libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK)

Soyeon Park Sangho Lee[†] Wen Xu Hyungon Moon[*] Taesoo Kim

Georgia Institute of Technology
[†]*Microsoft Research*
[*]*Ulsan National Institute of Science and Technology*

Abstract

Intel Memory Protection Keys (MPK) is a new hardware primitive to support thread-local permission control on groups of pages without requiring modification of page tables. Unfortunately, its current hardware implementation and software support suffer from security, scalability, and semantic problems: (1) vulnerable to protection-key-use-after-free; (2) providing the limited number of protection keys; and (3) incompatible with `mprotect()`'s process-based permission model.

In this paper, we propose `libmpk`, a software abstraction for MPK. It virtualizes the hardware protection keys to eliminate the protection-key-use-after-free problem while providing accesses to an unlimited number of virtualized keys. To support legacy applications, it also provides a lazy inter-thread key synchronization. To enhance the security of MPK itself, `libmpk` restricts unauthorized writes to its metadata. We apply `libmpk` to three real-world applications: OpenSSL, JavaScript JIT compiler, and Memcached for memory protection and isolation. Our evaluation shows that it introduces negligible performance overhead (<1%) compared with the original, unprotected versions and improves performance by 8.1× compared with the secure equivalents using `mprotect()`. The source code of `libmpk` is publicly available and maintained as an open source project.

1 Introduction

Maintaining and enforcing memory access permission is an important duty of OSes and CPUs. Traditionally, they have used page tables to specify whether processes have rights to read from, write to, or execute specific memory pages. OSes can change access permission by updating page-table entries (PTEs) and flushing corresponding translation lookaside buffer (TLB) entries to reload them. In addition to page tables, some CPUs, e.g., ARM [5] and IBM Power [18], allow OSes to maintain the permission of a *page group* together by assigning the same *key* to the correlated pages and controlling the key's permission. To modify the permission of the page groups, OSes should, on behalf of the process, change the permission of the corresponding registers.

Recently, Intel deployed a similar key-based permission control, called Intel Memory Protection Keys (Intel MPK) [20], that allows a *userspace* process to change the permission of the page groups. MPK has three key benefits over page-table-based mechanisms: (1) performance, (2) group-wise control, and (3) per-thread view. First, MPK utilizes a protection key rights register (`PKRU`) to maintain the access rights of individual keys associated with specific pages: read/write, read-only, or no access. Processes only need to execute a non-privileged instruction (`WRPKRU`) to update `PKRU`, which takes less than 20 cycles (§2.3) and requires no TLB flush and context switching. Note that `PKRU` and page-table permissions cannot override each other, so the *effective permission* is the intersection of both.

Second, MPK can change the access rights of up to 16 different *page groups* at once, where each group consists of pages associated with the same key[1]. This group-wise control allows applications to change access rights to page groups according to the types and contexts of data stored in them (e.g., per-session data of a web server).

Third, MPK allows each thread (i.e., each hyperthread) to have a unique `PKRU`, realizing per-thread memory view. Accordingly, even if two threads share the same address space, their access rights to the same page can be different.

Although MPK is a promising primitive in concept, its current hardware implementation as well as standard library and kernel support suffer from three problems: (1) *security*, (2) *scalability*, and (3) subtle *semantic* differences, hindering its broader adoption. First, we found that MPK suffers from the *protection-key-use-after-free* problem. The Linux kernel provides two system calls, `pkey_alloc()` and `pkey_free()`, to allocate and de-allocate protection keys, respectively. During key de-allocation (`pkey_free()`), however, it does not invalidate pages associated with a de-allocated key, resulting in ambiguity when the de-allocated key is re-allocated and assigned to different pages later.

Second, MPK fails in scaling because `PKRU` can manage only up to 16 protection keys because of its hardware limitation. When an application tries to allocate more than 16 protection keys, `pkey_alloc()` simply fails, implying that the application itself should implement its own mechanism to

[1]The default group (0) has a special purpose, so only 15 groups are available for general uses.

multiplex these protection keys.

Third, the semantic of MPK is different from the conventional `mprotect()`, i.e., thread-view versus process-view, which results in potential security and performance problems. For example, the Linux kernel implements an execute-only memory with MPK by disabling read access through `PKRU` and allowing execution through a page table: `mprotect(addr, len, PROT_EXEC)`. Although this feature is invoked via `mprotect()`, it only changes the `PKRU`'s permission of the calling thread, meaning that other threads sharing the same address space can still read the execute-only memory. In other words, it is non-trivial to apply MPK securely and efficiently to legacy applications that rely on a process-level memory permission model.

In this paper, we propose `libmpk`, a secure, scalable, and semantic-compatible software abstraction to fully utilize MPK in a practical manner. In particular, `libmpk` implements (1) *protection key virtualization* to eliminate the protection-key-use-after-free problem and to support the unrestricted number of memory page groups, (2) *lazy inter-thread key synchronization* to selectively ensure per-process semantics with MPK, allowing us to substitute `mprotect()` in an efficient and compatible manner, and (3) *metadata integrity* to ensure the integrity of the mapping information while minimizing the number of system call invocations. `libmpk` consists of a userspace library mainly for efficient permission change and a kernel module mainly for synchronization and metadata integrity.

To show the effectiveness and practicality, we apply `libmpk` to three real-world applications: OpenSSL library, JavaScript just-in-time (JIT) compiler, and Memcached. First, we modify the OpenSSL library to create secure memory pages for storing cryptographic keys to mitigate information leakage. Second, we modify three JavaScript JIT compilers (i.e., SpiderMonkey, ChakraCore, and v8) to protect the code cache from memory corruption by enforcing the W⊕X security policy. Third, we modify Memcached to secure almost all its data, including the slab and hash table, whose size can be several gigabytes. The evaluation results show that `libmpk` and its applications have negligible overhead (<1%). Furthermore, `libmpk` is 1.73–3.78× faster than `mprotect()` when changing the permission of 1–1,000 pages at the view of a process, and, especially, the throughput of Memcached with `libmpk` is 8.1× higher than that of Memcached with `mprotect()`.

We summarize the contributions of this paper as follows:

- **Comprehensive study.** We study the design, functionality, and characteristics of Intel MPK in detail. We identify the critical challenges of utilizing MPK in terms of security, scalability, and semantics.
- **Software abstraction.** We design and implement `libmpk`, a software abstraction to fully utilize MPK. The protection key virtualization, metadata protection, and inter-thread key synchronization of `libmpk` allow applications to effectively overcome the three challenges.
- **Case studies.** We apply `libmpk` to OpenSSL library, JavaScript JIT compiler, and Memcached to show its effectiveness and practicality. `libmpk` secures them with a few modifications and negligible overhead.

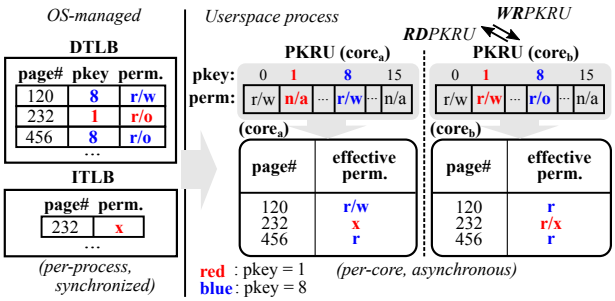


Figure 1: An example showing how MPK checks the permission of a logical core (hyperthread) on a specific memory page according to PKRU and page permissions. The intersection of the permissions determines whether a data access will be allowed. An instruction fetch is independent of the PKRU.

2 Intel MPK Explained

In this section, we describe the hardware design of Intel MPK and current kernel and library support. Also, we check the performance characteristics of MPK to show its efficiency.

2.1 Hardware Primitives

Intel MPK updates the permission of a group of pages by associating a protection key to the group and changing the access rights of the protection key instead of individual memory pages (Figure 1).

Protection key field in page table entry. MPK assigns a unique protection key to a memory page group to update its permission at the same time. MPK exploits the previously unused four bits of each page table entry (from 32nd to 35th bits) to store a memory page's corresponding key value. Thus, MPK supports up to 16 different page groups. Since only supervised code can access and change PTEs, the Linux kernel (from version 4.6) starts to provide a new system call, `pkey_mprotect()`, to allow applications to assign or change the keys of their memory pages (§2.2).

Protection key rights register (`PKRU`). MPK uses the value of `PKRU` to determine its access right to each page group. Two bits representing the right are *access disable* (AD) and *write disable* (WD) bits. The value of (AD, WD) represents a thread's permission to a page group: read/write $(0,0)$, read-only $(0,1)$, or no access $(1,x)$. `PKRU` exists for each hyperthread to provide a per-thread view.

Instruction set. MPK introduces two new instructions to manage the `PKRU`: (1) `WRPKRU` to update the protection information of the `PKRU` and (2) `RDPKRU` to retrieve the current protection information from the `PKRU`. `WRPKRU` uses three registers as input: the `EAX` register containing new protection

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`

Ref. `mprotect()`: 1,094.0 / MOVQ (`rbx` to `rdx`): 0.0 / MOVQ (`rdx` to `xmm`): 2.09

Table 1: Overhead of MPK instruction, system calls, and standard library APIs. *ref* shows the overhead of `mprotect()` and normal register move instructions for comparison. We averaged 10 runs of microbenchmarks, where each one executes individual instruction, system call, or API 10 million times while measuring the latency with the `RDTSCP` instruction.

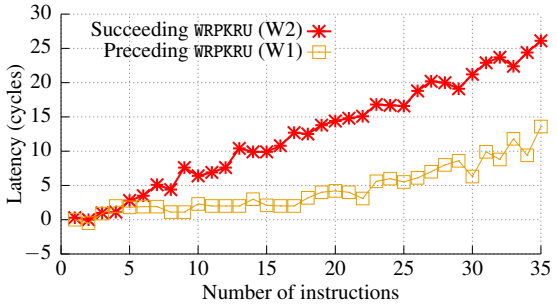


Figure 2: Effect of `WRPKRU` serialization on simple (i.e., ADD) instructions either preceding or succeeding `WRPKRU` (average of 10 million repetitions).

information to overwrite the `PKRU`, and the other two registers, `ECX` and `EDX`, filled with zeroes. `RDPKRU` also uses the three registers for its operation: it returns the current `PKRU` value via the `EAX` register while overwriting the `EDX` register with 0. The `ECX` register also should be filled with zeroes to execute `RDPKRU` correctly. Note that the actual usage of `ECX` and `EDX` registers is undocumented.

2.2 Kernel Integration and Standard APIs

The Linux kernel has supported MPK since version 4.6, and glibc has supported MPK since version 2.27. They focus on how to manage protection keys and how to assign them to particular PTEs. The Linux kernel provides three new system calls: `pkey_mprotect()`, `pkey_alloc()`, and `pkey_free()`. The kernel also changes the behavior of `mprotect()` to provide execute-only memory. glibc provides two userspace functions, `pkey_get` and `pkey_set`, to retrieve and update the access rights of a given protection key. Table 1 summarizes the APIs.

pkey_mprotect(). The `pkey_mprotect()` system call extends the `mprotect()` system call to associate a protection key with the PTEs of a specified memory region while changing its page protection flag. Interestingly, `pkey_mprotect()` does not allow a user thread to reset a protection key to zero, the default protection key value assigned to newly created memory pages such that it should be public to avoid accidental application crashes. We anticipate that resetting a key to zero is prohibited to avoid such potential crashes made by mistakes (i.e., denying access to the key zero).

pkey_alloc() and pkey_free(). The Linux kernel provides two other new system calls to allocate and de-allocate memory protection keys: `pkey_alloc()` and `pkey_free()`. When a user thread invokes `pkey_alloc()` with access right, the kernel allocates and returns a protection key with corresponding permission according to a 16-bit bitmap that tracks which protection keys are allocated. When a user thread invokes `pkey_free()`, the kernel simply marks the freed key as available in the bitmap. The `pkey_mprotect()` function examines the bitmap afterward to prohibit the use of non-allocated keys.

Execute-only memory. The Linux kernel supports execute-only memory with MPK. If a user thread invokes `mprotect()` only with `PROT_EXEC`, the kernel (1) allocates a new protection key, (2) disables the read and write permission of the key, and (3) assigns the key to the given memory region.

2.3 Quantifying Characteristics of Intel MPK

To evaluate the overhead and benefits of MPK, we measure (1) the overhead of the MPK instructions, (2) the overhead of the MPK system calls, and (3) the overhead of `mprotect()` for contiguous memory and sparse memory.

Environment. Our system consists of two Intel Xeon Gold 5115 CPUs (each CPU has 20 logical cores at 2.4 GHz) and 192GB of memory. Linux kernel version 4.14 configured for MPK is installed to this system.

Instruction latency. We measure the latency of `RDPKRU` and `WRPKRU` to identify their micro-architectural characteristics. Table 1 summarizes the results. The latency of `RDPKRU` is similar to that of reading a general register, but the latency of `WRPKRU` is high. We anticipate that `WRPKRU` performs serialization (e.g., pipeline flushing) to avoid potential memory access violation resulting from out-of-order execution. To confirm this, we insert various numbers of `ADD` instructions before (`W1`) and after (`W2`) `WRPKRU` and measure the overall latency (Figure 2). The results show that `W2` is always slower than `W1`, implying that the instructions executed right after `WRPKRU` fail to benefit from out-of-order execution because of the serialization.

System calls. We measure the latency of the four Linux system calls for MPK (Table 1). The latency of `mprotect()` and `pkey_mprotect()` on a 4 KB page is almost the same because they all rely on `do_mprotect_pkey()` internally. `pkey_alloc()` and `pkey_free()` are fast since they involve only simple operations in the kernel, and the domain switching between kernel and userspace dominates their time costs.

Contiguous versus sparse memory pages. Using MPK to change page permission involves only an update on the `PKRU` and thus is independent of the number of targeted pages and their sparseness. To show the performance benefit of MPK over `mprotect()`, we check how the number and sparseness of the targeted pages affect the performance of `mprotect()`.

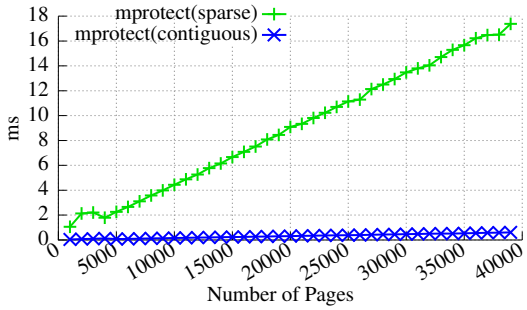


Figure 3: Overhead of `mprotect()` on contiguous and sparse memory (average cost of 10 million repetitions). Protecting contiguous pages takes less time than protecting sparse pages.

To construct contiguous memory pages, we call `mmap()` one time with certain memory size. For sparse memory pages, we call `mmap()` several times with one page size. Figure 3 shows that the overhead of `mprotect()` increases in proportion to the number of pages. The number of pages affects how many virtual memory areas (VMAs) [10] `mprotect()` needs to look up for permission update. Moreover, the overhead of `mprotect()` on sparse memory pages is high because multiple `mprotect()` calls introduce frequent context switchings between kernel and userspace.

Summary. Intel MPK allows a thread to rapidly change the per-thread access rights to a group of pages associated with the same protection key by updating a thread-local register `PKRU` which only takes around 20 cycles. Its performance is independent of the number of pages composing a group and their sparseness, unlike `mprotect()`.

3 Challenges of Utilizing Intel MPK

In this section, we explain the challenges of using MPK in terms of security, scalability, and synchronization.

3.1 Potential Security Problem

The existing OS support of MPK [3] suffers from the protection-key-use-after-free problem. In particular, `pkey_free()` just removes a protection key from a key bitmap and does not update the corresponding PTEs. Regardless of whether a key could already be associated with some pages, the kernel will allocate the key if it is freed by `pkey_free()`. If a program obtains a key that is still associated with some memory pages through `pkey_alloc()`, the new page group will include unintended pages that it is supposed to have. A developer can face this vulnerable situation unconsciously, as current kernel implementation neither handles this automatically nor checks if a free key is still associated with some pages. The developer community also recognized the problem and recommends not to free the protection keys [2, 13]. Handling this problem superficially (i.e., wiping protection keys in PTE) without a fundamental design change of memory management in the kernel will introduce huge performance overhead because it requires traversing the page table and VMAs to detect entries associated with a freed key to update them and flushing all corresponding TLB entries.

3.2 Limited Hardware Resources

Currently, MPK relies on a 32-bit `PKRU` such that it supports up to 16 keys. Developers are responsible for ensuring that an application never creates more than 16 page groups at the same time. This implies that developers have to examine at runtime the number of active page groups, which are used by both the application itself and the third-party libraries it depends on. Otherwise, the program may fail to properly benefit from MPK. This issue undermines the usability of MPK and discourages developers from utilizing it actively. Using a large register (e.g., 1024 bits) does not scale because MPK needs additional storage to associate keys with pages. For example, to support 512 protection keys, nine bits are necessary for each PTE, requiring enlarged page tables, shrunken address bits, or separate storage.

3.3 Semantic Differences

To change the permission of any page group, MPK modifies the value of the `PKRU`. However, the value is effective only in a single thread because `PKRU` is thread-local intrinsically as a register. As a result, different threads in a process can have different permissions for the same page group. This thread-local inherence helps to improve security for the applications that require isolation on memory access among different threads, but hinders MPK from optimizing and improving `mprotect()`. `mprotect()` semantically guarantees that page permissions are synchronized among all threads in a process on which particular applications rely. This not only makes it difficult to accelerate `mprotect()` with MPK, but also breaks the guarantee of execute-only memory implemented on mprotect in the latest kernel. `mprotect()` supporting executable-only memory relying on MPK does not consider synchronization among threads, which developers basically expect of `mprotect()`. Even when the kernel successfully allocates a key for the execute-only page, another thread might have a read access to it due to a lack of synchronization. To make MPK a drop-in replacement of `mprotect()` for both security and usability, developers need to synchronize the `PKRU` values among all the threads.

4 Software Abstraction of libmpk

`libmpk` provides a secure and usable abstraction for MPK by overcoming the challenges (§3). A developer can use MPK easily by either adding calls to `libmpk` APIs or replacing existing `mprotect()` calls with those of `libmpk`. By decoupling the protection keys from APIs, `libmpk` is immunized against protection-key-use-after-free. Also, `libmpk` allows an application to create more than 16 page groups by virtualizing the protection keys and provides a lightweight inter-thread `PKRU` synchronization mechanism. Figure 4 illustrates an overview of `libmpk`. The current version of `libmpk` consists of 1.5k

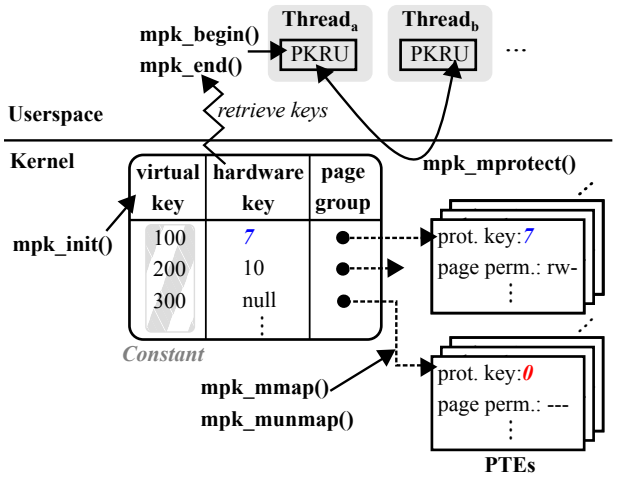


Figure 4: `libmpk` overview. `mpk_init()` pre-allocates hardware keys and initializes the metadata table. `mpk_mmap()` creates a page group with metadata, and `mpk_munmap()` destroys the page group and the corresponding metadata. `mpk_begin()` and `mpk_end()` provide domain-based thread-local isolation. `mpk_mprotect()` synchronizes permission changes globally.

Name	Argument	Description
`mpk_init()`	`evict_rate`	Initialize `libmpk` with an eviction rate
`mpk_mmap()`	`vkey, addr, len, prot flags, fd, offset`	Allocate a page group for a virtual key
`mpk_munmap()`	`vkey`	Unmap all pages related to a given virtual key
`mpk_begin()`	`vkey, prot`	Obtain thread-local permission for a page group
`mpk_end()`	`vkey`	Release the permission for a page group
`mpk_mprotect()`	`vkey, prot`	Change the permission for a page group globally
`mpk_malloc()`	`vkey, size`	Allocate a memory chunk from a page group
`mpk_free()`	`size`	Free a memory chunk allocated by `mpk_malloc()`

Table 2: `libmpk` APIs.

lines of C/C++ code in total.

Goals. To utilize MPK for domain-based isolation and as a substitute for `mprotect()`, we have to overcome the three challenges: (1) insecure key management, (2) hardware resource limitations, and (3) different semantics from `mprotect()`. `libmpk` adopts two approaches: (1) *key virtualization*, and (2) *inter-thread key synchronization*, which effectively solve the challenges. `libmpk` also protects its internal metadata from corruption.

4.1 Threat Model and Assumptions

`libmpk` has the following threat model and assumptions, in accordance with prior studies [21, 33, 35].

`libmpk` aims to prevent an adversary from reading from or writing in sensitive pages through memory corruption vulnerabilities. `libmpk` achieves this goal by protecting the sensitive pages with new MPK APIs and preventing the adversary from arbitrarily executing the `WRPKRU` instruction. We assume that a program should solely use the `libmpk` APIs to utilize MPK in a controlled manner. That is, the program should not use conventional MPK APIs together with the `libmpk` APIs. Also, any uncontrolled execution of the `WRPKRU` instruction should

```c
#define GROUP_1 100
#define GROUP_2 101

int domain_based_isolation () {
  mpk_init(-1); // default eviction rate: 100%
  char* addr = (char *)mpk_mmap(GROUP_1, NULL, 0x1000,
    PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
  // page permission: rw- & pkey permission: --

  mpk_begin(GROUP_1, PROT_READ | PROT_WRITE);
  // page permission: rw- & pkey permission: rw

  // write data in GROUP_1

  mpk_end(GROUP_1);
  // page permission: rw- & pkey permission: --

  printf("%s\n", addr); // SEGMENTATION FAULT
}

int quick_permission_change () {
  mpk_init(0.5); // set cache eviction rate: 50%
  void* addr = mpk_mmap(GROUP_2, NULL, 0x1000,
    PROT_READ | PROT_WRITE, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
  // page permission: rw- & pkey permission: --

  mpk_mprotect(GROUP_2, PROT_READ | PROT_WRITE | PROT_EXEC);
  // page permission: rwx & pkey permission: rw
}
```

Figure 5: Example code for `libmpk` APIs.

be prohibited by using existing countermeasures, such as data execution prevention [1], control-flow integrity [4, 22, 23, 38], and call gate [35].

4.2 libmpk API

`libmpk` provides eight APIs, shown in Table 2. To utilize `libmpk`, an application first calls `mpk_init()` to obtain all the hardware protection keys from the kernel and initialize its metadata. `mpk_mmap()` allocates a page group for a virtual key, which should be a constant integer that the developer passes. `mpk_munmap()` destructs a page group by freeing a virtual key for the group and unmaps all the pages. `libmpk` maintains the mappings between virtual keys and pages to avoid scanning all pages at this destruction step. On top of these, `libmpk` also provides simple heap over each page group (`mpk_malloc()` and `mpk_free()`), so that a developer can also use one or more page groups to create a heap memory region for sensitive data.

`libmpk` provides two usage models for developers. The first model, a thread-local domain-based isolation model, allows an application to temporarily grant permission to a page group only for the calling thread. `mpk_begin()` and `mpk_end()` are the APIs for this model, which make a page group accessible and inaccessible, respectively. The second model allows an application to quickly change the access rights to a page group by replacing `mprotect()` with `mpk_mprotect()`. Figure 5 shows an example of utilizing `libmpk` APIs.

4.3 Protection Key Virtualization

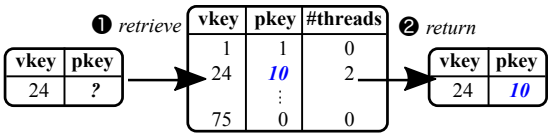
`libmpk` enables an application to create more than 16 page groups by virtualizing the hardware protection keys. When an application creates a new page group by calling `mpk_mmap()`, a virtual key passed as argument is associated with newly

allocated metadata for the new group. The application uses the virtual key to obtain or release the permission, or free the group, while being prohibited from manipulating hardware keys. The exact physical key that a page is associated with is hidden from the program and developer.

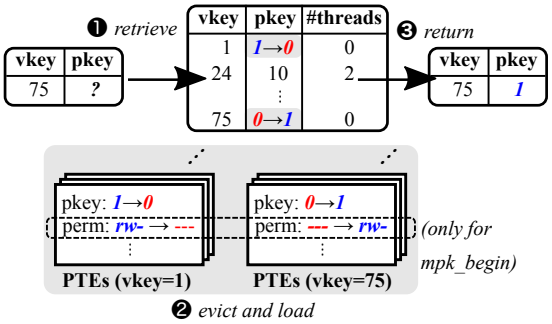
`libmpk` maintains the mappings between virtual and hardware keys through a cache-like structure (Figure 6). If a virtual key is already associated with a hardware key, the virtual key exists inside the cache and further access to it produces a few latencies. Otherwise, `libmpk` evicts another virtual key or does nothing but just invokes `mprotect()` for performance to change page permission. The frequency of eviction or calling `mprotect()` is determined by the eviction rate. The cache guarantees that a frequently updated virtual key will be mapped with a hardware key since it has a high possibility to be placed into the cache.

`libmpk` provides two policies to determine the mappings between virtual and hardware keys. When an application grants permission to a page group thread-locally by calling `mpk_begin()`, `libmpk` always maps the group's virtual key with a hardware key and uses it to grant access to the *calling thread*. `libmpk` maintains the mapping until the thread calls `mpk_end()` to release the access. For this reason, `libmpk` does not ensure that a calling thread always obtains the access due to hardware limitations. That is, if all hardware keys are actively used, `libmpk` is no longer able to provide any key. In this case, `mpk_begin()` raises an exception and lets the calling thread handle it (e.g., sleeps until a key is available). If a page group is not used by a thread, `libmpk` evicts the group by changing its protection key to 0 (default) and revoking its page permission to disallow subsequent accesses.

The second policy, `mpk_mprotect()`, also needs to map the virtual key to a hardware key, but not exclusively. Even when the page group is accessible, `libmpk` can unmap a hardware key and rely solely on the page attributes because all threads have the access. Hence, `libmpk` maps only the page groups whose access rights change frequently. If `libmpk` fails to find an available hardware key when it handles `mpk_mprotect()`, it unmaps and uses the least recently used (LRU) key for handling `mpk_mprotect()`. The hardware key of the evicted page group turns to 0. To avoid excessive overhead resulting from frequent unmapping, a developer can configure an *eviction rate* to control whether a hardware key has to be evicted according to how frequently its permission updates. In our approach, enforcing executable-only permission is not straightforward because a conventional approach (i.e., `mprotect()`) does not support executable-only permission. Therefore, `mpk_mprotect()` reserves one key for execute-only pages when an application creates them first, and does not evict this key until all executed-only pages disappear. Every incoming executable-only permission request is guaranteed to get a hardware protection key to achieve executable-only permission. If `mpk_mprotect()` has already been invoked for executable-only page groups, further requests will merge the incoming page groups with the existing executable-only ones to utilize the reserved key.



(a) Hit case: ❶ A thread calls `mpk_begin()` or `mpk_mprotect()` with a vkey; ❷ `libmpk` returns the corresponding pkey immediately.



(b) Miss case: ❶ A thread retrieves a vkey, but no corresponding pkey exists (pkey=0); ❷ `libmpk` evicts the LRU pkey. In addition, `mpk_begin()` updates the page permission of the evicted and loaded page groups using `mprotect()`; ❸ `libmpk` returns the new pkey.

Figure 6: Key virtualization in `libmpk`. *vkey* and *pkey* represent a virtual key and its corresponding hardware protection key associated with a page group. #threads indicates the number of threads running parallel inside a particular domain.

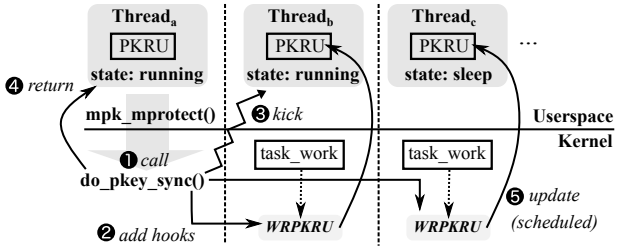


Figure 7: `PKRU` synchronization: ❶ `mpk_mprotect()` calls `do_pkey_sync()` to update the `PKRU` values of remote threads; ❷ `do_pkey_sync()` adds hooks to the threads' `task_work`; ❸ `do_pkey_sync()` kicks all the running threads for synchronization; ❹ `do_pkey_sync()` returns to its caller; ❺ The threads update their `PKRU`s when they are scheduled to run.

The integrity of `libmpk` metadata (e.g., the mappings between virtual and hardware keys, and the page group information) is important to prevent attackers from manipulating `libmpk`'s protection. For the metadata integrity, `libmpk` maps each metadata physical page into two virtual pages: a read-only page for its userspace code and a writable page for its kernel-space code. Updating the metadata can be done only by the `libmpk` kernel module and slightly modified system calls (e.g., `mmap()`, `munmap()`, and `mprotect()`). Most of simple metadata retrieval can be done by the userspace code to avoid unnecessary user-kernel mode switches.

Application	Protection	Protected data	#pkeys	#vkeys	Changed LoC
OpenSSL	Isolation	Private key	1	1	83
JIT (key/page)	W⊕X	Code cache	15	> 15	CC 10 | SM 18
JIT (key/process)	W⊕X	Code cache	1	1	CC 18 | SM 24 | v8 134
Memcached	Isolation	Slab, hashtable	2	2	117

Table 3: Three real-world applications of `libmpk`. To enable W⊕X in JavaScript engines, we use two approaches, including using a virtual key for every page in the code cache (*One key per page*) and using a single protection key for all the pages in the code cache (*One key per process*). **CC**, **SM**, and **v8** indicate Microsoft ChakraCore, Mozilla SpiderMonkey, and Google v8, respectively. **pkeys** and **vkeys** stand for protection keys and virtual keys, respectively.

4.4 Inter-thread Key Synchronization

`libmpk` implements an inter-thread `PKRU` synchronization technique, `do_pkey_sync()`, in `mpk_mprotect()` for two purposes: (1) to ensure no thread has the read access to an execute-only page and (2) to replace existing page-table-based `mprotect()` for performance. `do_pkey_sync()` guarantees that a `PKRU` update is globally visible and effective as soon as it returns. Intuitively, this requires a synchronous inter-thread communication; the calling thread needs to send messages to the other threads and wait until they update the `PKRU` value and acknowledge it, which suffers from a high cost.

We minimize the inter-thread `PKRU` synchronization latency in a *lazy* manner, leveraging the fact that the `PKRU` values are utilized in the userspace. If a remote thread is not currently being scheduled, it does not need the up-to-date `PKRU` value immediately. Even if the thread is currently being scheduled, we only need to update its `PKRU` value when it returns to the userspace. If the calling thread can create a hook that the other threads will invoke right before jumping back to the userspace and ensure that they are not in the userspace, we can guarantee that all the other threads have the new `PKRU` value when `do_pkey_sync()` returns. Figure 7 illustrates the overall procedure of `mpk_mprotect()`. `do_pkey_sync()` utilizes an existing hooking point in the Linux kernel to enforce the remote threads to update the `PKRU` values right before returning to the userspace and ensures that all threads use the new `PKRU` value by sending rescheduling interrupts. In Linux, a thread can register a list of callback functions (`task_work`) that are invoked at designated points (e.g., when returning to the userspace) by calling `task_work_add()`. In this way, `do_pkey_sync()` guarantees that all the remote threads eventually acquire the new `PKRU` value. Although `do_pkey_sync()` still needs to send inter-processor interrupts to ensure that no other thread uses the old `PKRU` value after a certain point, our evaluation shows that the overall latency of `mpk_mprotect()` is less than that of `mprotect()` (§6.2).

5 Applications

We demonstrate the security benefit, efficiency, and usability of `libmpk` by augmenting three types of popular applications: an SSL library, three JavaScript Just-in-time (JIT) compilation engines, and an in-memory key-value store. Table 3 summarizes the mechanisms (e.g., page isolation or W⊕X) that we aim to provide as well as the protected data (e.g., key or code). Evaluation results are described in §6.

5.1 OpenSSL

OpenSSL is a popular open-source library implementing the secure sockets layer (SSL) and transport layer security (TLS) protocols. Since it manages sensitive information (e.g., private keys and encrypted data), its information leakage bugs are security-critical. For example, OpenSSL's Heartbleed bug [26] allowed attackers the chance to leak sensitive data from millions of web servers.

We apply `libmpk` to OpenSSL to protect its private keys from potential information leakage by storing the keys in isolated memory pages. More specifically, the isolated memory pages are protected by single pkey or multiple pkeys assigned per private key to show the trade-off between performance and security. First, we identify all the data types that store private keys (e.g., `EVP_PKEY`) and replace their heap memory allocation function from `OpenSSL_malloc()` to `mpk_malloc()` for single pkey or `mpk_mmap()` for multiple pkeys to store them in an isolated memory region. Next, we locate all the functions that access private keys (e.g., `pkey_rsa_decrypt()`) and modify them to access the isolated memory region by inserting `mpk_begin()` and `mpk_end()` before and after their call sites. Note that assigning pkey per private key offers finer-grained security, which minimizes the attack window for the isolated memory region. For example, even if a function whose call site is located between `mpk_begin()` and `mpk_end()` has a memory leakage bug, it cannot access any other isolated pages except the single page isolated with the pkey provided to `mpk_begin()` as argument.

5.2 Just-in-time (JIT) Compilation

JIT compilation dynamically translates interpreted script languages, e.g., JavaScript and ActionScript, into native machine code or bytecode to avoid the overhead of full compilation and repeated interpretation. Technically, it relies on writable code, resulting in potential arbitrary code execution. To support JIT compilation, the *code cache* that stores code generated at runtime needs to be writable for a JIT compilation thread and be executable for an execution thread. Thus, if attackers compromise the JIT compilation thread, they can make the execution thread execute the code they provide.

ChakraCore [27] and SpiderMonkey [28] mitigate the above-mentioned problem by enforcing the W⊕X security policy on the code cache with `mprotect()`. They make the code cache writable while disallowing execution when they are updating code, and, after it has updated, they make the code cache executable while disallowing write. However, they can suffer from *race condition attacks* [33] because they use `mprotect()` to change page permissions; that is, when a

thread makes the code cache writable with `mprotect()`, other threads compromised by attackers can also manipulate the code cache with the same permission.

We apply `libmpk` to the three popular JavaScript engines (SpiderMonkey, ChakraCore, and v8) to enforce the W⊕X security policy without the race condition problem while ensuring better performance. We propose two approaches to implement the W⊕X policy with `libmpk`.

One key per page. A context-free solution is to replace `mprotect()` with `libmpk` APIs to perform fast permission switches on targeted pages in the code cache. All the protection keys are initialized with read-only permission when a new thread is created. We dedicate *one protection key* to *one page* when it is the first time to be re-protected via `mprotect()` and change its page permission to `rwx`. Later, we only need to call `mpk_begin()` and `mpk_end()` before and after when the JIT compiler updates the corresponding page. Based on the observation that generally only one page is updated at a time, we still invoke `mprotect()` if multiple pages change permission.

One key per process. Another approach is to use a single protection key for the entire code cache. When pages are first committed from the preserved memory region into the code cache, they are assigned with the protection key and their page permission is set to `rwx`. Whenever any page in the code cache is to be updated, the script engine needs to call `mpk_begin()` and `mpk_end()`. Although more pages become temporarily writable, the security of the code cache is ensured thanks to the per-thread view of the protection key.

5.3 In-Memory Key-Value Store

In-memory key-value stores, such as Memcached, are widely used to manage a large amount of data in memory to ensure low latency and high throughput. With a high requirement for performance, such key-value stores normally avoid using security techniques whose performance depends on input size (e.g., `mprotect()` and encryption) to protect stored data. This implies that, if an in-memory key-value store has arbitrary read or write vulnerabilities, attackers are able to leak or corrupt sensitive information stored inside.

`libmpk` manages to efficiently mitigate such attacks. To demonstrate this, we apply `libmpk` to Memcached. `libmpk` protects Memcached's slabs that contain values and hash tables that maintain key-value mappings by replacing Memcached's `malloc()` function with `mpk_malloc()`, and wraps the call sites of all the legitimate functions (e.g., `ITEM_key()` and `assoc_find()`), which operate on protected data with `mpk_begin()` and `mpk_end()`. Note that we assign two different keys to slabs and hash tables, to narrow the attack surface. It is possible to use more keys to secure slabs in a fine-grained manner, e.g., differentiating them according to their sizes. More importantly, `libmpk`'s performance is independent of the size of memory to protect, and thereby efficiently works with Memcached even when protecting data of several gigabytes.

6 Evaluation

In this section, we evaluate `libmpk` in terms of its security implication and performance by answering the following questions:

- What security guarantees does `libmpk` provide? (§6.1)
- Does `libmpk` solve the security, scalability, and semantic-gap problems of existing MPK APIs without introducing much performance overhead? (§6.2)
- Does `libmpk` have negligible performance impact and outperform `mprotect()` in real-world applications? (§6.3)

The same system environment explained in §2.3 is used for performance evaluations.

6.1 Security Evaluation

We first evaluate the security benefits from `libmpk` regarding memory protection and isolation. For OpenSSL and Memcached, `libmpk` provides domain-based isolation to protect memory space that stores sensitive data. The permission for the particular memory space set by `libmpk` is locally effective, which also prevents malicious accesses from other compromised threads. In particular, exploiting a memory corruption bug to leak or ruin sensitive data stored in the isolated pages is killed by segmentation faults resulting from the lack of permission. To verify this, we mimic the Heartbleed vulnerability by deliberately introducing a heap-out-of-bounds read bug and inserting a decoy private key placed next to the victim heap region. When the vulnerability is triggered, OpenSSL hardened by `libmpk` crashes with invalid memory access. However, `libmpk` cannot fully mitigate memory leakage that originates inside the protected domain. Thus, developers should carefully design the domain to minimize the potential attack surface when using `libmpk` in their applications.

JavaScript JIT compilers can use `libmpk` to guarantee W⊕X for JIT code pages. Unlike `mprotect()`, `libmpk` is immune to race condition attacks launched by compromised threads running in parallel resulting from the thread-local effectiveness of protection keys. When the JIT compiler uses `libmpk` to switch the permission of a code page for updates, other threads controlled by attackers cannot write malicious shellcode into the page simultaneously. To verify this, we introduce two custom JavaScript APIs for arbitrary memory read and write to SpiderMonkey and ChakraCore, and test a simple PoC that leverages these two APIs to locate a JIT code page and write shellcode into it. Both engines crash with a segmentation fault at the end.

6.2 Microbenchmarks

We run several microbenchmarks to understand the performance behavior of APIs in `libmpk`.

Cache performance. `libmpk` introduces a cache to enable protection on more than 16 page groups, whose performance

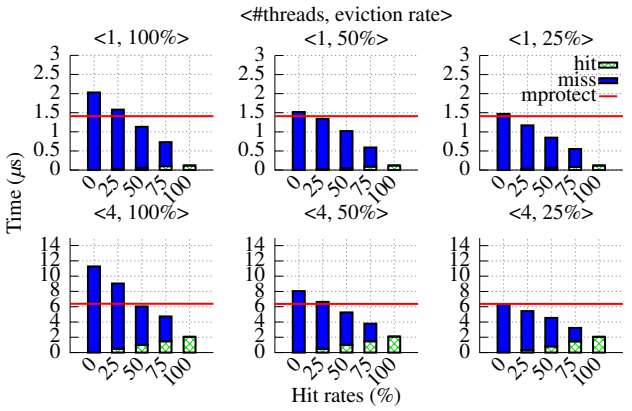


Figure 8: Latency of `libmpk`'s key cache with various hit rates, eviction rates, and different number of threads. `mpk_mprotect()` and `mprotect()` are invoked on a 4 KB page. Red line marks the overhead of `mprotect()`. When the hit rate is 100%, `mpk_mprotect()` is 12.2× faster than `mprotect()` for one thread and 3.11× faster for four threads.

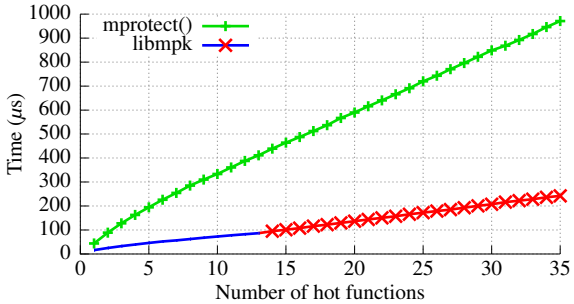


Figure 9: Average time cost to update permission when original and modified ChakraCore JIT-compile an increasing number of hot functions demanding distinct virtual keys.

is affected by its eviction rate and hit rate and the number of virtual keys in use. We run the following two microbenchmarks to check the cache performance.

Hit rate and eviction rate. The first benchmark measures cache performance with different hit rates, eviction rates, and number of threads. We run the benchmark with both one thread and four threads, where each thread warms up by filling the key cache to evade cold miss and invokes `mpk_mprotect()` on one page for a hundred times after 15 entries are filled. Figure 8 presents the evaluation results, where (1) the green box indicates the overhead incurred by the cache hit, which is dominated by the time cost on `WRPKRU` and maintaining internal data structures; (2) the blue box indicates the overhead incurred by the cache miss, which is dominated by the time cost on key eviction. More specifically, `mpk_mprotect()` needs to unset the protection key that is to be evicted and bind a new virtual key to it. We test the microbenchmark with three eviction rates that indicate the ratio of cache misses that eventually leads to key eviction. If a cache miss occurs without key eviction, `mprotect()` is invoked to change the permission of the pages.

Experimental results show that `mpk_mprotect()` outperforms `mprotect()` except when the cache hit rate is below 25% with an eviction rate above 50%. This is because, unlike `mprotect()`, `mpk_mprotect()` does not merge and split the VMAs of targeted pages. It becomes slow when being tested with four threads, but is still comparable with `mprotect()`, whose latency also increases in a multi-threading program.

Number of virtual keys. To evaluate how the number of used virtual keys affects the cache performance of `libmpk`, we re-implement W⊕X in ChakraCore in a *one-key-per-page* approach (see §5.2) and set the eviction rate as 100%. To introduce an increasing number of pages to be protected (i.e., an increasing number of virtual keys to be used) during the execution of ChakraCore, we design a simple microbenchmark. The microbenchmark consists of a set of JavaScript files, and the *i*th file contains *i* hot functions being invoked for 100,000 times. For each hot function, ChakraCore allocates one new executable page to store the native code and performs nine permission switches on the page through one virtual key at runtime. Without any hot function, ChakraCore allocates one page in the code cache. We run the original ChakraCore (version 1.9.0.0-beta) and the modified one with our microbenchmarks, and record the time cost of changing permission of the pages in the code cache (i.e., the execution time of `VirtualProtect()` and that of `mpk_begin()` and `mpk_end()`) in total. Each JavaScript file is executed 200 times, and the average time is presented in Figure 9.

The result shows that with the `libmpk`-based implementation of W⊕X, the time cost on permission switches linearly increases when more hot functions are emitted and thus more virtual keys are allocated to protect the code pages of the hot functions. In particular, after 15 virtual keys are allocated (marked in red), the time cost increases slightly faster than before (marked in blue) as a result of cache eviction. Nevertheless, the ChakraCore hardened by `libmpk` still outperforms by 3.2× the original ChakraCore using `mprotect()` to enforce W⊕X.

Memory overhead. `libmpk` dedicates memory space to store its internal data structures for maintaining the metadata of these page groups under protection (see §4.3). Each `mpk_mmap()` allocates 32 bytes of memory to store the information of a new page group (e.g., base address and size). `libmpk` maintains a hashmap to store the mapping between virtual keys and hardware keys for fast query and access. In the current implementation, we pre-allocate 32 KB of memory for the hashmap, and its size will automatically expand when a program invokes `mpk_mmap()` more than about 4,000 times.

Synchronization latency. Figure 10 shows the latency of inter-thread permission synchronization using `mpk_mprotect()` and `mprotect()` on memory of varying sizes.

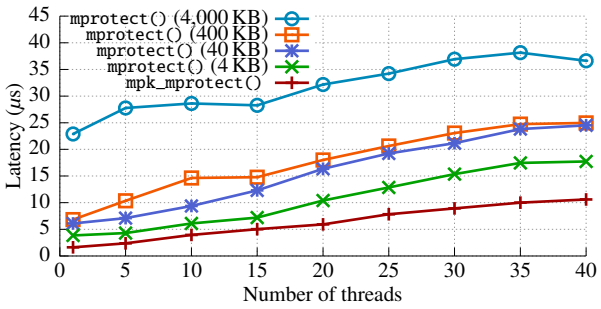


Figure 10: Latency of inter-thread permission synchronization using `mpk_mprotect()` and `mprotect()` calls on memory of varying sizes. `mpk_mprotect()` outperforms `mprotect()` 1.73× for a single page and 3.77× for 1,000 pages.

`mpk_mprotect()` is 1.73× faster than `mprotect()` when updating the permission of a single page. The latency of `mprotect()` increases with the number of pages it changes due to the expensive operations of managing VMAs. Compared to `mpk_mprotect()`, `mprotect()` costs at least 3.78× to change the permission of 1,000 pages. The performance overhead of `mpk_mprotect()` is independent of the number of pages whose permission has been updated. Figure 10 also shows that when there are many threads, the latency of both `mprotect()` and `mpk_mprotect()` increases; `mprotect()` flushes more TLBs, whereas `mpk_mprotect()` creates many hooks in the kernel.

6.3 Application Benchmarks

We measure the performance overhead of `libmpk` in practice by evaluating three applications proposed in §5.

OpenSSL. The Apache HTTP server [12] (`httpd`) uses OpenSSL to implement SSL/TLS protocols. To evaluate the overhead caused by `libmpk`, which is introduced to protect private keys, we use ApacheBench to test `httpd` with both the original OpenSSL library and the modified one with `libmpk`. ApacheBench is launched 10 times and each time sends 1,000 requests of different sizes from four concurrent clients to the server. We choose the DHE-RSA-AES256-GCM-SHA256 algorithm with 1024-bit keys as a cipher suite in the evaluation.

Figure 11 presents the evaluation result. On average, `libmpk` introduces 0.58% and 4.82% performance overhead, respectively, in terms of the throughput. In the single pkey case, the negligible overhead mainly comes from internal data structure maintenance in `libmpk`. In the multiple pkeys case, `httpd` utilizes more than 1,000 pkeys, as it allocates a new pkey while creating a new session. These pkeys are maintained by cache invoke eviction, so the multiple pkeys generates higher overhead than the single pkey case.

Just-in-time compilation. We applied two proposed W⊕X solutions based on `libmpk`, namely, *one key per page* and *one key per process* (§5.2) to both SpiderMonkey (version 59.0) and ChakraCore (version 1.9.0.0-beta) and evaluated their performance with the Octane benchmark [15], which involves heavy JIT-compilation workloads at runtime. Each

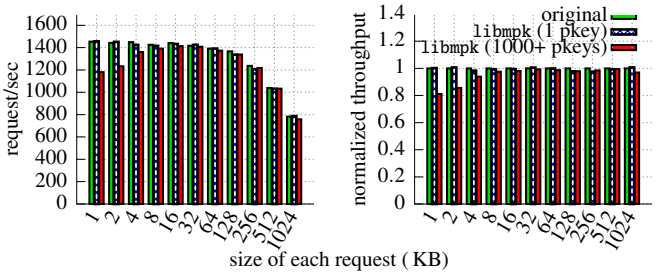


Figure 11: Throughput of original `httpd` and `httpd` hardened by `libmpk`. Protecting private keys with single pkey and 1000+ pkeys, `libmpk` slows down `httpd` by at most 2.52% and 18.84% respectively. The averages of overhead are 0.58% and 4.82% respectively.

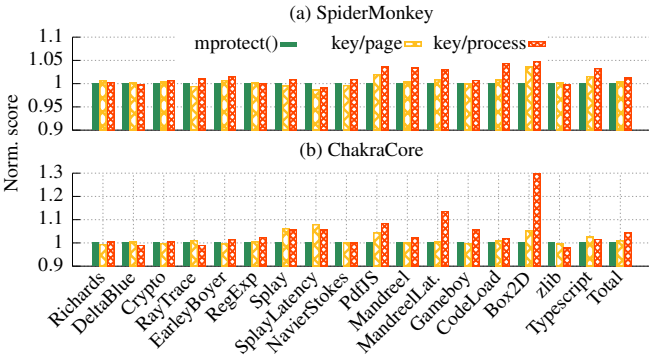


Figure 12: Octane benchmark scores of SpiderMonkey and ChakraCore with original and `libmpk`-based W⊕X solutions. `libmpk` outperforms the original, `mprotect()`-based defense by at most 4.75% (SpiderMonkey) and 31.11% (ChakraCore).

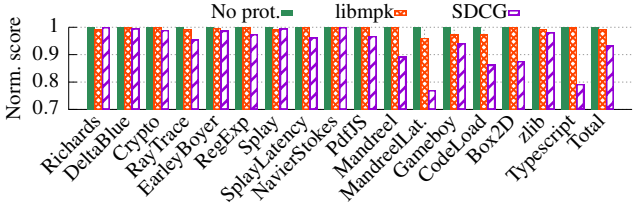


Figure 13: Octane benchmark scores of original v8 and two modified versions of v8 ensuring W⊕X by SDCG and `libmpk`. `libmpk` only introduces 0.81% overall performance overhead for W⊕X in v8, compared with 6.68% caused by SDCG.

JavaScript program in the benchmark was directly executed by the original and modified script engines for 20 times, and we calculated the average score (Figure 12).

For SpiderMonkey, both `libmpk`-based approaches outperform the `mprotect()`-based approach on the total score, namely, 0.38% and 1.26%, which is consistent with the claim from Firefox developers that enabling W⊕X with `mprotect()` in SpiderMonkey introduces less than 1% overhead for the Octane benchmark. The reason is that SpiderMonkey is designed to get rid of unnecessary `mprotect()` calls when its JIT compiler works. The performance scores of nearly all

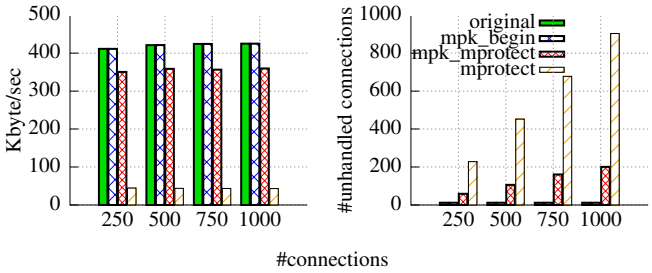


Figure 14: Throughput and unhandled concurrent connections of original Memcached and three versions of Memcached whose key-value pairs are protected by `mpk_begin()`, `mpk_mprotect()`, and `mprotect()`. `mpk_begin()`'s overhead is negligible compared to the original. `mpk_mprotect()` outperforms `mprotect()` 8.1× while ensuring the same semantics.

the programs increase through *one key per page* (at most 3.60% on `Box2D`) and *one key per process* (at most 4.75% on `Box2D`), except for `SplayLatency` protected by *one key per page* (dropped by 1.36%). `SplayLatency` becomes worse because it barely updates the code cache such that the initial overhead to associated keys with pages cannot be hidden.

Our two `libmpk`-based approaches improve ChakraCore by 1.01% and 4.39% on the total score of the Octane benchmark, respectively. ChakraCore is suitable for `libmpk`-based W⊕X solutions since it only makes one page writable per time regardless of emitted code size. *One key per page* increases the performance score of ChakraCore at most 7.96% when testing `SplayLatency`, while *one key per process* improves the performance by at most 31.11% on `Box2D`. Similar to the results of SpiderMonkey, we observe a few performance degradations when benchmarks rarely update code cache.

For v8, we compare our approach with a `mprotect()`-based scheme, SDCG [33]. SDCG protects the JIT code pages of v8 with W⊕X by emitting the code in a dedicated process. No other processes can change the code pages. To demonstrate the performance advantage of our in-process `libmpk`-based approaches, which are free of race condition attacks, we applied one of our approaches, *one key per process*, to Google v8 (version 3.20.17.1 used in [33]) and evaluated the performance through the Octane benchmark as well. Figure 13 presents the performance comparison among the original v8, v8 with SDCG, and v8 with `libmpk`. Note that originally, v8 has not deployed W⊕X to protect its code cache so far. Our approach only introduces 0.81% overall performance loss, compared with 6.68% caused by SDCG.

To summarize, our `libmpk`-based approaches, which are free of the race condition attacks, outperform the `mprotect()`-based approach currently applied in practice to enforce W⊕X protection on code cache pages with negligible overhead.

In-memory key-value store. To study the performance overhead of `libmpk` when protecting large memory, we evaluate the modified Memcached whose key-value pairs are isolated by `libmpk`. More specifically, the modified Memcached preallocates 1 GB memory, which is used instead of slab pages allocated by glibc `malloc()` to store key-value pairs. Besides the original Memcached, we also evaluate the Memcached whose key-value pairs are protected by `mprotect()`. To study the performance of `mpk_mprotect()` in real-world applications, we also create the Memcached guarded by `libmpk` with permission synchronized as another evaluation target for comparison. Each aforementioned version of Memcached launches with four concurrent threads, and we connect to it remotely through `twemperf` [34]. We create from 250 to 1,000 connections per second, and 10 requests are sent during each connection.

Figure 14 presents the evaluation results. The modified Memcached hardened by `libmpk` only has 0.01% overhead in terms of data throughput and almost no overhead regarding concurrent connections processed per second, which indicates that `libmpk` performs well even when protecting a huge number of pages. By contrast, `mprotect()` introduces nearly 89.56% overhead in terms of data throughput when protecting 1 GB memory in Memcached and a large number of unhandled concurrent connections accumulate in this case. This is because `mprotect()` involves page table traversing, which is considered expensive when dealing with a large number of pages. To evaluate the synchronization service of `libmpk` in practice, we also run Memcached protected by `mpk_mprotect()`. This design ensures the same semantics but outperforms `mprotect()` 8.1× regarding throughput.

`libmpk` provides the same functionality of `mprotect()` with much better performance when protecting huge memory. Moreover, in multi-threading applications, using `mprotect()` to ensure in-thread memory isolation requires lock, which is not required when using `libmpk` because of its inherent property.

7 Discussion

In this section, we discuss a potential attack on both Intel MPK and `libmpk`.

Rogue data cache load (Meltdown). We found that Intel MPK can suffer from the rogue data cache load, also known as the Meltdown attack [19,24]. The Meltdown attack is possible because current Intel CPUs check the access permission to a specific memory page after they have loaded it into the cache. MPK is not an exception because Intel CPUs check the access rights of `PKRU` when checking the page permission at the same pipeline phase. This allows attackers to infer the content of a present (accessible) page even when its protection key has no access right. Since Intel is considering hardware-level mitigation techniques [19], we believe this problem will be solved in the near future.

8 Related Work

MPK applications. While conducting our study, we noticed that there were a few ongoing studies using MPK to achieve different goals. Burow et al. [8] leverage both MPK

and memory protection extension (MPX) to efficiently isolate the shadow stack. ERIM [35] utilizes MPK to isolate sensitive code and data. MemSentry [21] provides a unified memory isolation framework to use various hardware features, including MPK and Memory Protection Extensions (MPX), with the same interface. XOM-Switch [39] relies on MPK to enable execute-only memory for unmodified binaries, and IskiOS [16] leverages MPK and kernel page table isolation (KPTI) to enforce execute-only memory in kernel. Our effort to provide a software abstraction for MPK is orthogonal to these studies, which are all potential applications of `libmpk`. These schemes can leverage `libmpk` to achieve secure and scalable key management to create as many sensitive memory regions as required securely.

Memory protection with other hardware features. Other hardware features exist for efficient memory protection such as ARM Domain [5] and IBM Storage Protection [18], which have a similar concept to MPK. For instance, ARMlock [40], FlexDroid [31], and Shreds [9] rely on Domain to isolate untrusted program modules, third-party libraries, and sensitive code modules, respectively. `libmpk` helps to port these applications from ARM to the Intel platform.

Software-based fault isolation (SFI). SFI [36] prohibits unintended memory accesses by inserting address masking instructions just before load and store instructions. This idea motivates many applications to utilize and further optimize it. Sandboxing mechanisms, such as Native Client (NaCl) [14, 30], relies on SFI to isolate untrusted code. Code-Pointer Integrity [23] also uses SFI to protect the code pointers from unsanitized memory accesses. SFI enables an application to partition its memory into multiple regions, but the cost of address masking limits the shape of partitions, which are commonly contiguous pieces of memory. By contrast, MPK enables an application to partition the memory into the regions with arbitrary shape. Further, the overhead of SFI on address masking increases by the number of isolated memory regions, unlike MPK.

Multiple virtual address spaces. Using multiple virtual address spaces for a single program can protect the memory of sensitive or untrusted components from the others. Some systems [7, 17, 29, 37] rely on multiple page tables to isolate the memory of threads in a single process from each other. Other systems [6, 11, 25] also provide different memory views to individual threads or small execution units using separated page tables. Kenali [32] uses a page-table-based isolation mechanism to protect sensitive data in which a separate page table is created for each thread. Unlike `libmpk`, these mechanisms suffer from non-negligible performance overhead resulting from slow and frequent page table switches.

9 Conclusion

Intel MPK supports efficient per-thread permission control on groups of pages. However, its hardware implementation and software support suffer from security, scalability, and semantic-gap problems. `libmpk` proposes a secure, scalable, and semantic-gap-mitigated software abstraction of MPK for developers to perform fast memory protection and domain-based isolation in their applications. Evaluation results show that `libmpk` incurs negligible performance overhead (<1%) for domain-based isolation and better performance for a substitute of `mprotect()` when adopted to real-world applications: OpenSSL, JavaScript JIT compiler, and Memcached.

10 Acknowledgment

We thank the anonymous reviewers, and our shepherd, John Criswell, for their helpful feedback. This research was supported, in part, by the NSF award CNS-1563848, CNS-1704701, CRI-1629851 and CNS-1749711 ONR under grant N00014-18-1-2662, N00014-15-1-2162, N00014-17-1-2895, DARPA TC (No. DARPA FA8650-15-C-7556), and ETRI IITP/KEIT[B0101-17-0644], and gifts from Facebook, Mozilla and Intel.

References

[1] "Exec Shield", new Linux security feature, 2003. https://lwn.net/Articles/31032/.

[2] Linux kernel, v4.20, 2018. https://elixir.bootlin.com/linux/v4.20-rc1/source/mm/mprotect.c#L630.

[3] Pkeys(7) linux programmer's manual, 2018. http://man7.org/linux/man-pages/man7/pkeys.7.html.

[4] Martín Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, November 2005.

[5] ARM. ARM® Architecture Reference Manual ARMv7-A and ARMv7-R edition, 2018.

[6] A. Bittau, P. Marchenko, M. Handley, and B. Karp. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, April 2008.

[7] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the 13th USENIX Security Symposium (Security)*, San Diego, CA, August 2003.

[8] Nathan Burow, Xinping Zhang, and Mathias Payer. SoK: Shining light on shadow stacks. In *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[9] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained Execution

Units with Private Memory. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[10] Gustavo Duarte. How the Kernel Manages Your Memory, 2009. https://manybutfinite.com/post/how-the-kernel-manages-your-memory/.

[11] Izzat El Hajj, Alexander Merritt, Gerd Zellweger, Dejan Milojicic, Reto Achermann, Paolo Faraboschi, Wen-mei Hwu, Timothy Roscoe, and Karsten Schwan. SpaceJMP: Programming with Multiple Virtual Address Spaces. In *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, April 2016.

[12] Apache Software Foundation. Apache HTTP Server Project, 2018. https://httpd.apache.org/.

[13] Free Software Foundation. The gnu c library, 2018. https://www.gnu.org/software/libc/manual/html_mono/libc.html#Memory-Protection.

[14] Google. NaCl SFI model on x86-64 systems. https://developer.chrome.com/native-client/reference/sandbox_internals/x86-64-sandbox.

[15] Google. The JavaScript Benchmark Suite for the modern web, 2017. https://developers.google.com/octane.

[16] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. IskiOS: Lightweight defense against kernel-level code-reuse attacks. *arXiv preprint arXiv:1903.04654*, 2019.

[17] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[18] IBM. Power ISATM Version 3.0 B, 2017.

[19] Intel. Intel Analysis of Speculative Execution Side Channels, 2018.

[20] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual, 2018.

[21] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, April 2017.

[22] Hyungjoon Koo, Yaohui Chen, Long Lu, Vasileios P Kemerlis, and Michalis Polychronakis. Compiler-assisted Code Randomization. In *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2018.

[23] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-Pointer Integrity. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, October 2014.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.

[25] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.

[26] N. Mehta and Codenomicon. The Heartbleed Bug, 2014. http://heartbleed.com/.

[27] Microsoft. ChakraCore is the core part of the Chakra Javascript engine that powers Microsoft Edge, 2018. https://github.com/Microsoft/ChakraCore.

[28] Mozilla. Spidermonkey, 2018. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey.

[29] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, August 2003.

[30] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, August 2010.

[31] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Taesoo Kim, and Insik Shin. FlexDroid: Enforcing In-App Privilege Separation in Android. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[32] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William R. Harris, Taesoo Kim, and Wenke Lee. Enforcing Kernel Security Invariants with Data Flow Integrity. In *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2016.

[33] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and Protecting Dynamic Code Generation. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[34] Twitter. twemperf, 2018. https://github.com/twitter-archive/twemperf.

[35] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter 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.

[36] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient Software-based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP)*, Asheville, NC, December 1993.

[37] Jun Wang, Xi Xiong, and Peng Liu. Between Mutual Trust and Mutual Distrust: Practical Fine-grained Privilege Separation in Multithreaded Applications. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, July 2015.

[38] Chao Zhang, Chengyu Song, Kevin Zhijie Chen, Zhaofeng Chen, and Dawn Song. VTint: Protecting Virtual Function Tables Integrity. In *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2015.

[39] 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.

[40] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS)*, Scottsdale, Arizona, November 2014.