key.[6]

Name	Cycles	Description
pkey_alloc()	186.3	Allocate a new pkey
pkey_free()	137.2	Deallocate a pkey
pkey_mprotect()	1,104.9	Associate a pkey key with memory pages
pkey_get()/RDPKRU	0.5	Get the access right of a pkey
pkey_set()/WRPKRU	23.3	Update the access right of a pkey

Fig. 4. API's

pkey-mprotect()

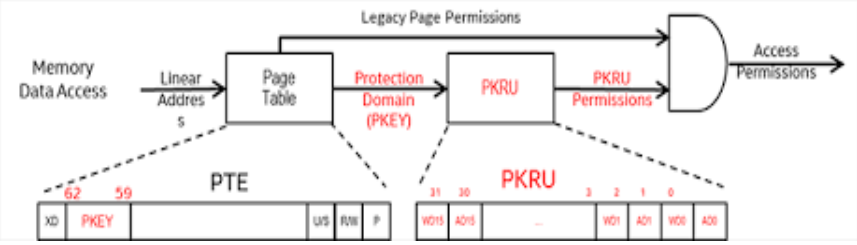
The system call pkey-mprotect() is extension of mprotect() which is used to assign a protection key to the page table entry of particular memory without changing the page protection flag. A newly created page is assigned with protection key zero which is default protection key and is not available to user thread. Hence only 15 protection keys are available to user and only 15 page groups can be created[6].

pkey-free() and pkey-alloc()

The two new additional system calls pkey-free() and pkey-alloc() can be used to de- allocate and allocate protection keys. A 16 bit bitmap is used to track the information about allocated keys. When a user with proper rights calls pkey-alloc(), the kernel assigns a protection key with given permissions which is marked as unallocated form the bitmap entry. Similarly pkey-free() is called by user the key is marked as available in bitmap by the kernel. pkey-protect() also uses the bitmap to ban utilization of allocated keys[6].

C. MPK Access Calculation

The effective permission of the page is calculated based on the memory protection bits of page table and the PKRU register permission. The permission is set to the minimum permission from both the page table and the PKRU register.[2]



III. CHALLENGES

Limited number of physical keys

Intel's MPK uses the previously unused 4 bits of page table, a maximum of only 16 physical keys can be created. The size of PKRU register is 32 bits and each key needs 2 bits to define the permission for a particular

key. Also of these 16 keys the key 0 is reserved and is not available for users. So the user has only 15 possible physical keys[1]. This can cause a problem if the user wants to have more than 15 physical keys. Hence if a user wants to have more than 15 keys it will require additional space to store the keys permission[1].

Protection-key-use-after-free

The pkey-free() system call provided by kernel only marks the freed key as available in bitmap[1]. It does not un-tag the previous pages which were tagged with the given protection key. The next thread which calls pkey-alloc() can find this freed key as available in bitmap and the kernel can allocate the freed key for this thread thereby providing it access to the pages which were tagged with the given freed key[1]. This can give unnecessary access for the thread to pages which it was not supposed to have access to[1]. Handling this problem without fundamental change in the kernel can cause a huge overhead since it will require to traverse each page and check for pkey associated to it and also removing the pkey for specific pages and flushing of TLB's.

mprotect and mpk implementation

Mpk uses PKRU register to set the permission for a given page group. Since the PKRU register is thread local for each thread, change in permission of one thread's PKRU does not affect the PKRU register of another thread[1]. So if we change the permission of PKRU for one thread we cannot guarantee that other threads will have the same permission[1]. This is not the case in mprotect. Mprotect ensures the change in permission of a particular memory page to be the same for all the processes.

IV. CONCLUSION

Intel MPK is thus a useful way to provide group wise memory protection for threads. The instructions RDPKRU and WRPKRU, which are user level instructions, makes it easy to change the permission of pages. However since the instruction is an unprivileged instruction the PKRU register can be controlled by an attacker. Since the performance overhead of MPK is less it can be used to protect the memory pages by just tagging the pages with protection keys.

REFERENCES

[1] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Soft- ware abstraction for Intel Memory Protection Keys (Intel MPK). In Proceedings of USENIX Annual Technical Conference (ATC), 2019.
[2] Pkeys(7) linux programmer's manual, 2018. http://man7.org/linux/man-pages/man7/ Pkeys.7.html.
[3] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Pe- ter Druschel, and Deepak Garg. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In Proceedings of the 28th USENIX Security Symposium (Security), Santa Clara, CA, August 2019.
[4] Mingwei Zhang, Ravi Sahita, and Daiping Liu. eXecutable-Only-Memory-Switch (XOM-Switch): Hiding Your Code From Advanced Code Reuse Attacks in One Shot. In Black Hat Asia Briefings (Black Hat Asia), Singapore, March 2018.
[5] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'20), March 17, 2020, Lausanne, Switzerland. ACM, New York, NY, USA, 14 pages.
[6]https://en.wikipedia.org/wiki/Memory_protection
[7]https://man7.org/linux/man-pages/man2/mprotect.2.html