libppkey: In-Process Memory Isolation for Modern Linux Systems

Chakshu Tandon
Computer Science Department
Rutgers University

***Abstract*—Modern Linux systems extend the inter-process memory isolation guarantees of past systems with active mitigation techniques like Data Execution Protection (DEP) and Address Space Layout Randomization (ASLR). We show these mitigations are not enough to protect applications from all kinds of memory vulnerabilities. Explicitly, a Format String Vulnerability is exploited to leak stack and heap addresses with ASLR enabled. Future hardware support for memory protection such as Intel Memory Protection Keys (MPK) show a lot of promise for in-process memory isolation, however, their benefit for multi-threaded applications is minimal. We present a user-facing library, libppkeys, along with modifications to a recent version of the Linux operating system to better support protection across multi-threaded applications. We are able to show that it is possible to keep the performance benefits of Intel MPK while offering per-process protection domains satisfying the guarantees provided by mprotect(). We observe a $\sim$ 2x latency decrease in inter-thread permission synchronization as compared to mpk_mprotect() and do_pkey_sync() from the libmpk library.**

***Index Terms*—Linux virtual memory, memory isolation, Intel MPK**

I. Introduction

Our lives are becoming increasingly digital, necessitating computer security be addressed more than ever. Modern operating systems have come a long way in terms of security. One of the central abstractions in many commodity operating systems is the isolation between different processes and their memory. One process cannot access the memory of another except in specific cases. This is known to increase the security and dependability of applications. Isolation also gives application developers the freedom to concern themselves about application details rather than interactions with other applications running on the system. These interactions include other processes that run on the same shared hardware contending for machine resources. To access shared hardware resources, the OS exposes a set of ABI to ensure safe and efficient resource multiplexing. In addition to process memory isolation, implementations of Linux Security Modules (LSM) including SELinux [1] and AppArmor further restrict the impact of one application towards another by predefining application policy.

With the various guarantees provided by the operating system for memory isolation between processes, a sort of natural question arises: what set of isolation exists within a processes' address space? This study aims to answer this question by looking at existing techniques and proposes a new solution to increase in-process memory isolation for Linux applications. The result of which is to create memory protection domains which reduce application attack surface and limit the ability for malicious actors and faulty multi-threaded code to go unnoticed.

Traditionally, intra-process memory protection has been provided through techniques such as Data Execution Protection (DEP) which prevents areas of memory from being executed by the processor, Address Space Layout Randomization (ASLR) which randomizes the locations of important user and kernel data structures to prevent many forms of attacks, and base-bounds checking in Intel MPX. The mprotect() system call allows modification of the protection bits in the process page table. More recently, a key-based permission control model named Intel Memory Protection Keys (MPK) has been ported to the x86_64 architecture.

In the next section, we survey the virtual memory subsystem in the Linux kernel (§2). Section 3 discusses recent attacks and other possible memory vulnerabilities (§3). We discuss current memory protection techniques and attack mitigations (§4). The challenges of these existing protections are also evaluated. We dedicate a section to Memory Protection Keys to understand the goals of Intel MPK and the hardware and kernel support that is required (§5). Next, we introduce an abstraction over Intel MPK called libppkeys which addresses a specific issue in the MPK design(§6). A brief evaluation of this system as compared to existing protection techniques is then presented (§7).

II. Background

Like any hardware resource, memory is a scarce and contended resource. Most operating systems provide virtual memory as a subsystem which is responsible for providing an isolated view of physical memory. The central data structure in Linux behind the virtual memory support is called the page table. The page table provides translations between physical system memory and virtual application address space. The smallest unit of memory recognized by the operating system is called a page. The role of the page table is then to provide a map between physical pages and virtual application pages. It ensures that two processes that do not share memory are allocated a disjoint set of physical pages. Importantly, virtual memory must be contiguous to transparently enable the application's illusion of having the entire system memory to itself. This section will discuss the design of the page table,

how it supports the role of virtual memory, and the memory protections it offers in relation to the hardware memory management unit (MMU).

In a 32-bit system, there exist up to 2^{32} addresses or roughly 4 GB of memory. A page varies from system to system but is generally 4096 bytes. Since the page table requires a page table entry (PTE) per page in the virtual address space, nearly one million entries per process would be needed in a "flat" page table design. A 64-bit system allows 2^{64} system addresses meaning 2^{52} PTEs are required per process. A flat page table structure simply has too many entries and does not work for modern systems. Fortunately, it is noted that most applications maintain sparse address spaces and thus a hierarchal page table design is appropriate. Ahn et al. discuss a case in which hypervisors are better suited using flat nested page tables to reduce unnecessary memory references for nested walks [2]. In most cases, using a multi-level hierarchal page table generally saves memory at the expense of multiple memory lookups.

In Linux, four levels of hierarchy are used which are independent of the hardware page-walk. The highest level PML4 points to various page directories (PGD) which in turn point to multiple middle directories (PMD) which finally point to page table entries (PTE) containing the virtual-to-physical mapping. For our purposes, we are only interested in the PTE as they contain the mapping and protection bits. Figure 1 shows the layout of the PTE and the various bits contained within. Bit 1 contains the first protection offered by the hardware. If the entry contains a zero, writes to the page are disabled. Bit 2 indicates if a page belongs to a user or the operating system. A user attempting to access a supervisory page will fault to the operating system. Bits 12 to M-1 contain the physical page frame number (PFN). Bits 59 to 62 contain the protection key used in Intel MPK. Finally, the XD/NX bit (63) indicates that the page supports execution. Later, we will discuss the memory protection features the page table structure provides in conjunction with hardware and operating system support (§4). The x86 specifications provide further detail about each of the bits contained in the page table entries [3].

There are several assumptions made by Linux regarding virtual memory. First, each process is represented by a task_struct, which contains reference to the address space in an mm_struct. By virtue, this address space exists globally to the process to which it belongs. Therefore, all processors see the same page entries regardless of execution context. This assumption is deeply rooted within the kernel and severely restricts the ability to keep a logical page table per protection domain. This would allow different threads to specify which memory is shared for performance and simplicity and page which are exclusive for security and fault-containment. In general, the assumption of a single address space per process leads to the conclusion that it is the role of the operating system to mostly isolate memory between applications. Evident by the countless application vulnerabilities discovered daily, an expectation of secure-by-default and in-process memory protection requires further support from both the operating system and the hardware.

III. Motivations

Applications built today have the expectation to scale far beyond previous levels. Developers have largely adopted distributed "microservice" architectures. As a result, the complexity of applications has skyrocketed. Security, however, has not kept up with the growth in demand and many serious vulnerabilities are being discovered as a result. Furthermore, virtual machines and containerization have largely enforced the concept of a single-purpose execution environment. Inter-process memory safety at higher tiers of the application stack are less important than in-process memory isolation. Here, we discuss recent attacks and other possible memory vulnerabilities. The following section will demonstrate that current mitigation techniques are insufficient to handle such vulnerabilities.

A. OpenSSL Heartbleed

Many in the industry will be able to recall CVE-2014-0160, known commonly as Heartbleed. This bug was a serious vulnerability in the popular OpenSSL cryptographic library which leaked protected SSL/TLS encryption secrets due to missing bounds checking in user-provided input. Attackers were able to read up to 64 KB of surrounding memory by passing a buffer that had a smaller length than indicated. OpenSSL would copy memory into the buffer past the actual data to return to the user completing the attack. The significance of this attack is that it relied on primitive techniques and affected code that had been reviewed by the open-source community countless times. With a secure-by-default design implemented in the operating system, many such attacks can be avoided. A similar type of exploit known as a format string vulnerability is explained below.

B. Format String Vulnerability (FSV)

A format string vulnerability (FSV) occurs in programs that pass unvalidated user-input or other malformed format strings into calls such as printf(), scanf(), and fprintf(). To understand the attack, a simple example is taken. The following, printf("%d\n", foo), places the string "%d\n" and the variable "foo" on the user stack before proceeding with the function call. Printf will parse the format string and start reading variables off the application stack. Since the first agument is provided as a string, it is difficult to statically check if the number of arguments does in fact match with the format specifier provided. For example,

printf("%x %d %p %s");

reads several parameters from the stack when none were provided. Specifically, it will read an unsigned hex integer, an integer, a pointer, and a null-terminated character string possibly running into the function context of a previously called function. If the first argument is provided by an untrusted user, this has the potential of leaking sensitive stack data such as return pointers, application secrets, and user data from

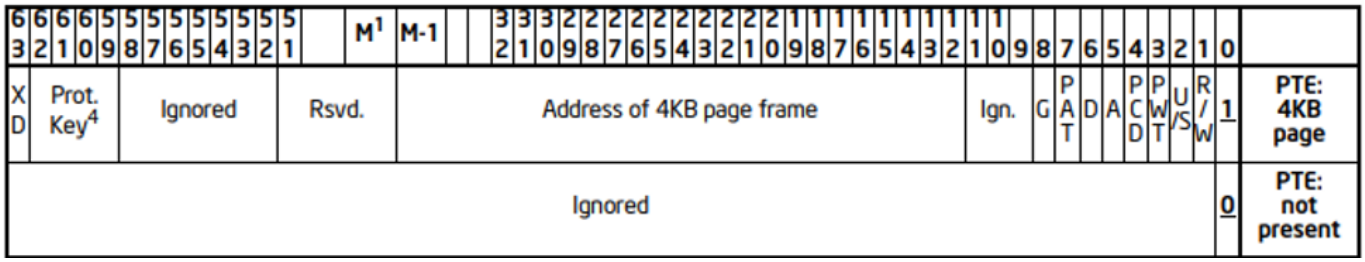

Fig. 1. Format of a IA-32 Page Table Entry mapping to a 4-KByte page. Bits 12 to M-1 contain the virtual-to-physical page translation. Several bits are used to store additional details about the page including access bits, memory protection key (prot. key), and data execution prevention bit (XD/NX).

previous stack frames. Newsham shows how this attack can even be used to overwrite stack data leading to code execution and application corruption [4]. Current versions of gcc will give a compile-time warning, however, it is not known if all compilers support this feature. Later, we show that a modern Linux system with ASLR enabled exhibits this vulnerability and application secrets are leaked at runtime (§4).

IV. Current Techniques

A. Data Execution Protection (DEP)

Data Execution Protection labels different virtual area mappings (e.g. code, heap, stack, dynamic libraries) with various protection bits contained in the page table (§2). The XD/NX bit is common among modern processors and enables or disables code execution for pages outside code regions. If the NX bit is set to 1 for a particular memory page referenced by the instruction pointer, a processor will call the registered operating system handler which terminates the offending process. Since code regions are set to read-only by default, this has the effect of mitigating many types of attacks involving buffer overflows and stack corruptions. DEP, however, does not provide memory isolation and cannot protect against a common vector of attack called return-oriented programming [5] which uses existing segments in program and/or library code known as gadgets to modify control flow.

B. Address Space Layout Randomization (ASLR)

Traditionally, attacks involving buffer overflows and other memory vulnerabilities result from a known memory layout. Address Space Layout Randomization is a technique employed by modern operating systems to randomize the start location of several important data structures in memory to render such attacks useless. To avoid leaking potentially sensitive information, the stack and heap segments are placed at random locations in a process address space. While a thoughtful measure, security through obscurity is a flawed philosophy. In addition to limited entropy bits (Figure 2), a single leak of information is enough to circumvent the mitigation. Table 1 shows the number of unique addresses witnessed during 500,000 invocations of the benchmark. A dedicated adversary may be able to perform a brute force attack bypassing such mitigation. ASLR is controlled using:

/proc/sys/kernel/randomize_va_space

TABLE I
ASLR: Distinct Addresses in 500,000 Invocations

	code	data	heap	mmap	stack
x86_32	1	1	8192	256	347319
x86_64	1	1	8192	418298	418522

0: No randomization

1: Conservative randomization. Shared libraries, stack, mmap(), VDSO and heap are randomized

2: Full randomization. In addition to elements listed in the previous point, memory managed through brk() is also randomized.

Several challenges exist when using ASLR as a memory protection technique. In the implementation of ASLR in Linux 3.7, a weak get_random_int_hash() is used which relies on the MD5 PRNG cipher. Yoo et al. demonstrate the cipher is known to be vulnerable to a generic entropy-search attack [6]. Secondly, we show that an FSV bypasses the ASLR protections. The attack is possible since it does not require knowledge of the exact location of the stack frames of victim functions. In memory, the stack is a contiguous structure with a known format. The FSV exploits locality of adjacent stack frames. The following code provides some insight.

```
void createSecret(char *secret) {
    sprintf(secret, "s3cr3t_key");
}
void vuln(char *input) {
    printf(input);
}
int main() {
    char input[50];
    char *secret = malloc(25);
    createSecret(secret);
    fgets(input, 50, stdin);
    vuln(input);
}
```

Here the vulnerability lies in the naked printf() call. The function takes user input from fgets() directly without sanitization. The following FSV exploit results in a stack leak of the char *secret heap address which is used to print the value.

```
> echo $(python -c
    'print " %p" * 12 + "|| %s ||"'
  ) | ./fmt_str_vuln
```

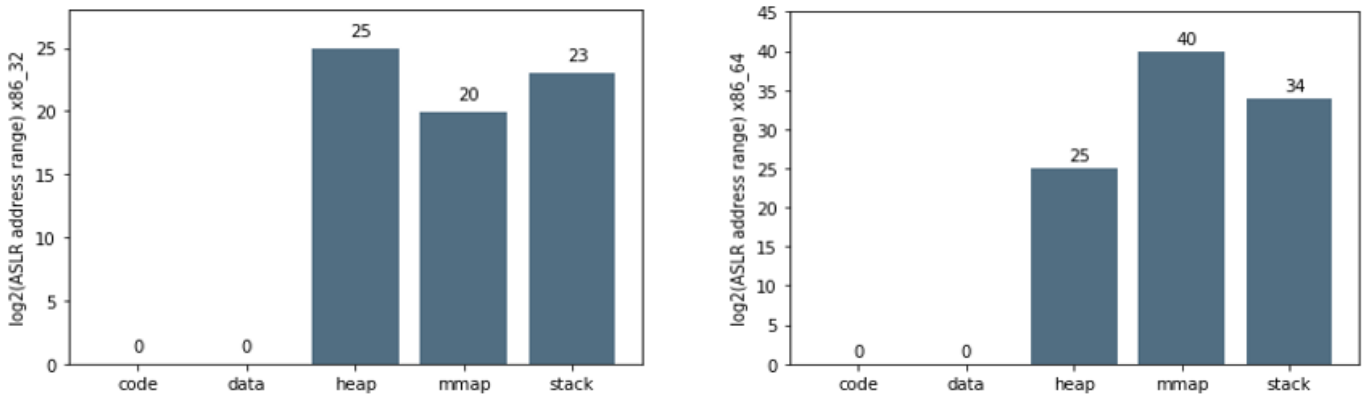

Fig. 2. Bits of ASLR entropy across different virtual area mappings. A log transform is taken on the observed addresses to get bits of randomness. A total of 500,000 invocations are captured. (Left) x86_32 machine with only 20 bits of entropy in mmap region corresponding to approximately 1 million address range. (Right) x86_64 machine with significantly higher entropy in stack and mmap region with low entropy in heap region.

The printf() stack frame expects there to be twelve pointers followed by null-terminated string. Since no arguments are provided, it starts reading variables from vuln()'s stack frame followed by main() which contains the address *secret. The "%s" follows the pointer and displays the value "s3cr3t_key." The following sections demonstrate mitigations to an attack such as FSV using memory protection domains.

C. mprotect() System Call

The mprotect() system call sets protection on a region of memory. It takes an address and length along with a combination of protection flags (PROT_NONE, PROT_READ, PROT_WRITE). It applies the protection to all pages in the range by setting the appropriate bits in the page table entries corresponding to the pages. Further, the mprotect() call is synchronous causing the process to trap and produce faults for any accesses after the call returns. It is this behavior that allows the protection to be applied across different execution contexts in multi-processor environments. Unlike Intel Memory Protection Keys (MPK), the mprotect() call is process-global.

One of the biggest shortcomings of mprotect() is that it lacks the performance characteristics to be used in high-frequency updates to region of memory. As Corbet states, this is beneficial for "handling of sensitive cryptographic data. A network-facing daemon could use a cryptographic key to encrypt data to be sent over the wire, then disable access to the memory holding the key (and the plain-text data) before writing the data out. At that point, there is no way that the daemon can leak the key or the plain text over the wire; protecting sensitive data in this way might also make applications a bit more resistant to attack. [7]" Further, multi-threaded applications can use fast updates to memory regions to prevent and detect "stray" write operations that are difficult to debug. Park et al. show applications for securing management of OpenSSL TLS key material, the code cache of JIT compilers, and performance sensitive key-value stores such as Memcached [8].

Unfortunately, updates to page-table bits are very expensive. They require a trap into the kernel (100-500 cycles), updates to hundreds or thousands of page table entries in memory, and equally many Translation-Lookaside Buffer (TLB) cache invalidations across multiple processors. Figure 3 shows the cost of an mprotect() on 1000 pages compared to other methods explained below. We can see mprotect() is several times as expensive as other techniques. In the following section, we will discuss Intel MPK and the performance benefits it has over the traditional mprotect() system call.

V. Intel Memory Protection Keys (MPK)

Compared to software, hardware changes far less frequently and at greater organizational cost. New features generally get added to software before they are implemented more efficiently in hardware. This is the case with Intel Memory Protection Keys (MPK). It builds upon features existing in modern operating systems to increase the performance and flexibility of memory protection.

A. Mechanism

Instead of updating the page protections directly in the page tables which cause all of the problems explained previously in the mprotect() section (kernel trap, TLB invalidations, etc.), it assigns each page one of 16 keys[1] per process using four previously reserved bits in the PTE. A new register, accessible from userspace, protection key rights for userspace (PKRU), maintains two permission bits per key. At runtime, the hardware checks each access against the intersection of the per-thread (hardware hyperthread) PKRU register and traditional page table protection bits. Updates to an entire group of pages, which may or may not be virtually contiguous, need not go through the kernel (e.g. page table updates), and do not update page table entries causing expensive TLB shootdowns. Two new un-privileged hardware instructions, RDPKRU and WRPKRU, allow reading and writing to the PKRU register

[1]There are only 15 protection domains available to user applications. Key zero is used for the default protection domain.

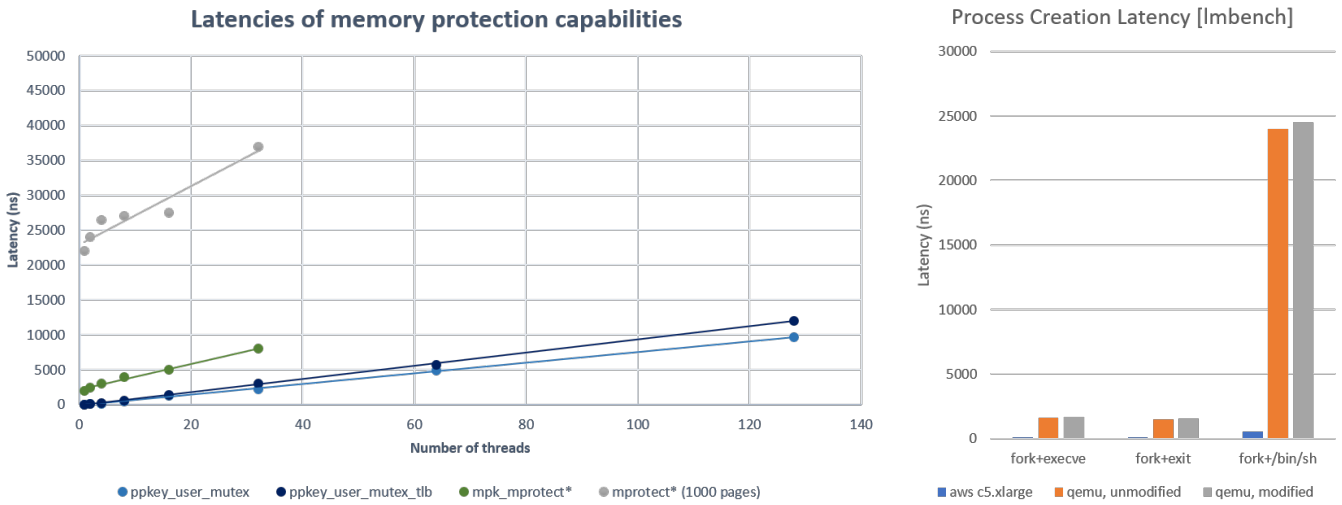

Fig. 3. (Left) Latencies of various memory protection capabilities compared to number of application threads. ppkey_user_mutex and ppkey_user_mutex_tlb correspond to the latency of acquiring a mutex, updating the protection bits, and releasing the lock. The gray line represents a mprotect() of a region consisting of 1000 pages. The green line are results gathered by syncing PKRU values in the libmpk paper. (Right) Results of the lmbench lat_proc tool to benchmark process creation latency between our modified and unmodified kernel. Process creation overhead increases by less than 2%.

TABLE II
LATENCY OF INTEL MPK

Register	General (ns)	PKRU (ns)	Δ (%)
Read	1.64	3.95	141
Write	2.22	14.28	543
PTE Update	mprotect (ns)	pkey_mprotect (ns)	Δ (%)
	399.65	399.07	-0.15

with a latency between $\sim$ 5 and 15 ns (Table 2). The apparent overhead incurred by RDPKRU and WRPKRU as compared to general-purpose registers can most likely be attributed to flushes to the processor pipeline. This is required in order to maintain correctness during branch-prediction and speculative execution. Previously, quite difficult, this mechanism allows two threads that share the same address space to have different access rights to the same page. This can be useful for example if two user requests are handled in separate threads. Each request can be protected with the same key yet have different access rights.

B. Linux Integration

In addition to the reading and writing to the PKRU register, Linux has added support for three additional system calls since v4.9. Glibc has further added library support since v.2.27. pkey_alloc() and pkey_free() allocate and de-allocate keys respectively. Internally, a bitmap is used by the kernel to identify keys which have been given to the process. pkey_free() releases the key from the bitmap, however, does not change or invalidate the protection key bits in the page table leading to a *protection-key-use-after-free* vulnerability [8]. If the freed key is reassigned, then as a side-effect, the old protection group also gets included into the new group. Changing the page table entries is prohibitively expensive, however, is needed to resolve this issue.

The pkey_mprotect() system call is introduced to maintain a backwards-compatible ABI with the mprotect() call. It is functionally quite similar and is used to update the key of a set of contiguous pages within the page table. Note this call suffers from the same performance bottlenecks as mprotect(). Applications should avoid changing the key of a page through pkey_mprotect() and rather group similar pages which are updatable through the PKRU register.

Effort was taken to understand the relationship between protection domains using Intel MPK and existing memory mapping policy in the Linux kernel. Linux uses Virtual Memory Areas (VMA) to represent distinct areas within an address space (e.g. code, heap, stack). The VMA is a contiguous area of virtual memory which holds information about the set of pages that it encompasses including the protection and replacement policies. In a traditional mprotect() call, VMAs may be split to encompass the change in permission. Two VMAs may be merged if they are contiguous and have the same permissions. The behavior of the pkey_mprotect() call in regard to this matter does not seem to be written down but has a predictable pattern. The same rules apply in terms of mprotect(), except that two VMAs with different pkeys will not be merged and/or split accordingly. This rule does not take into account the value of the pkeys, rather the key itself which significantly reduces the cost of VMA creation and deletion in high-frequency protection updates.

C. PKRU Inter-Thread Sync

Here are the biggest issues with the implementation of Intel MPK. The protection for a given domain is maintained in the thread-local un-protected user-accessible PKRU register. This causes two main issues: (i) Updates to a given thread's

PKRU are not made coherent with other threads, (ii) A thread can maliciously update its own PKRU to gain access to any page belonging to the process. To emulate the per-process protection guarantees provided by the mprotect() system call, it is important that two threads are able to consistently see the protection of a given page. This goes against the design of MPK. In the next section, we show that it is possible to emulate this behavior on existing machines with nearly the same performance characteristics. Secondly, updates to the thread's PKRU are unprotected and unsupervised. To increase performance, designers opted for user-level updates to the PKRU register instead of an expensive system call for protected read/write instructions. While our design does not explicitly consider this case, updates are still made to user-accessible memory locations, an adapted design which considers a higher-protection trusted thread to handle protection updates could be used.

In order to address the first issue, Park et al. introduce do_pkey_sync() to sync the PKRU register across threads. They do this in a lazy manner using Linux kernel hooks which get called just before a thread is about to be rescheduled [8]. In that case, the kernel will check if any updates have been made in any of the other processors before returning to user-mode. While the latency of kernel hooks is greatly less than the latency of an mprotect() call (Figure 10 in [8]), the alternative design we present has a significantly reduced overhead. We will see the cost of a TLB shootdown of a single ppkey page is less than the invocation of kernel hooks across many application threads.

VI. Our Design

Our solution to this problem follows from the observation that updates to the numerous PTEs are expensive. They require kernel invocation, numerous memory accesses, and TLB shootdowns in multi-processor configurations. Instead, a group-wise protection policy which can be modified without a kernel crossing enables applications to make frequent protection updates. To satisfy the per-process protection requirement missing from the Intel MPK implementation, we consider a hypothetical processor containing a PPKRU register. This register references a ppkey structure in memory which has the same semantics as the PKRU register in Intel MPK. All threads belonging to a process reference the same address in a protected PPKRU register. At runtime, the hardware checks the intersection of three protection signifiers; (i) The PTE protection bits, (ii) The PKRU register, and (iii) The ppkeys structure located at memory referenced by PPKRU. In effect, this allows both a per-process and thread-local view of memory that satisfies the performance benefits provided by Intel MPK and the coherence guarantees provided by the mprotect() system call. We utilize Intel MPK to simulate the behavior of such a system and the results are analyzed. Figure 4 shows the system as described above.

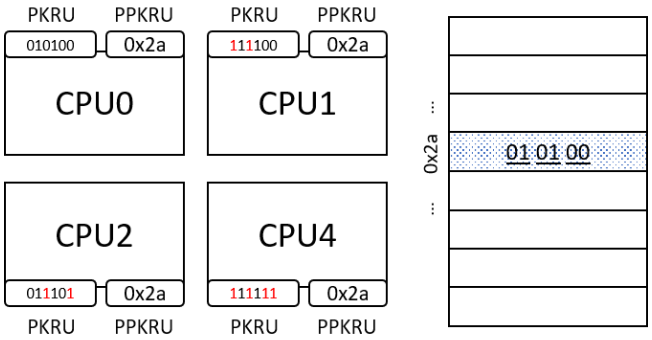

Fig. 4. libppkey overview. Here four processors are shown each containing a PKRU and PPKRU register. The red bits in the PKRU registers show the thread-local updates as compared to CPU0. The PPKRU register on all four CPUs points to memory location 0x2a which additionally contains a set of ppkeys checked by the hardware during runtime satisfying the per-process memory protection requirement.

A. Kernel Modifications

To enable support for ppkeys, some modifications were made to a recent Linux kernel (v.5.4-rc2). We have decided to allocate the ppkey region inside the kernel to increase portability and adoption with various system run-times. Like the vDSO region, the ppkey is installed as a special read/write mapping into every process address space during process creation. This occurs during load_elf_binary() in fs/binfmt_elf.c. First, the kernel finds a free memory page and registers it as a special mapping. A signal handler is also specified to handle page faults. Since the mapping is needed immediately, a virtual zeroed memory page is allocated and installed into the page table at the address of the special mapping. Then just before returning to user-mode it must set the PPKRU register. In our implementation, since no such physical hardware register exists, we confound it with the R15 general purpose register. While the initial intention was to confound with one of the SIMD registers, the former turned out to be easier to implement.

B. libppkey User Library

The libppkey user library is responsible for maintaining the protection bits stored at the ppkey region mapped into the process address space by the kernel. Observing the /proc/-self/maps file should reveal an entry labeled ppkey containing a single page mapped with read/write permissions. The address of the beginning of this page is stored in the R15 general purpose register which is read by libppkey to make updates to the ppkey area. To do so, it exposes two function calls int ppkey_read(int key) and int ppkey_write(int key, unsigned int prot).

Of the two, ppkey_write() is slightly more interesting. It reads the location of the ppkey region using ppkey_get(). Next a ppkey is generated using bit operations on the key and prot values that were passed as arguments. Currently, the hardware does not support applying protections based on the value of the ppkey vector, therefore, writing the ppkey

to memory is not sufficient. Instead, we utilize the existing PKRU register to write the ppkey protections across all threads via a signal handler. Actually, we store the intersection of PKRU and ppkeys to simulate hardware support for ppkeys. In hindsight, this was a poor decision and updating the PKRU should have been done in the kernel. When a signal is created using pthread_sigqueue, threads that registered with the signal handler are suspended and a *new* thread is created for the signal handler. Updates to the PKRU from the signal handler are not carried to the register file of the child thread. A siginfo_t structure is passed to the signal handler which contains the context of the previously scheduled thread. In theory, this should contain the PKRU state needed to context switch back to the previous thread. We could update this state on the user stack, however this structure seems to be missing the PKRU register in current versions of libc. In the kernel, a watchdog thread could listen for changes to the ppkeys vector and apply updates to each of the PKRU registers. Briefly attempting the above, the CPU running the watchdog would softlock and waste CPU resources. More work is needed to implement a working proof-of-concept utilizing the PKRU register. In theory, it should be entirely possible to emulate per-process ppkey behavior with Intel MPK.

VII. Analysis

A. Hardware

Analysis was conducted on a custom 2nd generation Intel Xeon Scalable Processor (Cascade Lake) with a sustained all core Turbo frequency of 3.6GHz and single core turbo frequency of up to 3.9GHz. On Amazon Web Services (AWS), this corresponds to the c5.large instance type with 2vCPU and 4 GB memory. A machine nearly identical in specs on the Google Cloud Platform (GCP) did not have the necessary support for Intel MPK. It is possible there are kernel or hardware modifications made by the cloud providers which enable or disable support for the feature. More exploration is needed to identify the root cause of this behavior.

B. ASLR and FSV

Figure 2 shows the number of bits of entropy in 32-bit and 64-bit ASLR machines. With just over one million possible address locations, a brute force attack is certainly feasible. In many cases, even this is not required. Exploits such as the Format String Vulnerability (FSV) attack (§4) abuse the contiguity of the stack data structure to leak heap addresses with sufficient accuracy. Many times, Data Execution Protection (DEP) can be bypassed maliciously by using existing code within the application or linked libraries. A strange observation is made on 64-bit machines. While the mmap and stack produce 35 to 40 bits of entropy, the heap is again limited to 25 bits of entropy. Further work may investigate this phenomenon.

C. mprotect()

In Figure 3, we see the latency of a well-known system call mprotect() on a region of memory 4 MB in size (1000 pages). For our application, this is quite a conservative region. We may be further interested in regions several GB in size. In any case, there is a clear upward relationship between the number of threads and the latency of mprotect(). This is due to the number of TLB shootdowns that must occur for each execution unit. We also note that the latency is significantly higher than any of the other capabilities presented in this paper.

D. Intel MPK and mprotect()

Intel Memory Protection Keys (MPK) reduce the overhead of the mprotect() call significantly by writing changes to page protection in a user accessible register PKRU. Table 2 shows the cost of reading and writing to general-purpose registers and the PKRU register. Compared to thousands of cycles required by mprotect(), reading and writing to the PKRU register takes less than 15 ns. The latency is much higher than reading or writing to a general purpose register for the reason mentioned above. Instead of incurring TLB shootdowns, all processor pipelines must be flushed in order to observe consistent and secure memory accesses. We further see that updating the keys in the page table is as expensive as changing the PTE protection bits. There is less than a 0.15% difference which is expected as they both rely on the same do_mprotect_pkey() in the kernel.

E. Intel MPK and libmpk

Again in Figure 3, we see the synchronization of the PKRU register as enabled in the libmpk library. It uses a lazy approach using a Linux feature called kernel hooks. It is noted that the latency of synchronizing the PKRU of 32 threads is significantly lower than an mprotect() on the same region. We further note that for a single thread, the latency of mpk_mprotect() with do_mprotect_pkey() is much higher than values found in Table 2. This may speak to the overhead caused by such synchronization. Secondly, at 32 threads, the latency is nearly 10,000 ns which is much higher than expected. We show using libppkeys, that we can reduce the overhead due to synchronization to provide per-process memory protection.

F. libppkey

In (§6), we present our approach to in-process memory protection building on the ideas of Intel MPK. We utilize a per-process memory region ppkeys referenced by a hypothetical hardware register PPKRU. We simulate the behavior of PPKRU by setting the values of the PKRU thread by thread. Figure 3 shows that even with the required locking and TLB invalidation of the ppkey page, performance of ppkeys grows linearly to the number of application threads. At 32 threads, the cost to protect 1000 pages is nearly half that of the libmpk implementation. Nearly twice as many threads are needed to be synchronized in order to achieve the same latency as above. Also in Figure 3, we see less than 2% increase to process creation latency due to ppkey_setup().

VIII. Conclusion

It is demonstrated that existing attack mitigation techniques such as Data Execution Prevention (DEP), Address Space Layout Randomization (ASLR), and mprotect() are insufficient for the needs of many applications. We have shown that simple extensions to Intel Memory Protection Keys (MPK) can provide per-process along with thread-local memory protection at nearly the same efficiency as MPK. We can use per-process memory protection keys to emulate the behavior of mprotect() without incurring the overhead of kernel crossings, TLB shootdowns, and numerous memory accesses. Process creation latency overhead is less than 2% and synchronization latency is reduced by 50%. A practical proof-of-concept is lacking due to poor choice in selecting signal handlers in user-mode to emulate ppkey updates. In the future, it is possible to use a watchdog thread in the kernel to update PKRU register on the user's behalf.

Acknowledgment

The author would like to thank Sudarsun Kannan and members of the Rutgers Systems Research Lab without which this work would not have been possible.

References

[1] T. Jaeger, R. Sailer and X. Zhang, "Analyzing integrity protection in the SELinux example policy," SSYM'03 Proceedings of the 12th conference on USENIX Security Symposium, 2003.

[2] J. Ahn, S. Jin and J. Huh, "Revisiting hardware-assisted page walks for virtualized systems," 2012 39th Annual International Symposium on Computer Architecture (ISCA), 2012.

[3] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," 11 November 2019. [Online]. Available: https://software.intel.com/en-us/articles/intel-sdm.

[4] T. Newsham, "Exploiting Format String Vulnerabilities," 2000.

[5] R. Roemer, E. Buchanan, H. Shacham and S. Savage, "Return-Oriented Programming: Systems, Languages,," ACM Transactions on Information and System Security, 2012.

[6] D. Yoo, Y. Kim, T. Yoo and Y. Yeom, "Analysis of the Random Number Generator Using MD5 PRNG," Advanced Science and Technology Letters, 2017.

[7] J. Corbet, "Memory protection keys," lwn.net, 2015. [Online]. Available: https://lwn.net/Articles/643797/.

[8] S. Park, S. Lee, W. Xu, H. Moon and T. Kim, "libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)," Proceedings of the 2019 USENIX Annual Technical Conference, 2019.