École Polytechnique Fédérale de Lausanne

Leveraging Accelerated Hardware Memory Protection on FreeBSD

by Frédéric Gerber

Master Semester Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Andrés Sánchez Marín
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 7, 2022

“I love summer in Denmark. It is my favourite day of the year.”
— Meik Wiking

Dedicated to my friends, who still let me come on holiday with them, even though they knew that I had to finish this semester project and thus they would have to cook for me.

Back in June, when I first got in touch with my supervisor Andrés, I was a few days away from leaving Switzerland for a bike tour around Europe with two friends of mine. This trip took us from Budapest to Copenhagen over an entire month, and we finished our journey in Denmark with yet another friend who had spent the summer there. While taking my mind off academic purposes over the holidays, I still had my upcoming semester project in mind during that time, which is why this report on memory protection keys reminds me of those carefree summer days in Denmark.

As I am writing this report, I am now sitting in Belgium, again on vacation, but this time with a bigger task to finish, namely the completion of the code and the report of this project. A whole semester has passed since our bike trip, and I have learned a lot in between, but somehow I needed this second period out of town to actually have large time slots to allocate to my practical work – even though it meant missing out on some fun activities. Taking time off is always a refreshing opportunity to take a step back from attending lectures, completing exercises and preparing exams, in order to actually focus on what I like most in my studies: enjoying meaningful moments with people I care about and learning from exciting computer science projects.

Acknowledgments

First of all, many thanks to Prof. Payer for his Software Security class, which made me realize that there actually exists an area combining operating system design and hardware component development that realistically takes advantage of both to engage in the evolution of modern computer systems. I had always been interested in operating systems and their interactions with hardware, and this project with a taste of security was a natural fit from the beginning.

Therefore, I want to thank my supervisor Andrés Sánchez, since he came up with the idea of using the very recent Intel memory protection keys as part of his research. Over this semester project, he has been of continuous support, providing me with weekly feedback to guide me through my sometimes clumsy steps. Moreover, I have always been able to count on his responsive help, be it besides our scheduled meetings or even outside regular working hours.

Finally, I am grateful for the discussions with my friends and family, both when things were advancing well and when I was encountering issues, since I could always establish a comforting discussion when I ran into issues.

Lausanne, January 7, 2022 Frédéric Gerber

Abstract

In modern computer systems, practical abstractions are provided to the applications by the operating system, *e.g.* Linux, FreeBSD or other. One example of such abstraction is the one of virtual memory, which enables the address space of a process to be protected from the one of others. Usually, this protection mechanism is dealt with by the page table entries themselves, which additionally to holding mappings to physical page frames, assign to each virtual page number two bits corresponding to read/write/execute protections.

Since this traditional protection mechanism needs to go through a system call due to its interaction with page tables, Intel has recently added support for memory protection keys, which are also encoded in a page table entry, but their associated rights can be modified directly in user-space. To this end, different software support has been added on Linux and FreeBSD, but a convenient library called LibMPK has been implemented only for Linux. In this project, we adapt the library so as to make it work on FreeBSD as well, leveraging the support it offers in the OS.

After describing the register holding permissions associated to protection keys, as well as the software library that can manipulate them, we discuss the new system calls that are necessary to make it work and evaluate whether it successfully operates on a remote machine. More precisely, the examples from the LibMPK paper are shown to work and some additional code snippets have also been tested. Finally, a literature review of related work is presented, illustrating parallel research and motivating the use of LibMPK.

Contents

Chapter 1

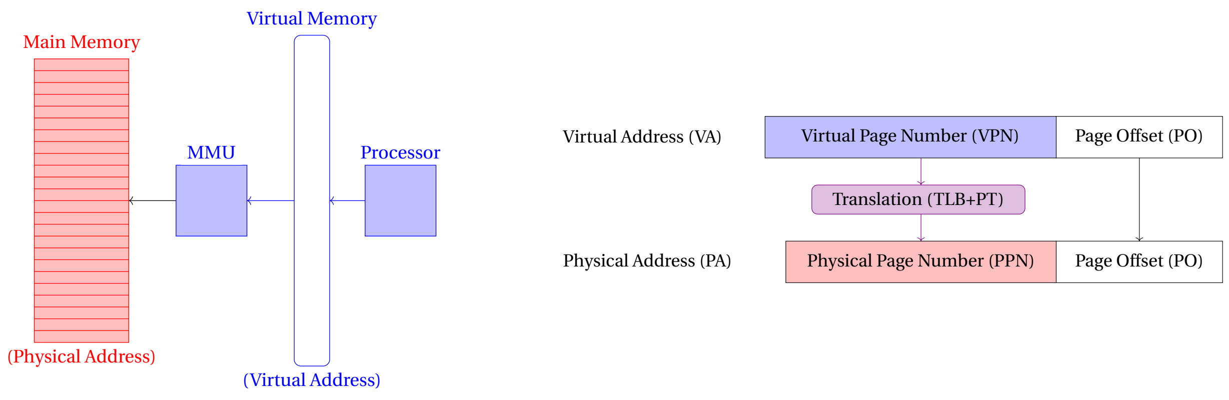
Introduction

In modern computer systems running on a conventional von Neumann architecture, the *central processing unit* (CPU), which is connected to some main memory module, is usually executing a piece of software called the *operating system* (OS). The goal of the OS is among other things to provide abstractions to the programmer in order to ease their access to hardware. More generally, it brings different devices, both virtual and physical, into a single coherent picture that is arbitrated as fairly as possible.

In this project, two OSes are relevant: Linux and FreeBSD. Both of them are open-source Unix-like OSes for x86, but they have been ported to more architectures since their original development. They are quite similar, but they come with different components upon installation [4]: Linux delivers only a kernel and device drivers, but no third party software; whereas FreeBSD delivers a kernel, device drivers, as well as user-space utilities. Additionally, they also differ in licensing: while Linux has a GPL license, FreeBSD comes under a BSD license. The latter allows some more code adaptations, but remains compatible with standard GPL.

If there is one aspect in which computer systems shine, it is probably *virtualization*. Indeed, pretending to be something else is not usually an easy task for a human, but computers do it all the time. Here are some famous examples: processes virtualize the machine, drivers virtualize devices, the file system virtualizes the storage disk, and virtual addresses virtualize physical addresses. In particular, this last example of addressing memory locations forms part of the *virtual memory* (VM) system. VM has been introduced with three goals in mind: (1) providing the illusion of an endless memory, (2) isolating the memory of different processes, and (3) enabling memory protection of *virtual memory areas* (VMAs).

When it comes to protecting memory, the OS delegates everything to the *memory management unit* (MMU), which is responsible for translating addresses using *page tables* (PTs) and *translation lookaside buffers* (TLBs), where the content of PTs is "cached". A typical MMU, whose design is sketched on Figure 1.1, is able to manage protection rights on so-called *pages*, which are essentially fixed-sized chunks of contiguous memory addresses. These protection rights on virtual memory are encoded on two control bits of the *page table entries* (PTEs), and are translated to the OS software as `PROT_NONE`, `PROT_READ`, `PROT_WRITE` or `PROT_EXEC`.



(a) Memory Management Unit (b) Address Translation

Figure 1.1 – Virtual Memory Management in the OS

Of course, a process can change these rights by using the `mprotect()` *system call* (syscall), which works at the page granularity, but this invokes a kernel function, which incurs two costly context switches as well as a TLB flush. As a result, page-based memory protection rights have significant overhead if they are updated too frequently. To mitigate this issue, Intel has recently added a new CPU feature, which enables *memory protection keys* (MPKs). These keys are manageable by user-space functions, which makes them particularly suitable for tackling the challenges raised above. As explained in [14], four previously unused bits of each PTE (bits 62:59, as shown on Figure 1.2) are used to hold the associated MPK, yielding 16 keys in total. When such a key is set, its associated protection rights are checked upon each access, additionally to the page table protections.

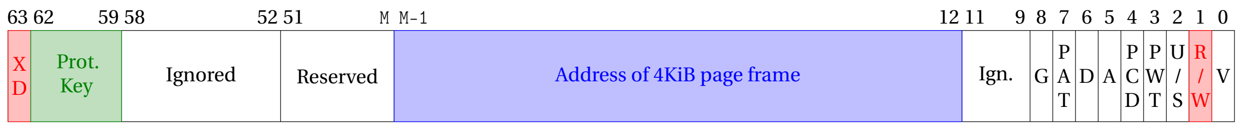

63 62 59 58 52 51 M M-1 12 11 9 8 7 6 5 4 3 2 1 0
XD
Prot. Key
Ignored
Reserved
Address of 4KiB page frame
Ign.
G
PAT
D
A
PCD
PWT
U/S
R/W
V

Figure 1.2 – Page Table Entry in Intel 64 and IA-32 Architectures (adapted from [12])

In order to manipulate MPKs, Linux offers three new syscalls: `pkey_alloc()`, `pkey_free()` and `pkey_mprotect()`. FreeBSD, on the other hand, provides four pseudo-syscalls that are accessible through the `sysarch()` syscall: `x86_pkru_get_perm()`, `x86_pkru_set_perm()`, `x86_pkru_protect_range()` and `x86_pkru_unprotect_range()`. Since these calls are not yet part of a standard, their usage is subject to specific versions of the kernel. For example, this project leverages FreeBSD 13.0 to make sure everything is set up to work as expected.

Moreover, some limitations remain to MPKs when they are used as a raw primitive: (1) protection keys may be reused after being freed, (2) the number of *physical keys* (pkeys) is limited to 16, and (3) they are per-thread and thus do not integrate well with process-based protection.

Thus, a project that eases MPK handling has been published under the name of LibMPK [7]. The paper that introduces this library lists eight functions that should be used for dealing with MPKs, and these have been implemented on Linux 4.14.2. (Ubuntu 16.04) [8], calling among other new syscalls the existing three that were created for the use of Intel MPKs. The three key contributions of said paper are (1) key virtualization for unlimited allocation, (2) lazy synchronization of key permissions across threads, and (3) metadata integrity to minimize the number of additional syscalls.

The goal of this semester project is to port the LibMPK code to FreeBSD 13.0, which just like Linux 4.14.2 also has specific functions for MPK manipulation, while staying as close as possible to the Linux code that is provided on the *GitHub repository* (repo) of the LibMPK project [8]. As such, the project involved modifying the user-space library, adapting the install script and creating new syscalls in the kernel. Regarding this report, the project description will be laid out as follows: in order, we discuss the background, sketch the design, explain some implementation details and then evaluate the code.

Chapter 2

Background

To understand the port of LibMPK from Linux to FreeBSD, there are some key concepts to understand. First, the PKRU register for access rights holds two bits per pkey, which will be illustrated in section 2.1. Second, Intel has introduced two new assembly instructions which are exploited by software in the two OSes we consider, as explained in section 2.2. Third, LibMPK leverages these instructions on Linux to provide a practical interface to the programmer, which will be covered in section 2.3. Let's look at these three elements one by one.

2.1 Protection Key Register for User-Space (PKRU)

A memory protection key (MPK) indexes a so-called protection *domain* that "colors" the address space of a thread into partitions with different access rights. The benefit of this additional level of indirection is that when an application changes protection domains, there is no need to access and modify the page table, since the key is already set. Therefore, each thread has its own *protection key register for user-space* (PKRU), which can selectively protect user-mode pages depending on the 4-bit key assigned to each page. For each key, there are two bits in the PKRU that describe its associated access and write permissions, as laid out on Figure 2.1.

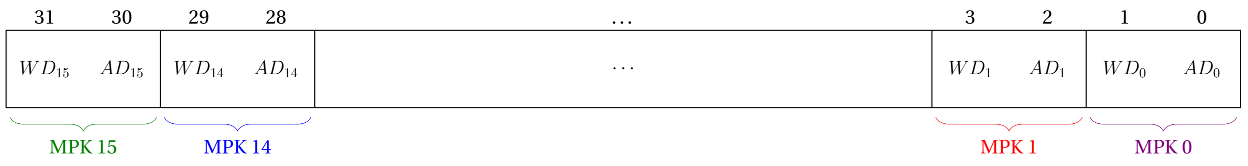


Figure 2.1 – Per-Thread PKRU on 32 Bits

More precisely, the figure shows that key number i has two corresponding bits in the PKRU, which are AD_i and WD_i:

- **Access Disable** AD_i (bit $2i$): if set, the processor prevents any data access to user-mode linear addresses with protection key i;
- **Write Disable** WD_i (bit $2i+1$): if set, the processor prevents write accesses to user-mode linear addresses with protection key i (but supervisor-mode write accesses are authorized if `CR0.WR == 0`).

Both of these restrictions only apply to data accesses and not to instruction fetches. As such, this covers memory operations like pointer dereferencing, functions like `memcpy()`, but also any syscall that accesses memory on behalf of the considered application. Indeed, it is really the calling thread that holds the PKRU, and it is managed by user-space function calls. In other words, storing protection rights into the PKRU is completely handled by software.

In the kernel, for both Linux and FreeBSD, the MMU uses the following macros to encode the protections of a memory page on two bits:

```
#define PROT_NONE      0x00
#define PROT_READ      0x01
#define PROT_WRITE     0x02
#define PROT_READWRITE 0x03
#define PROT_EXEC      0x04
```

As shown on Figure 1.2, these are mapped to two specific bits in the corresponding page table entry:

- *Execute Disable* XD (bit 63): if set, the page cannot be executed (*i.e.* it is an active-low bit);
- *Read or Write* R/W (bit 1): if set, the access is writable, otherwise it is read-only.

Since MPKs are only checked upon data accesses and not instruction fetches, execution rights are not considered in the PKRU, but read an write permissions need to be covered by the two bits we introduced earlier, namely WD_i and AD_i. Therefore, for a given permission key i, the four combinations shown in Figure 2.2 are possible on the two bits corresponding to an MPK.

WD_i	AD_i	**Effect**	**Description**
0	0	Accesses not disabled, writes not disabled	Read+Write access
0	1	Accesses disabled, writes not disabled	No access
1	0	Accesses not disabled, writes disabled	Read-only access
1	1	Accesses disabled, writes disabled	No access

Figure 2.2 – Possible Rights for a Memory Protection Key

This interface is important for system designers, since they implement user-space access to the PKRU. When it will come to programmers dealing with software management of MPKs (with or without LibMPK), only the macros shown previously are important, since they are translated automatically to the corresponding WD_i and AD_i bits.

2.2 Software Access to the PKRU

When it comes to reading from and writing to the PKRU, as described in the previous section, three interfaces will be described in this background chapter: the new Intel assembly instructions, the Linux syscalls and the FreeBSD syscalls. The interface provided by LibMPK will then be introduced in the next section.

2.2.1 Assembly Instructions

Intel has introduced two new instructions to their *instruction set architecture* (ISA) which are associated to the and its manipulation:

- `RDPKRU` (Read PKRU): Reads the PKRU into register EAX and clears register EDX (ECX must be 0 and CR4.PKE must be 1 when `RDPKRU` is executed);
- `WRPKRU` (Write PKRU): Writes EAX into the PKRU (ECX and EDX must be 0 and CR4.PKE must be 1 when `WRPKRU` is executed).

These two instructions can be wrapped for easy inlining in C code as follows:

```
void wrpkru(unsigned input) {
  asm volatile("wrpkru\n" : : "a"(input), "c"(0), "d"(0) :);
  return;
}

unsigned rdpkru() {
  unsigned eax;
  asm volatile("rdpkru\n" : "=a"(eax) : "c"(0) :);
  return eax;
}
```

Of course, while these two functions work, they are widely impractical to use and also quite error-prone, since the entire register with information for all MPKs needs to be changed at once. Instead, OSes provide more user-friendly APIs, as the next two subsections describe.

2.2.2 Linux System Calls

There are three system calls in Linux 4.14.2 which directly interact with protection keys:

```
int pkey_alloc(unsigned long flags, unsigned long init_access_rights);
int pkey_free(int pkey);
int pkey_mprotect(unsigned long start, size_t len, unsigned long prot, int pkey);
```

The typical workflow therefore consists of allocating a key, then associating protections to it on specific memory areas, and finally freeing it once the thread is done. Alongside these syscalls, there is also a user-space function which wraps the assembly instructions presented earlier:

```
int pkey_set(int pkey, unsigned long rights, unsigned long flags);
```

The goal of this function, as opposed to `pkey_mprotect()`, is only to change protections of a key, but not to change its binding with a specific memory address.

For example, the following code snippet creates `real_prot` as actual MMU protections, but additionally sets the pkey to `PKEY_DISABLE_WRITE`:

```
int real_prot = PROT_READ | PROT_WRITE;
int pkey = pkey_alloc(0, PKEY_DISABLE_WRITE);
void* ptr = mmap(NULL, PAGE_SIZE, real_prot, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
pkey_mprotect(ptr, PAGE_SIZE, real_prot, pkey);
pkey_set(pkey, 0);  // clear PKEY_DISABLE_WRITE
*ptr = foo;  // No segmentation fault
pkey_set(pkey, PKEY_DISABLE_WRITE);  // set PKEY_DISABLE_WRITE again
// *ptr = bar;  // Segmentation fault
munmap(ptr, PAGE_SIZE);
pkey_free(pkey);
```

2.2.3 Memory Protection in FreeBSD

The above syscalls are defined on Linux. However, other OSes may run on the new Intel architecture that provides hardware support for MPKs. In particular, there are already existing functions to deal with them in FreeBSD 13.0:

```
int x86_pkru_get_perm(unsigned int keyidx, int *access, int *modify);
int x86_pkru_set_perm(unsigned int keyidx, int access, int modify);
int x86_pkru_protect_range(void *addr, unsigned long len, unsigned int keyidx, int flag);
int x86_pkru_unprotect_range(void *addr, unsigned long len);
```

These functions, accessible through the `sysarch()` syscall, enable MPK mechanisms besides simple `mmap()` and `mprotect()` calls. Internally, per-key permissions are still managed using the user-mode instructions `RDPKRU` and `WRPKRU`, though. However, there is notably no available interface for allocating and freeing keys. In other words, the programmer is fully responsible of choosing appropriate keys in the integer range $[\![0, 16[\![$. Also, note that the `access` and `modify` parameters are "active-low", *i.e.* disabling their corresponding rights when set.

The equivalent behavior to the previous snippet for Linux can be achieved in FreeBSD by writing the following code, but now with manual MPK allocation:

```
int real_prot = PROT_READ | PROT_WRITE;
int pkey = baz;  // no explicit pkey allocation available (baz in [0, 16[)
void* ptr = mmap(NULL, PAGE_SIZE, real_prot, MAP_ANONYMOUS | MAP_PRIVATE, -1, 0);
ret = x86_pkru_protect_range(ptr, PAGE_SIZE, pkey, 0);
x86_pkru_set_perm(pkey, 0, 0);  // set MODIFY DISABLE to 0
*ptr = foo;  // No segmentation fault
x86_pkru_set_perm(pkey, 0, 1);  // set MODIFY DISABLE to 1
// *ptr = bar;  // Segmentation fault
munmap(ptr, PAGE_SIZE);
x86_pkru_unprotect_range(ptr, PAGE_SIZE);  // no explicit pkey freeing available
```

This naturally calls for a more automatic allocation scheme, which will also be part of the job when porting the LibMPK library described in the following section.

2.3 "LibMPK: Software Abstraction for Intel Memory Protection Keys (Intel MPK)"

Intel MPKs provide a practical interface to deal with page-based protection. In particular, they have three key advantages over traditional memory protection from the OS MMU:

1. **Performance**: all data protection rights are kept in a specific register (rw/r/.) by a fast non-privileged instruction;
2. **Group-wise control**: a single instruction can update up to 16 page groups at once;
3. **Per-thread memory view**: each thread has its own register for MPKs.

These points seem to promise a bright future to MPKs, but on the other hand, they also have three major problems, which mitigate their practicality for now:

1. **Security**: protection keys may be "used-after-free" because pages are not invalidated upon `pkey_free()`;

2. **Scalability**: only 16 MPKs can be used at once due to hardware limitations, so the application programmer has to multiplex the keys manually;

3. **Semantic inconsistency**: traditional `mprotect()` has per-process view instead of per-thread view, which is bad for legacy applications that multi-thread, as all threads should have the same access rights.

To solve these problems and still enjoy the benefits of MPKs, the authors of [7] wrote the LibMPK user-space library that comes with some kernel modifications. More precisely, LibMPK addresses the issues with:

1. **Key virtualization**: the MPK use-after-free problem is eliminated and unlimited page groups can be created;

2. **Lazy inter-thread key synchronization**: per-process semantics can be selectively enabled if needed;

3. **Metadata integrity**: even though not strictly needed, this last feature is useful to minimize the number of syscalls since it makes LibMPK compatible with standard `mprotect()`.

One non-negligible challenge I have faced when trying to adapt LibMPK is that the notation across the published paper, the conference slides and the GitHub repo was widely inconsistent, whether it be for function names, argument names or even the number of function parameters. In order to make progress, I thus decided to discard both the paper *application programming interface* (API) and the examples in the slides to stick as closely as possible to the notation of the actual code that is available online. Therefrom, my port to FreeBSD offers the following interface:

```
// API defined (with different naming) in the LibMPK paper
int mpt_init(int evict_rate);
int mpt_mmap(void** addr, size_t length, int prot, int flags);
int mpt_mprotect(int id, int prot);
int mpt_destroy(int id);
int mpt_begin(int id, int prot);
int mpt_end(int id);

// Additional functions available in the code base
int pkey_alloc(int flags, int permit);
int pkey_free(unsigned long pkey);
int pkey_mprotect(void *ptr, size_t size, unsigned long orig_prot, unsigned long pkey);
int evict_mprotect(struct mprot* m1, struct mprot* m2);
int pkey_sync(void);
int pkey_read(int idx);
```

Each of these functions will be described in detail in the following chapter, along with some insight about the calling stack of the implementation through the library and the kernel.

Chapter 3

Design

In this chapter, the design of the LibMPK API will be highlighted. First, all functions of the interface are documented in section 3.1. Second, the library and kernel parts will be distinguished in section 3.2. Third, an example diagram will show in section 3.3 how the programmer function call propagates to the kernel in a typical interaction with LibMPK. Hence, in a way, with each section, the design will be described more precisely, starting from the visible interface of the library and going towards at a complete view of a library call.

3.1 Application Programming Interface (API) of LibMPK

When designing a module, some parts are hidden from the user for simplification reasons. Indeed, the implementation details should typically not matter for the programmer calling a library, but the abstraction that is available forms an interface consisting of functions that are ready to be called. This set of exposed functions is referred to as an *application programming interface* (API), and it forms a controlled way in which different modules communicate. To summarize, and to link back to LibMPK, the implementation abstracts away how the library works under its cover and the API only exposes what it does with a set of functions, as shown on Figure 3.1.

First of all, it is important to note that the authors of LibMPK assume that MPKs are only accessed through the newly provided API. In other words, as soon as other functions are used, like for instance inlined assembly calls to WRPKRU or direct uses of OS-specific syscalls presented in section 2.2. This ensures that PKRU manipulation be always safe, since the interface abstracts away the details of uncontrolled calls, but it also emphasizes the importance of understanding how to use each provided function. This is why they are described below, following the paper's descriptions [7], but with some additional information to clarify them even more.

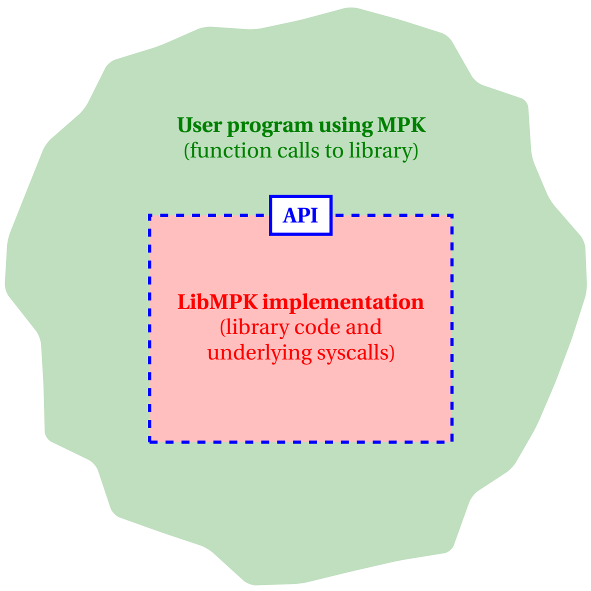


Figure 3.1 – API Interaction between User Program and LibMPK Implementation

First of all, we will go through the actual LibMPK functions, *i.e.* those which are meant to be used directly by the users of the library. Externally, they deal only with virtual keys, but internally they may call other functions, as will be alluded to in the descriptions that follow.

- `int mpt_init(int evict_rate);`

  The `mpt_init()` function initializes LibMPK, which consists in pre-allocating all hardware pkeys and setting up metadata, like the eviction rate for example. The latter sets the frequency at which protections applied by protection keys get evicted to page permissions using `mprotect()`. Indeed, LibMPK uses the protection keys as a "cache" for regular page-based protections.

- `int mpt_mmap(void** addr, size_t length, int prot, int flags);`

  The `mpt_mmap()` function allocates a set of pages associated with a *virtual key* (vkey, or in the code `id`). The arguments are basically passed to the `mmap()` syscall, and a new virtual key is generated for the current mapping. As a result, the pointer passed as `addr` is a hint to the kernel and may be changed by the underlying call.

- `int mpt_mprotect(int id, int prot);`

  The `mpt_mprotect()` function changes the permissions for a page group with vkey `id` to the ones given by `prot` and synchronizes these changes across threads by calling the "internal" function `pkey_sync()`.

- `int mpt_destroy(int id);`

  The `mpt_destroy()` function corresponds to a call to `munmap()`, since it unmaps all pages associated to vkey `id` by destroying the page group.

- `int mpt_begin(int id, int prot);`

  The `mpt_begin()` function creates permissions for the current thread to access a page group indexed by vkey `id` using the permissions given by `prot`. As such, it is the function to place at the beginning of a domain that should be isolated.

- `int mpt_end(int id);`

  The `mpt_end()` function is the dual to `mpt_begin()` when it comes to domain-based isolation. Its functionality consists in releasing the permissions for the given page group indexed by vkey `id` in the current thread.

The rest of the functions are not necessarily exposed to the user directly, but they nevertheless leak from the abstraction upon installing LibMPK, since they are defined in the corresponding header files. Their implementation merely forwards the received arguments to a syscall dealing with pkeys, *i.e.* physical MPKs.

- `int pkey_alloc(int flags, int permit);`

  The `pkey_alloc()` function allocates a pkey and initializes its corresponding bits in the PKRU to `permit`. It is called notably by `mpt_init()` when it pre-allocates all pkeys for the current thread. In Linux, this syscall exists already, but it does not in FreeBSD, which is why the mechanism needs to be coded, and this is done in the kernel.

- `int pkey_free(unsigned long pkey);`

  The `pkey_free()` function releases the given pkey if it has been allocated previously. To check this, the kernel needs a per-thread allocation map, which is stored on 16 bits (one for each pkey). Again, this is a syscall in Linux, but not in FreeBSD.

- `int pkey_mprotect(void *ptr, size_t size, unsigned long orig_prot, unsigned long pkey);`

  The `pkey_mprotect()` function is the last one that is ported from Linux, and it corresponds to protecting a memory region with a pkey, and with some permissions given by `orig_prot`.

- `int evict_mprotect(struct mprot* m1, struct mprot* m2);`

  The `evict_mprotect()` function takes two `struct mprot` pointers, each of which basically containing four components corresponding to the information given as parameters to `pkey_mprotect()`. This function takes two of such structures as parameters, but the reason why this is the case is not very clear from the paper. The function only uses an underlying syscall, but does not seem to be used otherwise. Note that eviction is taken care of by another syscall in the LibMPK implementation, as will be shown in section 3.2.

- `int pkey_sync(void);`

  The `pkey_sync()` function also needs the kernel, since its goal is to synchronize PKRUs across threads. This aspect is not yet on point in the original code base [5], so I am not emphasizing my focus on this aspect.

- `int pkey_read(int idx);`

  The `pkey_read()` function is simply a wrapper for reading the PKRU at the index given as parameter. It is the dual to `pkey_set()`, which is defined in section 2.2, so that they both exist in Linux. On FreeBSD, the situation is a bit different since both already exist (under the names specified in said section).

To summarize this API description, the user-space library contains a certain amount of functions that are available for use, and they should be the only ones used to manipulate the PKRU. In other words, with a unified interface to memory protection keys, the disparities between the Linux and FreeBSD syscalls presented earlier should become insignificant with the port from the former OS to the latter.

3.2 User-Space Library and Kernel-Space System Calls

A key principle for successful computer systems is the one of layering [3], whereby modules are stacked "vertically" on top of each other, so that each of them only communicates with modules directly above or below its own layer. While this kind of architecture model is observed widely in computer networking, OS design also divides its functionality into essentially two layers: user- and kernel-space. The controlled interface enabling communication between the two is the one of system calls. Further, user-space can be subdivided into applications, written by the programmer; and libraries, interacting with the OS when needed. As such, when a user-space process wants to invoke the kernel in some way, it uses a library function, which in turn performs a `syscall()` that traps into a pre-defined code location in the kernel with full access to the hardware. This model is illustrated on Figure 3.2, which represents a *monolithic kernel*, because both Linux and FreeBSD – and more generally Unix – follow this kind of structure.

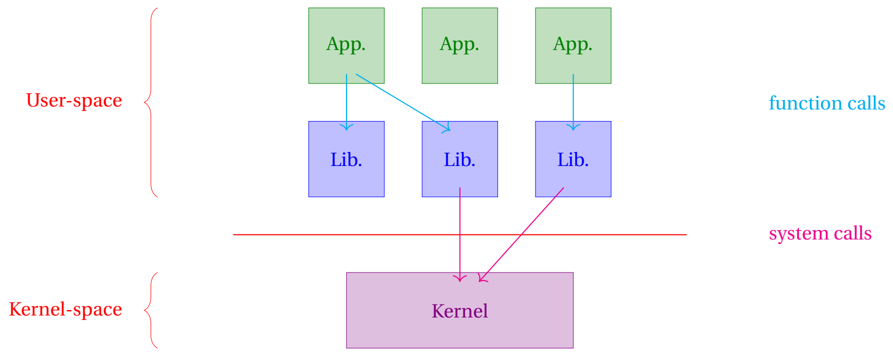


Figure 3.2 – Simplified Monolithic Kernel Model

More precisely, as explained in [13], *user-space* has its own memory, where untrusted code and most of the application data is located. Furthermore, it runs in unprivileged mode, which is why it does not have access to low-level system primitives like virtual memory mappings. On the other hand, *kernel-space* also has its own memory, which is usually smaller than the one in user-space, but it contains only kernel code, *i.e.* functions that are trusted, and thus runs in privileged mode. Finally, the kernel can copy user-space memory into its own memory if needed, using functions like `copy_from_user()` on Linux or `copyin()` on FreeBSD.

In the LibMPK library, the API that has been defined in the previous section corresponds to the library part of the design, *i.e.* the one that runs in user-space. Moreover, as has been alluded to several times, some syscalls are necessary in order to ensure proper interaction with the virtual memory system in the MMU. Indeed, since protection checks need to be performed at every access to memory, permissions are encoded in the page table entries, and to modify these we need to kernel to intervene. For this reason, the authors of LibMPK have written a certain amount of syscalls along with their library, in order to patch the Linux kernel for the purpose of their functionality.

When it comes to the adaptation of the library to FreeBSD, I have ported most of these syscalls, but at this point it seems like some of them are never used by the original library. Also, I wrote some of them in the beginning of the project without being sure they would even be necessary, just to be sure I have a superset of the expected functionality. Nevertheless, I present all of them briefly below, as I have done with the previous API, so as to reflect the work that has actually been done over the semester in the most accurate way.

- `int pkey_mprotect(struct thread *td, void *ptr, size_t size, unsigned long orig_prot, unsigned long pkey);`

  The `pkey_mprotect()` syscall is meant to be analogous to the one that Linux offers natively. Its goal is to protect a page group with a given pkey and some permissions, but it already has an equivalent `sysarch()` pseudo-syscall, which makes it a bit redundant. However, there are several syscalls below that are used as proxies for this one by LibMPK, which is why it is kept as it is.

- `int pkey_alloc(struct thread *td, unsigned long flags, unsigned long init_val);`

  The `pkey_alloc()` syscall allocates a pkey if still available, and assigns it the given `init_val` as protections in the PKRU. Internally, it also marks the pkey as allocated for the current thread it its allocation map.

- `int pkey_free(struct thread *td, unsigned long pkey);`

  The `pkey_free()` syscall frees the pkey given as parameter in the allocation map.

- `int pkey_evict(struct thread *td, struct mprot* m1, struct mprot* m2);`

  The `pkey_evict()` syscall corresponds to the `evict_mprotect()` library function, and the reason why it takes two arguments it unclear. Mostly, it protects the two memory regions described in the parameters.

- `int pkey_sync(struct thread *td, unsigned int val_pkru);`

The `pkey_sync()` syscall is meant to synchronize the PKRU across threads with the PKRU value given by `val_pkru`.

- `int int mpk_mmap(struct thread *td, void* addr, unsigned long len, unsigned long prot, unsigned long flags, int id);`

  The `mpk_mmap()` syscall is probably a backward compatibility feature, since the library call `mpt_mmap()` does not call it. It has been ported to FreeBSD, but will eventually be removed if the library does not use it.

- `int pkey_mprotect_set(struct thread *td, void* start, unsigned long len, unsigned long prot, int pkey, int id);`

  The `pkey_mprotect_set()` syscall is a way to create the mapping from a vkey to a pkey in the kernel as well. It also calls the `pkey_mprotect()` syscall right after.

- `int pkey_mprotect_evict(struct thread *td, void* start, unsigned long len, unsigned long prot, int pkey, int id);`

  The `pkey_mprotect_evict()` syscall is a way to evict the mapping from a vkey to a pkey in the kernel as well. It also calls the `pkey_mprotect()` syscall right after.

- `int pkey_mprotect_grouping(struct thread *td, void* start, unsigned long len, unsigned long prot, int grouping_key, int id);`

  The `pkey_mprotect_grouping()` syscall is a way to group the mapping from a vkey to a pkey in the kernel as well. It also calls the `pkey_mprotect()` syscall right after.

- `int pkey_mprotect_exec(struct thread *td, void* start, unsigned long len, unsigned long prot, int pkey, int id);`

  The `pkey_mprotect_exec()` syscall is a way to handle execution rights, since these cannot be captured by the PKRU, which only encodes read/write permissions. Thus, they are handled by `mprotect()` here.

- `int mprotect_exec(struct thread *td, void* start, unsigned long len, unsigned long prot, int pkey);`

  The `mprotect_exec()` syscall is very similar to `pkey_mprotect_exec()`, but it applies to the pkeys directly instead of going through the vkey indirection.

- `int mpk_munmap(struct thread *td, void* addr, size_t len, int id);`

  The `mpk_munmap()` syscall unmaps all pages with vkey `id`, which is why the kernel also keeps track of the vkey to pkey mappings.

Note that all of these syscall prototypes receive `struct thread` information as their first parameter: this is not only because the PKRU is per-thread, but mainly since syscalls in FreeBSD are defined in that way, *i.e.* their prototype must start as such.

3.3 Sequence Diagram Across the System

From the previous sections, the top-level abstractions of both user- and kernel-space have been specified. However, these interfaces do not define the entire behavior of a library call, since the underlying code does much more than just return. Moreover, the LibMPK API is not documented very clearly online, which justifies the importance of having a big-picture view across the different OS layers. In software engineering, the *unified modeling language* (UML) is sometimes used for such behavioral diagrams.

Now that (a superset of) the API in terms of library functions and system calls has been covered, let us take the example of a call to `mpt_init()`, which for the record sets up the LibMPK environment by allocating all pkeys, setting the eviction threshold and allocating memory for the hash table which keeps the mappings between vkeys and pkeys. In Figure 3.3, the sequence diagram across the system corresponding to this function is represented as clearly as possible in approximate UML, with the same color code as in Figure 3.3.

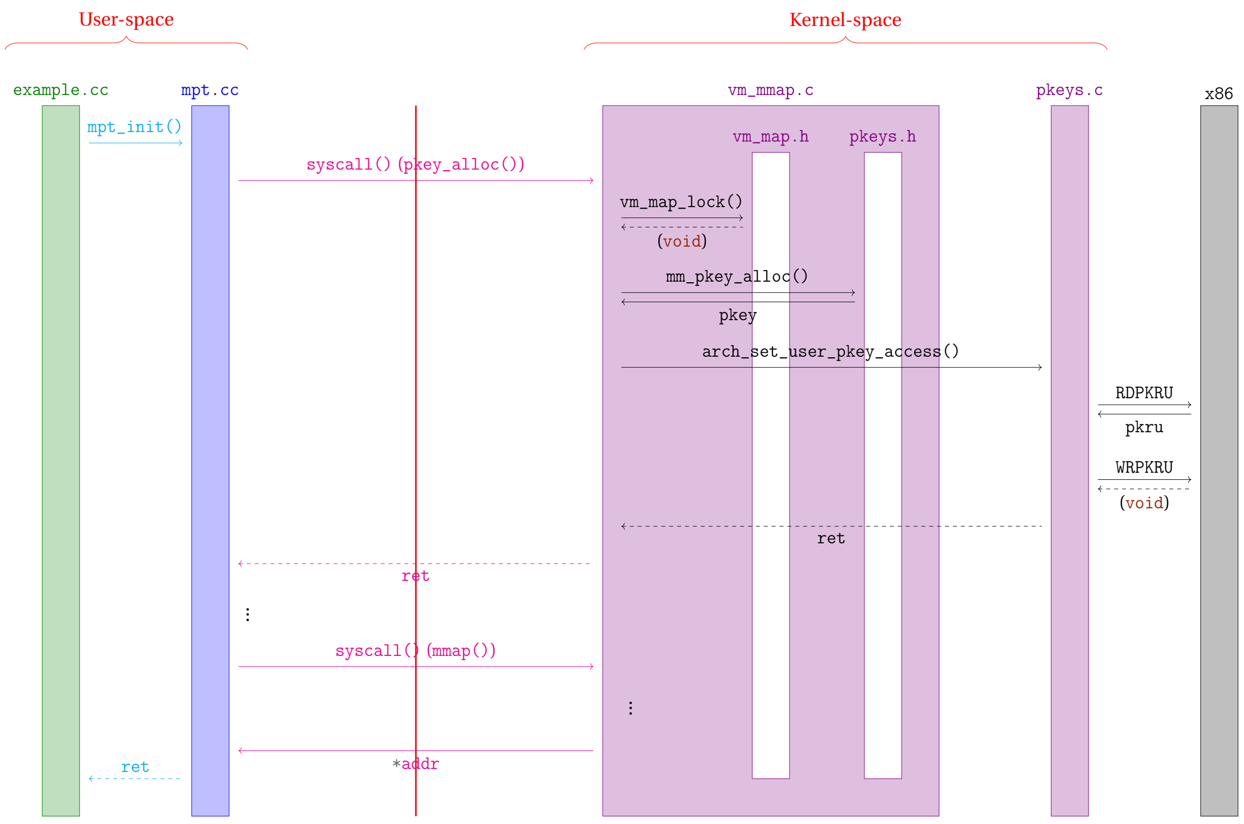


Figure 3.3 – Diagram for a successful call to `mpt_init()`

On this figure, the trace of a `mpt_init()` call is shown across the system layers. First, the call is transmitted from the user program to the LibMPK library, which is written in C++. Then, among other manipulations, a syscall to `pkey_alloc()` is operated. In the next chapter, I will explain why the relevant syscalls are defined at two different places, namely `libmpk.c` and `vm_mmap.c`, but the ones needed here come from the latter.

Now, since header files are not compiled themselves, but copied into each source file that includes them, the two calls to `vm_map.h` and `pkeys.h` are represented as internal flow in a nested way. Indeed, these calls actually reach either macros or static functions, which do not need to be in a source file, and my goal is to remain as close as possible to the "official" LibMPK implementation.

Finally, the kernel file `pkeys.c` has the responsibility of calling the x86 assembly instructions RDPKRU and WRPKRU, which corresponds to the interaction with the PKRU register. After returning from these functions, the call to `mmap()` proceeds as in the unpatched kernel, since this part has not been modified by LibMPK.

For the sake of time and brevity, other sequence diagrams are omitted, but the idea of this section was rather to give an overall intuition about how the files interact with each other. In other words, Figure 3.3 is the kind of picture that would have helped me to get an insight about LibMPK at the beginning of the project. More generally, this entire report is written in a way that would help a newcomer understand what the project is about.

Chapter 4

Implementation

In the previous chapter, the design of LibMPK was highlighted in a way that makes it quite clear how the different components interact; now, it is time to dive into some implementation details that are relevant about this project. To find the entire implementation, the code is hosted on `https://gitlab.epfl.ch/fgerber/libmpk-freebsd`, but I will not be able to discuss everything in this report. Instead, I will focus on some relevant aspects that were challenging. First and foremost, the process of adding syscalls in FreeBSD will be outlined in section 4.1. After that, some debugging steps will be described in section 4.2. Finally, section 4.3 will show how the library has been adapted to interact with the kernel.

4.1 Adding System Calls to FreeBSD

Besides adapting small bits and pieces here and there to move from Linux to FreeBSD compatibility, the library did not require major changes. Syscalls, on the other hand, follow not only a declaration and implementation scheme different on FreeBSD from the one on Linux, but they also call other kernel functions which do not necessarily exist in the same way in both OSes. Therefore, this section explains the steps that have been performed in order to define new syscalls in FreeBSD, following the documentation on [2]. This detailed article will be synthesized into three steps that should be clear enough for someone to reproduce: registering the syscalls, implementing their code and compiling the resulting kernel.

4.1.1 Registering the System Calls

The first step to add new syscalls to FreeBSD consists in modifying the file `sys/kern/syscalls.master`, which contains an entry for each syscall. As a result, this step is not really difficult, since it merely consists in adapting existing lines to match the prototype or the new syscall. Since the existing syscalls were numbered up to 579, the LibMPK syscalls could start at 580. Here is for instance the first one of them:

```
580 AUE_NULL STD {
  int pkey_mprotect(
    void *ptr,
    size_t size,
    unsigned long orig_prot,
    unsigned long pkey
  );
}
```

Along with adding such a registration for all syscalls, there is also a compatibility file for 32-bit systems that should be completed in an ideal world, but in the current project this has been left out since testing is done on a 64-bit system, and these are more and more common anyways.

Once these function-like prototypes are defined, the next step consists in regenerating some system files by running `make sysent` in the two directories `sys/kern` and `sys/sys`. This modifies the following files:

- `sys/kern/init_sysent.c`;
- `sys/kern/syscalls.c`;
- `sys/kern/systrace_args.c`;
- `sys/sys/syscall.h`;
- `sys/sys/syscall.mk`;
- `sys/sys/sysent.h`;
- `sys/sys/sysproto.h`.

For example, in `sysproto.h`, this generated the following structure encapsulating the arguments of the syscall in the previous example:

```
struct pkey_mprotect_args {
  char ptr_l_[PADL_(void *)]; void * ptr; char ptr_r_[PADR_(void *)];
  char size_l_[PADL_(size_t)]; size_t size; char size_r_[PADR_(size_t)];
  char orig_prot_l_[PADL_(unsigned long)]; unsigned long orig_prot;
    char orig_prot_r_[PADR_(unsigned long)];
  char pkey_l_[PADL_(unsigned long)]; unsigned long pkey;
    char pkey_r_[PADR_(unsigned long)];
};
```

In total, all 12 syscalls described in section 3.2 were added in this way, as visible in the comments that form part of `init_sysent.h`:

```
/* 580 = pkey_mprotect */
/* 581 = pkey_alloc */
/* 582 = pkey_free */
/* 583 = pkey_evict */
/* 584 = pkey_sync */
/* 585 = mpk_mmap */
/* 586 = pkey_mprotect_set */
/* 587 = pkey_mprotect_evict */
/* 588 = pkey_mprotect_grouping */
/* 589 = pkey_mprotect_exec */
/* 590 = mprotect_exec */
/* 591 = mpk_munmap */
```

Finally, these syscalls have to be listed by name in `lib/libc/sys/Symbol.map` under the most recent version of FreeBSD that is present. For this project, it was `FBSD_1.6`.

4.1.2 Implementing System Call Code

Once syscalls are correctly registered, it remains to implement them at an appropriate place. For this project, while not all syscalls are needed in the end, all of the ones that were present in the Linux implementation of LibMPK have nevertheless been ported to FreeBSD for the sake of completeness. With this caveat, the modified file structure looks as follows:

```
/usr/src
|-- sys
    |-- kern
    |   |-- libmpk.c         // implements the syscalls relative to pkey handling
    |   |-- pkeys.c          // does not implement syscalls, but called from them
    |-- sys
    |   |-- libmpk.h         // lists prototypes of all LibMPK syscalls
    |   |-- libmpk_struct.h  // contains a struct declaration included in sysproto
    |   |-- pkeys.h          // lists prototypes for pkeys.c and static functions
    |-- vm
        |-- vm_mmap.c        // implements the syscalls relative to memory protection
```

For illustration purposes, let's again take `pkey_mprotect()` as an example. This syscall appears in two files:

- In `libmpk_struct.h`, the top-level kernel prototype is listed as

  ```
  int kern_pkey_mprotect(struct thread *td, void *ptr, size_t size,
                         unsigned long orig_prot, unsigned long pkey);
  ```

  This prototype is useful for other kernel functions to access this syscall, since they cannot invoke more privilege levels using the `syscall()` function.

- In `vm_mmap.c`, there are two places where the files is modified. First, at the top of the file, we insert:

  ```
  int sys_pkey_mprotect(struct thread *td, struct pkey_mprotect_args *uap) {
    return (kern_pkey_mprotect(td, uap->ptr, uap->size, uap->orig_prot, uap->pkey));
  }
  ```

  This is the actual header of the syscall: it uses the corresponding `struct` from `sysproto.h` as defined earlier, and unwraps its arguments to pass them to the kernel function.

  Second, at the bottom of the file, the actual implementation can take place:

  ```
  int kern_pkey_mprotect(struct thread *td, void *ptr, size_t size,
                         unsigned long orig_prot, unsigned long pkey) {
    return do_mprotect_pkey(td, (unsigned long)ptr, size, orig_prot, pkey);
  }
  ```

  Here, the implementation is offloaded to a static kernel function that is inserted before in the file, since it factorizes code together with the functionality that the standard `mprotect()` syscall already has. Indeed, the only that changes is the `pkey` parameter, which by default is set to `-1` in the call to `do_mprotect_pkey()` when there is none. For other syscalls however, the implementation is mostly done directly in this kernel function. For syscall implementations referenced in subsequent parts of the report, this last place is the one that matters when it comes to the actual code.

Note that the `pkey_mprotect()` syscall is implemented in `vm_mmap.c` because it deals with memory protection, but in fact syscalls relative to pkey handling are implemented analogously in `libmpk.c`.

4.1.3 Compiling the Resulting Kernel

Once all relevant files have been modified as expected, the remaining step is the compilation of the kernel. In practice, the development of syscalls is iterative and it takes several tens of compilation steps until the syscalls work as expected. The process is somewhat frustrating, but it makes a working syscall so much more enjoyable once it finally compiles and executes correctly.

However, there is one final step to be completed in order to compile the modified files: nowhere has it been mentioned so far that added files should be compiled. Indeed, it is necessary to modify `sys/conf/files` by appending the path to all source files that have been added to the kernel. In the case of this project, this consisted in typing the following two lines at the end of the file:

```
kern/pkeys.c standard
kern/libmpk.c standard
```

Thanks to this step, the `Makefile` that will be invoked for the compilation process will be able to find the source files containing our syscall implementations. Here, as alluded to earlier, `pkeys.c` is a source file dealing directly with pkey allocation, whereas `libmpk.c` contains definitions of syscalls that may invoke kernel functions from the former. Note that `vm_mmap.c`, which is the second source file containing syscall implementations in our case, need not be added because it already existed before, in the unmodified kernel.

With this, we arrive at the point where we compile the kernel. To do this, one navigates to `/usr/src` and launches:

```
make buildkernel NO_CLEAN=YES
```

This command builds the kernel, but does not install it right away. Additionally, it saves intermediate steps, so that in the likely case of a compilation error, the build process does not have to start over from scratch, but can leverage existing compiled files to repeat only what is necessary. This is especially important on slow machines, since building the kernel can take up to several hours, and errors will happen in most cases.

Once the compilation succeeds, one can launch:

```
make installkernel
```

This step actually installs the built kernel so that the machine will also boot on it once restarted, *e.g.* with `shutdown -r now`. Additionally, it saves the old kernel, so that the machine may still be booted in case the kernel modifications rendered it unbootable.

4.2 Kernel Debugging

In the case of this project, I have encountered several compilation errors and will thus make a synoptic survey of them, so as to give a taste of debugging steps that may be useful:

1. **The `struct mprot` was initially not recognized.**

The structure in question is actually a simplification of the arguments passed to `pkey_evict()`, but it made things unusable for a while. Since this structure is directly part of the syscall prototype, it must be recognized in `syscalls.master` already, which is why I moved its declaration to the dedicated file `libmpk_struct.h`.

2. **All symbols starting with `sys_` were undefined.**

   Actually, I wasn't aware of the step just before compiling, which consisted in adding the source files to compile to the `files` text file. This being done, the problem was solved immediately.

3. **When adapting the Linux implementations of syscalls, some functions and types were unrecognized.**

   Indeed, it turns out that some types in Linux, like `u16` or `u32`, are not included in the same files in FreeBSD. Moreover, MPK syscalls do not take the same kinds of memory addresses: while Linux works with `unsigned long` addresses, FreeBSD uses `void*`, which is much more natural. Finally, Linux functions like `kzalloc()` do not exist in FreeBSD, and one has to find their way around them, for instance by using a simple `malloc()`.

4. **FreeBSD's kernel `malloc()` takes three arguments.**

   In fact, `malloc()` requires descriptive arguments in the FreeBSD kernel. This basically corresponds to invoking one (or optionally two) macros before every `malloc()`. For example, in LibMPK, a hash table is allocated for the mappings between vkeys and pkeys. This is performed as follows, following the *manpage* (manual page) of `malloc(9)`:

```
MALLOC_DECLARE(M_HASHTABLE);                                 // libmpk.h (optional)
MALLOC_DEFINE(M_HASHTABLE, "hashtable", "vkeys to pkeys");   // libmpk.c at top
table = malloc(TABLE_SIZE * sizeof(HashEntry),
               M_HASHTABLE, M_NOWAIT);                       // libmpk.c in function
```

5. **The virtual memory map locking process is different in FreeBSD.**

   To tackle this problem, I read some documentation on both Linux [1] and FreeBSD [15] OSes and found out that each had their own way of dealing with locks, as shown on Figure 4.1.

Linux	**FreeBSD**
`down_write(&current->mm->mmap_sem);`	`vm_map_lock(&td->td_proc->p_vmspace->vm_map);`
`// manipulate mmap_sem`	`// manipulate vm_map`
`up_write(&current->mm->mmap_sem);`	`vm_map_unlock(&td->td_proc->p_vmspace->vm_map);`

Figure 4.1 – Locking Examples on Virtual Memory Maps

Finally, there was an missing allocation error for the same hash table as discussed previously, during which I was brought to write a kernel driver for LibMPK, as if it were a device. This is actually done in the Linux implementation of LibMPK, but as of today it does not look like this is useful in FreeBSD, since said hash table can be allocated on the fly when it is first used.

4.3 Library Adaptation and Discussion

Once the syscalls are created, the library adaptation is quite simple. Quite naturally, one has to change the syscall numbers to the ones that have been newly created. But it turns out that there is actually more to be done: for example, the `mmap()` call considers LibMPK a memory-mapped device, which should not be the case for it to work on FreeBSD.

The general structure of the library looks as follows:

```
lib
|-- headers
|   |-- hash.h         // header file for internal hash table
|   |-- mpt.h          // header file for the API
|   |-- pkey.h         // header file for internal MPK manipulation
|-- heap
|   |-- CMakeLists
|   |-- install.sh
|   |-- mpk_heap.cc
|   |-- mpk_heap.h
|   |-- pkey.cc
|   |-- pkey.h
|-- install.sh
|-- Makefile
|-- mpt.cc             // source code for the API functions (interface)
|-- pkey.cc            // source code for internal MPK manipulation (syscalls)
```

As it appears on this tree, the headers are separated in their own directory, and there is an entire version for dynamic allocation on the heap. I have primarily focused on the non-heap version, though, since its interface is better explained in the paper.

All in all, the biggest challenge I faced when porting the library was that the big picture view was missing. In fact, the documentation of LibMPK is not so developed and I quickly dove into the syscall implementation without knowing what exactly they would be used for. In other words, the approach I took was bottom-up, *i.e.* starting with kernel functionality and then moving towards the library, when another way to see the project would have been top-down, *i.e.* the other way around. This would have enabled me to only implement the syscalls that are needed and to have a clear picture from the beginning. It is also how I would tackle the project if I had to start over now.

Chapter 5

Evaluation

In this chapter, I evaluate the port of LibMPK using some examples of programs. The experiments are performed on an Intel Comet Lake, but they were originally planned to be done on the FreeBSD 13.0 OS, but due to the remote work situation, I had to simulate some function calls on Linux 5.10.0. Other than that, most of the work when creating syscalls and compiling the kernel has been done on a QEMU virtual machine running the relevant version of FreeBSD.

Three sections will lay out how evaluation has been performed and which code examples have been run. First, the QEMU environment and its usage are presented in section 5.1. Second, some examples from the LibMPK paper are shown in section 5.2. Third, custom code tests are explored in section 5.3.

5.1 QEMU Virtual Machine

First of all, it is appropriate to spend some time on the QEMU virtual environment that was used to implement most of the code in kernel-space. It was built on a blank version of the FreeBSD 13.0 OS[1] found on `http://ftp-archive.freebsd.org/pub/FreeBSD-Archive/old-releases/ISO-IMAGES/13.0/FreeBSD-13.0-RELEASE-amd64-memstick.img`. Based on that, the disk image has been built and installed as follows under `/var/disks/`:

```
qemu-img create disk_FREEBSD13.img 15G
qemu-system-x86_64 -boot menu=on -m 8192 -smp 10 -hda disk_FREEBSD13.img
  -hdb FreeBSD-13.0-RELEASE-amd64-memstick.img
```

Once this step is done a single time, the virtual machine can be started as follows:

[1]I followed some notes on QEMU Andrés generously provided me. I also take this occasion to thank him for giving me access to his machine remotely.

```
qemu-system-x86_64 -boot menu=on -m 2048 -smp 1 /var/disks/disk_FREEBSD13.img -net
  user,hostfwd=tcp::40022-:22,hostfwd=tcp::40080-:80 -net nic -display none &!
```

I recommend launching the above command from a `screen`, since it does not stop once one disconnects from the remote machine. To connect to QEMU (when hosted on the above described remote computer), the command reads, with an empty root password:

```
ssh -p 40022 root@machine
```

where `machine` is the domain name of the machine where QEMU is running on.

Once this has been established, most of the transfers have been done over `git` and `scp`. For the latter, the option for entering the root password is an uppercase `-P` instead of the lowercase one in the `ssh` command. Thanks to this setup, I was able to debug kernel compilation and install the patched version of the kernel.

Unfortunately, according to the changelogs of QEMU, it seems the MPKs are not yet emulated as of version 6.2.50 that is available today. Therefore, all calls to the FreeBSD functions for MPKs described in section 2.2 return `-1` on this platform. As a result, I simulated the four FreeBSD functions for PKRU manipulation manually. As the following sections illustrate, this shows that if these functions do the right thing, the functionality of LibMPK can be made to work with the ported library.

5.2 Examples from the LibMPK Paper

In the LibMPK paper, two example use cases of the library are provided. Since they use the paper nomenclature, they are not compatible with the API exposed by the code – even the one on the official LibMPK GitHub repo. Thus, I adapted these two examples to compile and match the actual API that is provided by the code. In this section, both are tested on the modified library, relying on emulated `x86_pkru_get_perm()`, `x86_pkru_set_perm()` and `x86_pkru_protect_range()`.

```
void domain_based_isolation(void) {
  mpt_init(-1); // default eviction rate: 100%
  int* addr;
  int group_1 = mpt_mmap((void**)(&addr), 0x1000, PROT_READ | PROT_WRITE,
                         MAP_ANONYMOUS | MAP_PRIVATE);
  // page permission: rw- & pkey permission: --

  mpt_begin(group_1, PROT_READ | PROT_WRITE);
  // page permission: rw- & pkey permission: rw
  // write data in group_1
  *addr = 1;  // No segmentation fault

  mpt_end(group_1);
  // page permission: rw- & pkey permission: --

  // printf("%d\n", *addr); // Segmentation fault
}

void quick_permission_change(void) {
  mpt_init(-1); // set cache eviction rate: 100%
  int* addr;
  int group_2 = mpt_mmap((void**)(&addr), 0x1000, PROT_READ | PROT_WRITE,
                         MAP_ANONYMOUS | MAP_PRIVATE);
  // page permission: rw- & pkey permission: --

  // write data in group_2
  // *addr = 1; // Segmentation fault

  mpt_mprotect(group_2, PROT_READ | PROT_WRITE | PROT_EXEC);
  // page permission: rwx & pkey permission: rw

  // write data in group_2
  *addr = 1;  // No segmentation fault
}
```

As one can see on these example codes, the actual permissions that are enforced on an address really correspond to the intersection between those in the corresponding PTE and those in the PKRU at the positions of the pkey.

5.3 Custom Code Examples

Further, some experiments have been performed in order to check virtualization of the keys. These examples mainly have the goal to verify functionality along the way and check that pkeys can be accessed. Firstly, I checked that it was possible to create a large amount of virtual keys in a row:

```
void mmap_destroy_test(void) {
  mpt_init(-1);
  size_t size = 1;
  void *ptr;
  unsigned long orig_prot = PROT_READ | PROT_WRITE;
  unsigned long flags = MAP_PRIVATE | MAP_ANON;
  for (unsigned long i = 0; i < 100; ++i) {
    int vkey = mpt_mmap((void**)(&ptr), size, orig_prot, MAP_ANONYMOUS | MAP_PRIVATE);
    mpt_begin(vkey, orig_prot);
    printf("ptr : %p\n", ptr);
    mpt_end(vkey);
    mpt_destroy(vkey);
  }
}
```

Second, the following example is a new version of the example given in section 2.2. It illustrates the new unified interface of MPKs for both Linux and FreeBSD.

```
void initial_example_test(void) {
  mpt_init(-1);
  int real_prot = PROT_READ | PROT_WRITE;
  int* ptr;
  int vkey = mpt_mmap((void**)(&ptr), 4096, real_prot, MAP_ANONYMOUS | MAP_PRIVATE);
  mpt_mprotect(vkey, real_prot);  // enable read and write permissions
  *ptr = 0;  // No segmentation fault
  mpt_mprotect(vkey, PROT_READ);  // disable writes and only keep reads
  // *ptr = 0;  // Segmentation fault
  mpt_destroy(vkey);
}
```

These custom examples conclude the evaluation part of this report. In the last chapter, we will consider other related papers that deal with memory protection.

Chapter 6

Related Work

Memory protection has been a long-term challenge since the wide deployment of Unix-like OSes like FreeBSD. Some papers have been written about FreeBSD specifically, like [11], which provides a performant safety abstraction for FreeBSD 11.0, *i.e.* before MPKs were supported. As a result, this work relies on existing page-based protections, as it had always been done before, by leveraging the two bits in each page table entry that are dedicated to this purpose (bits 1 and 63, as shown on Figure 1.2). In this chapter however, I am aiming at presenting papers that build on top of the new MPK feature, since this is also what the considered library is doing.

Apart from LibMPK [7] and its attempt at solving issues of Intel MPKs, other researchers have also been playing around with the new hardware support for MPKs in recent x86 architectures. This chapter reviews three related papers: MemSentry protection of safe regions [10] in section 6.1, process isolation using ERIM [9] in section 6.2 and intra-unikernel isolation using MPKs [6] in section 6.3. A short section is dedicated to each of them.

6.1 "No Need to Hide: Protecting Safe Regions on Commodity Hardware"

This first paper we consider [10] was written at a time when Intel MPKs were only documented online, but not yet implemented on a real processor. Nevertheless, with the goal of protecting so-called "safe" regions deterministically in mind, as opposed to using techniques like *address space layout randomization* (ASLR), the authors of the paper separate two kinds of protections: address-based and domain-based protections. From the context, it is clear that address-based protections depend on the virtual address of the accessed data, as it is usually the case with `mprotect`; while domain-based protections may apply to different partitions that are not necessarily contiguous, as memory protection keys would be able to enforce.

In the domain-based approach, two domains are thus introduced: the sensitive domain and the non-sensitive domain, which correspond to set, respectively unset bits, in the corresponding two key bits of the PKRU. Given that MPK was not yet available on an x86 CPU when this work was accomplished, the authors simulated the PKRU register by copying another register in order to approximate their performance evaluation. The main limitation they identified is that since RDPKRU and WRPKRU instructions leverage parameter and return value registers, their use might cause unwanted register spilling on the compiler-side.

As a takeaway on MPKs, the paper states that using them probably achieves the best performance when it comes to domain-based isolation, but their conclusion is somewhat reserved since they did not know when MPKs would become available on Intel processors. Nevertheless, the register allocation issue is interesting to note, which is why I chose to document it here.

6.2 "ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK)"

One work that actually cites LibMPK as related work is ERIM, since both papers try to make MPKs more secure. In fact, the major drawback the ERIM paper [9] raises about MPKs is that since the its two ISA instructions are accessible to user-space, a malicious programmer can simply disable protections for any domain they want to access. As a result, ERIM uses binary inspection in search of unsafe WRPKRU instructions, to ensure that one cannot gain unauthorized access to sensitive data.

The binary inspection mechanism is designed around two simple checks, but safety can also be enforced by rewriting existing binary. Indeed, inspecting binaries basically consists in ensuring every WRPKRU instruction be followed either by a pre-defined jump to code of a trusted component, or by four specific assembly instructions guaranteeing that the program terminate in case the access was untrusted. Furthermore, the paper proposes a method to rewrite program binaries with compiler support, so that they do not contain unsafe WRPKRU instructions anymore.

Notably, the ERIM paper addresses a natural concern one might have about using MPKs from user-space. Additionally, together with the key virtualization implementation from LibMPK, it could be used for applications with a need for many different protection domains.

6.3 "Intra-Unikernel Isolation with Intel Memory Protection Keys"

The most recent of the three papers surveyed here is the one of intra-unikernel isolation [6]. It uses Intel MPKs because a unikernel address space is unified over user- and kernel-space, which breaks isolation inside the system. In order to enforce it nevertheless, it uses the per-thread nature of the PKRU, which makes it possible to attach different permissions to two threads trying to access the same address.

More precisely, three domains are maintained in protection keys: `UNSAFE` for the isolated data, `SAFE` for kernel-space memory and `USER` for user-space memory. Then, each of these MPKs changes permissions depending on what code is being executed. For example, upon entering an untrusted function call, protection key `SAFE` disables its access rights, and these are later re-established when returning.

What is important in this paper is that memory protection keys do have concrete applications in current systems, and their performance benefit is significant over prior work using `mprotect()`. Additionally, since the implementation has been done on Linux, the need for similar MPK management on other OSes like FreeBSD is justified. In other words, Intel MPKs are useful and it is important to make their manipulation portable across operating systems. This last part has been tackled first-hand in this semester project.

Chapter 7

Conclusion

When it comes to memory protection, the conventional OS solution is to use the `mprotect()` syscall, which inserts two protection bits in the page tables corresponding to the address and size that are being protected. However, this incurs many context switches to enter and leave the kernel, which is why Intel has been introducing hardware protection keys in its newest architectures. Moreover, support has been added on Linux and FreeBSD OSes, which is very practical for interacting with MPKs, but there is no standard interface for them yet.

As a result, the LibMPK paper [7] has introduced an API consisting of eight library functions, which enable not only key virtualization, but also metadata integrity and inter-thread synchronization of the PKRU register. This library is only implemented for Linux though, and this project tackled the challenge of porting it to FreeBSD. As such, the project consists in a set of library functions, as well as a kernel patch that creates new syscalls in FreeBSD which enable pkey allocation in a principled way.

When it comes to evaluating the implementation, the experiments have been run on a QEMU virtual machine of FreeBSD, as well as on a Linux environment with direct access to memory protection keys. They show that the LibMPK interface works for basic examples of programs and interacts widely with systems design.

Ultimately, is was very interesting to discover more in depth how an OS works under its cover and how the different layers of the system are able to communicate. Furthermore, I am planning to continue working on LibMPK and in particular the more advanced part on thread synchronization, which is not even fully functional in the original implementation yet.

Bibliography

[1] @0xAX. ***Synchronization primitives in the Linux kernel***. `https://0xax.gitbooks.io/linux-insides/content/SyncPrim/linux-sync-5.html`. 2021.

[2] Brooks Davis. ***Adding Syscalls To FreeBSD***. `https://wiki.freebsd.org/AddingSyscalls`. 2021.

[3] Butler Lampson. ***Hints and Principles for Computer System Design***. `https://arxiv.org/abs/2011.02455`, 2021.

[4] Wikipedia. ***FreeBSD***. `https://en.wikipedia.org/wiki/FreeBSD`. 2021.

[5] GitHub Issue. ***Problem occurs when using libmpk with multiple threads***. `https://github.com/sslab-gatech/libmpk/issues/1`. 2020.

[6] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. "**Intra-Unikernel Isolation with Intel Memory Protection Keys**". In: *VEE '20*. `https://www.ssrg.ece.vt.edu/papers/vee20-mpk.pdf` (2020).

[7] Soyeon Park Sangho Lee, Wen Xu Hyungon Moon, and Taesoo Kim. "`libmpk`**: Software Abstraction for Intel Memory Protection Keys (Intel MPK)**". In: *USENIX ATC '19*. `https://www.microsoft.com/en-us/research/uploads/prod/2019/05/park-libmpk.pdf` (2019).

[8] Park Soyeon, SSLab@GeorgiaTech. ***libmpk (a software abstraction for MPK)***. `https://github.com/sslab-gatech/libmpk`. 2019.

[9] 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: *28th USENIX Security Symposium*. `https://www.usenix.org/system/files/sec19-vahldiek-oberwagner_0.pdf` (2019).

[10] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. "**No Need to Hide: Protecting Safe Regions on Commodity Hardware**". In: *EuroSys '17*. `https://www.cs.ucy.ac.cy/~elathan/papers/eurosys17.pdf` (2017).

[11] 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: *USENIX OSDI '16*. `https://www.usenix.org/system/files/conference/osdi16/osdi16-litton.pdf` (2016).

[12] ***Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3***. `https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-system-programming-manual-325384.pdf`: Intel Corporation, 2016.

[13] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. ***Operating Systems: Three Easy Pieces***. `https://www.ostep.org`, 2015.

[14] Dave Hansen. ***[RFC] x86: Memory Protection Keys***. `https://lwn.net/Articles/643617/`. 2015.

[15] Bruce M Simpson. ***vm_map locking macros***. `https://manpages.debian.org/jessie/freebsd-manpages/vm_map_lock.9freebsd.en.html`. 2003.